ABSTRACT

Title of dissertation: HIGH PERFORMANCE XPATH EVALUATION IN XML STREAMS

Feng Peng, Doctor of Philosophy, 2006

Dissertation directed by: Professor Sudarshan S. Chawathe
Department of Computer Science

This thesis presents methods for efficiently evaluating structural queries over tree-structured data streams. A data stream usually consists of a sequence of items that arrive in an order determined by the source. An application that uses such data cannot revisit an earlier item in the stream unless it buffers the item itself. Naive buffering methods are not practical due to the high throughput and indefinite length of data streams. Compared with the flat, relational-like data model for data streams that has received recent attention, processing a tree-structured XML data stream poses additional challenges, since a data item cannot, in general, be interpreted without taking structural information into account.

In this thesis, we focus on the evaluation of XPath queries on streaming XML. As a W3C standard, XPath has become a core XML technology not only as a standalone query language but also as the foundation of XQuery and XSLT. Features such as subqueries and reverse axes make XPath a powerful query language but they also complicate XPath query processing. We present our work on XSQ, a streaming XPath query engine. Our methods are based on a novel segment-based

evaluation scheme. XSQ uses very little memory and is able to process unbounded and unsegmented streaming data because it does not build a DOM tree in memory. It also provides high throughput by only processing the relevant portions of the data and low response time by returning results as early as possible. XSQ is the first streaming system to support complex XPath features such as multiple predicates, closure axes, aggregations, reverse axes, and subqueries.

We also describe our work on XPaSS, an XPath-based publish-subscribe system that simultaneously evaluates a large number of XPath queries over XML streams. Unlike other similar systems that filter pre-segmented documents as results, XPaSS returns only the precisely delineated data specified by a user query. It uses a segment-sharing scheme instead of prefix- and suffix-sharing that are commonly used. In our experiments, XPaSS supports up to one million XPath subscriptions using a modest PC-class server, with a throughput comparable to that of the simpler filtering systems.

HIGH PERFORMANCE XPATH EVALUATION IN XML STREAMS

by

Feng Peng

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2006

Advisory Committee:

Professor Sudarshan S. Chawathe, Chair/Advisor
Professor Samrat Bhattacharjee
Professor Samir Khuller
Professor Nick Roussopoulos
Professor Martin Dresner

© Copyright by
Feng Peng
2006

DEDICATION

I would like to dedicate the dissertation to my family: my father, Zailiang Peng, my mother, Chunjin Liu, and my wife, Xin Lei. Without their love and support, there is no way I could have complished this.

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Chawathe. It is really fortunate for me to have the opportunity to work with Dr. Chawathe. Looking back at the last four years, I realize how much I have learned from him.

I would like to thank Dr. Samir and Dr. Bhattacharjee, for their continuous consideration and help during my study in Maryland.

I would like to thank Dr. Rossopoulos, who taught my first database course here. It definitely triggered my interests in database research and led my way into the area.

Last but not least, I would like to thank Dr. Dresner for taking the time to meet me and agree to be in my committee.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

This thesis presents our work on XPath evaluation in streaming XML data. We begin this chapter by describing the characteristics and sources of streaming XML data. The XPath query language is then briefly introduced using a simple example. After presenting the challenges faced by an XPath query engine that operates in a streaming environment, we outline the main contributions of our work. The chapter ends with a map of the rest of the thesis.

1.1 Streaming XML

The Extensible Markup Language (XML) has become a well-established data format and an increasing amount of information is becoming available in XML form [71]. An example XML document that describes the information of two books is illustrated in Figure 1.1. An **element** in an XML document is enclosed by its **start-tag** and **end-tag**, e.g., <pub> and </pub> for the `pub` element. The content of the element, listed between the start- and end-tag, may contain other nested elements or plain texts.

The term *streaming data* is used to describe data items that are available for reading only once and that are provided in a fixed order determined by the data source. Applications that use such data cannot seek forward or backward in the

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

Before further discussion, we briefly introduce the data model of XML here. More details can be found in Chapter 3.

XML data is usually modeled as an edge-labeled or node-labeled tree [1]. In the commonly used *Document Object Model (DOM)* [39], an XML document is modeled as a node-labeled tree. Figure 1.3 depicts the DOM tree of the XML document in Figure 1.1.

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

Figure 1.1: Example XML data

type	name	attributes
begin	DOC	
begin	pub	
begin	book	(id, "1")
begin	price	
text	price	(TEXT, "12")
end	price	
begin	title	
text	title	(TEXT, "First")
end	title	
begin	author	
text	author	(TEXT, "A")
end	author	
begin	price	(type, "discount")
text	price	(TEXT, "10.00")
end	price	
end	book	
begin	book	
...	...	...

Figure 1.2: Sequence of SAX events

For streaming XML data, building a DOM tree in memory is not usually desirable because the data may be unbounded. Further, we may not need all of the DOM tree to process the given query. Therefore, streaming XML data are better modeled using the SAX (Simple API of XML) model [53]. For each start- and end-tag of an element, a SAX-conformant parser generates, respectively, a **begin event** and an **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. Each chunk of text content enclosed by the start- and end-tag results in the SAX parser generating a *text* event.

Essentially, the sequence of the SAX events corresponds to a pre-order traversal of the DOM tree of the data in which the attribute nodes are combined with their parents. The SAX events generated by a SAX parser given the data of Figure 1.1 as input are shown in Figure 1.2.

1.2 Streaming XPath Evaluation

There have been a number of recent proposals on query languages for XML and XML-like data models [2, 27, 13, 22, 21, 10]. Of these proposals, XPath [21] and XQuery [10] have emerged as the standard recommendations that are likely to receive broad support. In this thesis, we focus on XPath. However, since XPath forms an important core of XQuery, the methods we describe are useful not only for XPath engines, but also for XQuery engines.

An XPath query consists a sequence of *location steps*, also called a *loca-*

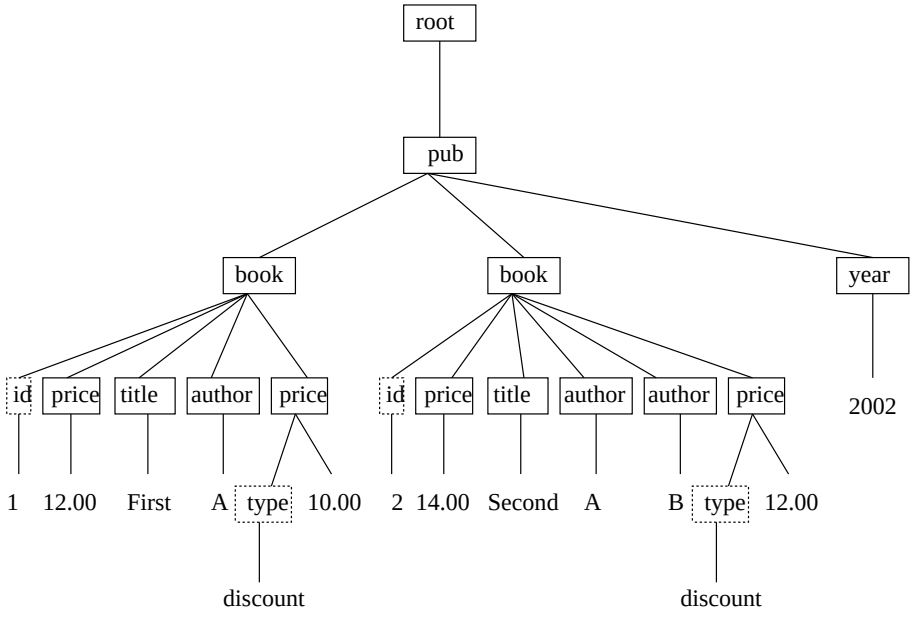


Figure 1.3: The DOM tree for the data in Figure 1.1

tion path, and an *output expression*. For example, the location path of the query `//pub[year > 2000]/book[price < 10]/title/text()` consists of three location steps: `//pub[year>2000]`, `/book[price<10]`, and `/title`. Each step contains an *axis*, a *node test*, and an optional *predicate*. For example, in the first step, "//" is the axis that denotes descendant-or-self relation, `pub` is the node test, and `[year > 2000]` is the predicate. This location path matches the title of a book if it is published later than year 2000 and its price is less than 10. The output expression, `text()`, indicates that the result consists of the text contents of titles matching the location path. Further details on XPath appear in Section 3.3.

We may think of the location steps as nested selection operators and the output expression as a projection operator. Each location step selects a set of XML elements out of the set selected by its previous step, according to the relation specified by the axis, the element name specified by the node test, and the conditions specified by the

predicates. The output expression then determines the parts, or functions, of those elements that are selected by the last step. While the projection operator in XPath is simple, the selection operators are fairly complex because they can be connected by complex navigations in the document tree and permit complex predicates on all elements that are selected by every step. For example, Figure 1.5 illustrates an XPath query that uses reverse axes, which allows backward navigation in the DOM tree, and boolean operators, which allows nested composition of predicates.

XPath is a succinct yet powerful path language that can be used to address parts of the XML documents. Given the hierarchical nature of XML data, XPath is the most natural method to retrieve desired data from XML documents. In many database systems that support XML, XPath is supported either as a standalone query language such as in the native XML databases [70, 40], or used together with SQL such as in the XML-extended relational databases [57, 41]. Moreover, it is a key component of other higher level XML query or transformation languages such as XQuery and XSLT [44]. These languages usually use XPath to select a subset from the document tree and apply higher level operations, such as transformation rules and joins, on the selected subset. As XPath being such an important component of these database systems and languages, efficient evaluation method of XPath queries will benefit their performance.

1.2.1 Join- and Navigation-based XPath Evaluation

Generally speaking, evaluation methods of XPath queries can be categorized into two main flavors: join-based and navigation-based. The prior 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 usually needs to parse the data, build the document tree, and navigate through the document tree every time it evaluates a path query. The navigation could be step-based, in which a location step is always evaluated on the result set of previous location step, or pattern-based, in which the pattern specified by the query is matched dynamically during the traversal of the document tree. 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. (We propose a new method in Chapter 7 that can perform pattern-based traversal even there are reverse axes specified in the path expression.)

Although join-based methods could be efficient of disk-residing data, it cannot fit in many scenarios, of which streaming data is an example, since it is expensive to perform join operations or build index for streaming data on-the-fly. The step-based navigation method is not suitable for streaming data either since we are only allowed to traverse the document tree in pre-order only once. In the sequel, we call the **streaming evaluation** as the evaluation diagram that has following features:

- Only one sequential pass of the data is required and no seeking-back in the

stream is allowed.

- No DOM tree is built in the main memory so that it can process possible infinite stream.
- The result should be available for the user as early as possible.
- Only the least amount of data are buffered, i.e., the potential results and data that are required to determine future results.

The first two features are usually mandate because of the nature of the streaming data. The last two features are performance-oriented and may be optional in cases where latency or memory footprint is not the main consideration.

1.2.2 Streaming Evaluation Scenarios

In may applications that process network-bound XML data, streaming evaluation of XPath queries is required, either because the data are in streaming form in nature, or because of performance considerations. Besides the widely used subscriber-publisher applications, we describe the applications of streaming XPath evaluation in the following three scenarios.

Scenario I:Personalized Web Content Delivery

An *information broker* accepts users' profiles specified in XPath queries and retrieves the target data for every user from heterogeneous data sources, such as news feeds, stock updates, Web pages, or differenced Web pages. Using XPath

```
<rss version="2.0" xmlns:dc="http://purl.org/dc/elements/1.1/">
  <channel>
    <title>XML.com</title>
    <link>http://www.xml.com/</link>
    <description>
        XML.com features a rich mix of information and services for
        the XML community.
    </description>
    <language>en-us</language>
    <item>
      <title>Normalizing XML, Part 2</title>
      <link>http://www.xml.com/pub/a/2002/12/04/normalizing.html</link>
      <description>
          Will Provost discusses when not to normalize, the scope of
          uniqueness and the fourth and fifth normal forms.
      </description>
      <dc:creator>Will Provost</dc:creator>
      <dc:date>2002-12-04</dc:date>
    </item>
  </channel>
</rss>
```

Figure 1.4: An example of an RSS2.0 feed

expressions to specify the profile instead of key words criteria has the benefit of being able to specify structural queries such as "*only prices for the stocks, not the merchandises*". Moreover, we can use an XPath query with a predicate to select "*stocks whose price is less than yesterday's*". Most news feeds used today are in RSS [62] format and each item inside an RSS feed (an XML document) represents a news item. An example new feed is shown in Figure 1.2.2. Usually user asks for the news that are related to certain topics, where we have to retrieve the single item out of the document, or even only the `description` element from that news item. For web pages written in XML (displayed using CSS or XSLT), it is also possible that user asks only for a part of the page. Since we can retrieve all the wanted nuts and bolts from the data sources, the broker can synthesize a personalized data digest for every user, even in a streaming form.

Scenario II:Web Service Requests

In Web service applications, a service request is embedded in a SOAP (Simple Object Access Protocol) [12] message, which is an ordinary XML document containing the information needed for the call of the requested service such as service name and parameters. Such information is retrieved from the SOAP message and used to invoke the requested service. In a server that needs to handle hundreds of thousands (or even more) of such service requests simultaneously, it is inefficient to start the service until the whole SOAP message (which could be as large as a few megabytes) has been received in full and loaded into main memory. If we can

use a streaming XPath engine to retrieve the parameters from the SOAP message while it is streaming in, we can start the service as soon as we get the initialization parameters the service needs. Moreover, those parts are discarded right away, and no main memory is needed to hold them. It is obvious that such an approach will lead to smaller response time and lighter service overhead.

Scenario III: Asynchronous JavaScript and XML

In Web applications such as online 3D geographical information systems that may require large amount data from the server yet small response time, the AJAX (Asynchronous JavaScript and XML) architecture is often used to provide fast response time and flexible user interface. For example, a GIS system that displays a 10-by-10 map may prefetch the data for a 12-by-12 map with the current focus region in the center. When the user wants to browse nearby region outside the current focus, those data are already in the local memory and the user interface can be updated almost in real-time. In the meanwhile, background script loads new surrounding data from the server.

An important component in the architecture is the DOM-based XPath evaluation subsystem. It allows the application selects from the data prefetched from the server in XML format. Currently AJAX implementations need to build the DOM-tree in the main memory and then fetch the desired data. Given a streaming XPath engine, those data can be read right away when they reach the local machine even the surrounding data is still in transfer. One of the benefits is shorter start-up

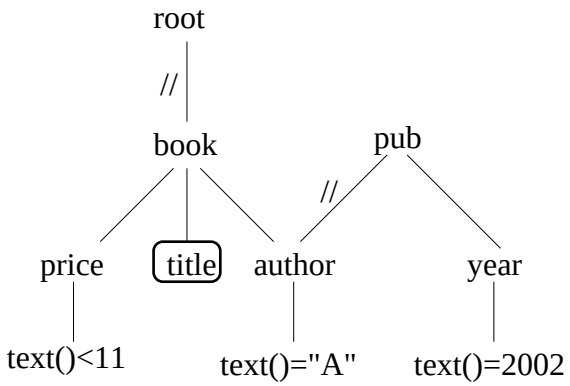
time for the application. Another benefit is a much smaller memory footprint since no DOM tree is built.

1.3 Challenges

Although its form looks like the regular expression, XPath is more powerful than the regular language since it can use arbitrary XPath queries in the predicate and boolean operators to connect those subqueries. Moreover, XPath provides a set of thirteen axes that allows the query specify very complex pattern to be matched with the document tree. Figure 1.5 shows a pattern specified by an XPath query, which uses the *ancestor* axis and therefore specifies a DAG (Directed Acyclic Graph) pattern. Since the predicates could be themselves complex patterns, traditional pattern matching algorithms can hardly be applied in XPath evaluation. Only recently did a polynomial algorithm for main-memory XPath evaluation is proposed in [32].

1.3.1 Querying Instead of Filtering

When the data are presented in streaming form, XPath queries are more difficult to evaluate. The intrinsic difficulty in streaming evaluation is that the potential result items may come before the data that are required to determine their memberships in the result. Therefore, we have to buffer the potential result items and keep track of different partial results for different buffer items. Given the complex pattern used in the predicate and potential multiple matchings between the elements and



Query:
```
//book[not(price<11) and author[text()="A"]
  /ancestor::pub[year=2002]]]
    /title
```

Note: 1. Boolean operators are not shown in the pattern.

2. The node with a box denotes the output node whose contents consist the result.

Figure 1.5: The DAG pattern specified by an XPath query

the query (as shown in Section 4.1), efficient management of these information is a challenge task.

Much of the previous work on processing streaming XML data focuses on *filtering* streaming XML documents using restricted XPath expressions [3, 25, 15, 35]. A filtering system returns a document identifier as the result of a query if *one* match of the query is found in the document.

There are several important difference between a filtering system and a querying system.

- A filtering system usually needs to assume that the streaming data are segmented into a collection of XML documents since they only return document identifiers as results. A querying system, in the contrary, may process unbound

and unsegmented streams and does not have this assumption.

- A filtering system needs to find one match for each query, and thus does not need to buffer the potential result whose membership in the final result cannot be decided according to the current available data. Consider the query in Figure 1.5, the filtering system needs only to find one such `title` element that satisfies the query, while a querying system needs to retrieve *all* the `title` elements that satisfy the query.

- A filtering system does not need to consider the complexities caused by multiple matchings between an element and the query. (The problem of multiple matchings is explained in detail in Section 4.1.)

1.3.2 Recursive Subqueries

Recursive subqueries being used as predicates, which means we can use several XPath queries connected by boolean operators as the predicate, further increases the complexities of the streaming XPath evaluation. Subqueries in predicates have different semantics than when it is used as a stand-alone query, since one instance that satisfy the subquery would evaluate the subquery to true while we have to return all the instances that match the stand-alone query. Only recently are the filtering systems able to process predicates in the queries [35, 25], even if the subquery can be treated the same as the stand-alone queries in the filtering setting (since both are essentially boolean queries).

Besides the different semantics, the *not()* function that can be used in the

predicates allows user to express universal quantification semantics in the predicate, while the predicate is usually using existential quantification semantics. In query used in Figure 1.5, if one of the book's price is less than 11, the predicate `not(price<11)` evaluates to false, i.e., if and only all the price children are larger than or equal to 11 does this predicate evaluate to true. Such universal semantics are more difficult to evaluate in streaming environment since we may need to test all the elements in an element set.

To address the complexities caused by the subqueries, we introduce in Chapter 6 a new method that marks the *useful* elements in the incoming streaming XML data with partial evaluation results. The process is guided by the pattern tree of the XPath query, and the potential result items and their partial predicate results are efficiently maintained. Unlike the alternating automata used in [35], which always wait until the end of an element to determine the result of its predicate, our method always keeps the most current predicate result for every buffer item, which means that the result can be sent to the user as early as they are available and there is not redundant content in the buffer at any time.

1.3.3 Reverse Axes

Reverse axes, such as *parent* and *ancestor*, also pose difficulties in streaming XPath evaluation since they are in nature requiring backward traverse in the document tree. Without reverse axes, an XPath query essentially specifies a tree pattern; with reverse axes in the query, an XPath query could specify a DAG pattern as we

show in Figure 1.5, which is more difficult to match in the document tree in the streaming evaluation.

To the best of our knowledge, the only streaming system that allows use reverse axes is XAOS [7], which essentially filters out the related elements in the stream that might be used in the evaluation and postpones the evaluation until the end of the document. Nonetheless, it does not handle the not() functions, and therefore does not need to handle the universal semantics.

We propose in Chapter 7 a new method that supports the reverse axes in the query by introducing a dynamic dependency graph, which keeps track of the relation between the potential result items and the undecided predicates. Whenever a pending result is evaluated, the dependency graph is updated and the new result items, if available, are sent to output.

To the best of our knowledge, XSQ is the first system that allows reverse axes in the query and evaluates the query in a true streaming manner, i.e., the evaluation is not postponed to the end of the stream, the result is always returned to the user as early as possible, and only least amount of data is buffered. As the experimental study in Section 7.4 shows, XSQ's performance is not only superior to most of currently publicly available XPath processor, but also not affected by the number of reverse axes in the query as the other systems since they may need multiple passes of the data for queries with reverse axes.

1.3.4 Multiple Query Evaluation

To the best of our knowledge, filtering systems today always assume that the streaming data are segmented into chunks beforehand. In order to query multiple queries over unlimited stream instead of filtering the pre-segmented documents, more complex mechanisms are required other than those used in filtering systems.

Consider a simple XPath query `//A[B]//C`. When it is used in a filtering system, the engine tests for every document the *existence of one* `C` element as a descendant of an `A` element who has a `B` child. While in a querying system, the engine returns for this query *all* such `C` elements. In the case some `C` descendants of an `A` element arrives earlier than the first `B` child of `A`, we need to *buffer* all those `C` descendants. Previous work has explored similar challenges in evaluating single XPath queries at a time.

The need to evaluate a large number of queries simultaneously makes for an even more challenging task. First, we need to identify the common features among all the queries that can be executed together and share those results for different queries. Previous work on filtering system addresses this problem by sharing common prefixed or suffixes among filters. However, when XPath expressions are used as queries, the same prefix of suffix may stand for different semantics. For example, in a filtering system, XPath expression `//A//B[//C]` and `//A[//B//C]` always return the same set of documents and share the prefix `//A//B//C`. However, in a querying system, the first query returns a set of `B` elements while the second returns a set of `A` elements. Therefore, extra mechanisms are required to handle these

differences. Second, we need new mechanisms to address the buffer operations for multiple queries, which are not at all required in filtering systems but essentially for querying systems. It is obvious that creating a single buffer for every query will not scale.

1.3.5 More Challenges

We have developed the XSQ system, which has been the first streaming system that supports the above features in XPath queries, and the XPASS system, which supports a large number of XPath queries simultaneously, there are still challenging tasks in streaming XPath evaluation. For example, a schema-aware runtime optimizer may further improve the performance of the systems. It is also important and challenging to support more complex query languages such as XQuery.

1.4 Contributions

In Chapter 4, 5, 6, and 7, we present our methods that *evaluates* single queries over XML streams. Figure 1.6 shows the screenshot of the XSQ system that is displaying the structure of the runtime engine. In Chapter 8, we present our XPath publisher-subscriber system that *returns the results* instead of document IDs. As the performance study in Section 4.4, 6.5, 7.4, and 8.4 shows, the new evaluation diagram is efficient in both CPU and memory usage. Moreover, they are the first set of methods that addresses the buffering problem in streaming XPath evaluation (not filtering) and supports the closures, multiple predicates, aggregations, subqueries,

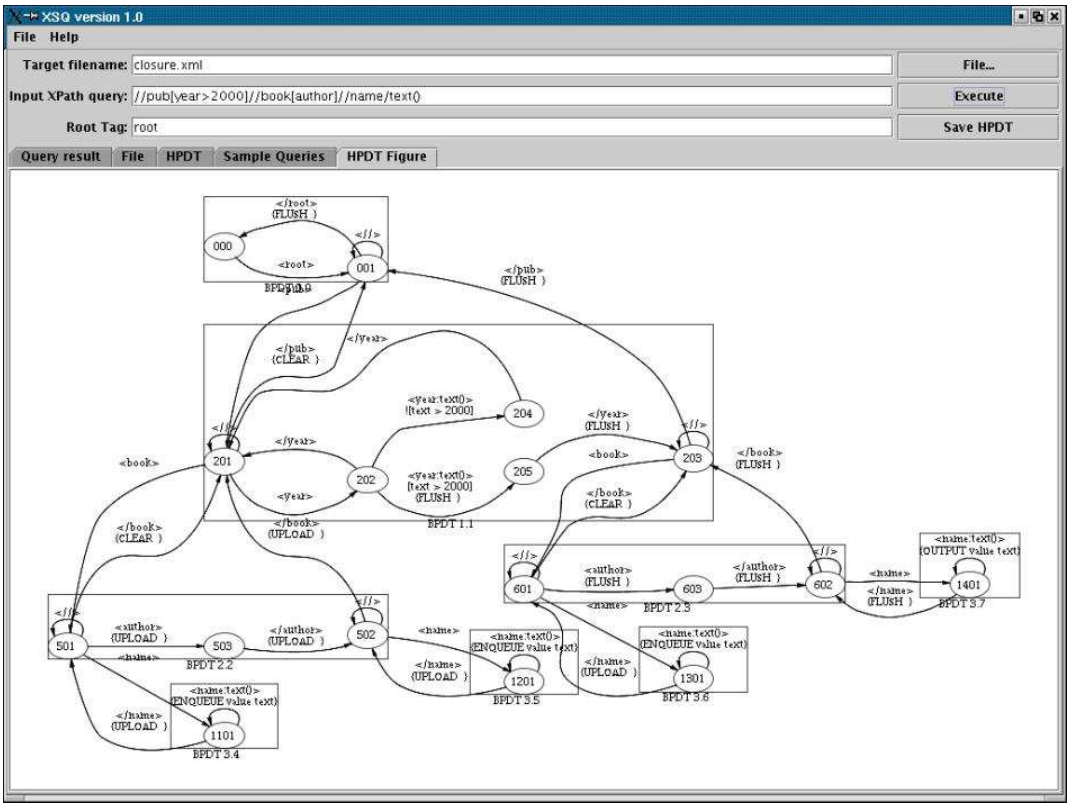


Figure 1.6: Screenshot of XSQ displaying a HPDT

reverse axes for XPath queries.

The major **contributions** of the XSQ and XPASS systems may be summarized as follows:

- To the best of our knowledge, our method for evaluating XPath queries over streaming data is the first one that handles closures, aggregations, multiple predicates, subqueries, and reverse axes (together). These features, especially in conjunction, pose significant implementation challenges.

- Our methods have a very clean design. The system is easy to understand, implement, and expand to more complex queries. Further, the methods provide a clean separation between high level design and lower-level implementation

techniques.

- To the best of our knowledge, the XPASS system is the first pub-sub system that supports XPath queries instead of filters. It shares the common segments among queries instead of the prefixes or suffixes, which are used by current filtering systems. The experiments show that the system has small processing time, uses only moderate amount of memory, and scales well to a million subscriptions.

- Our methods only buffer the least amount of the data that has to be buffered by any streaming system. They also guarantee that the results are returned to the user as early as possible. These features are important in a lot of streaming system since they usually lead to smaller response time and system overhead.

- We present a detailed experimental study of XSQ, XPASS, and almost all the related systems that are publicly available in Section 4.4, 6.5, 7.4, and 8.4. In addition to providing a comprehensive evaluation of the methods we propose, our study also illustrates the costs and benefits of different XPath features and implementation trade-offs as embodied by different systems.

- All the methods described in this proposal are fully implemented in the XSQ system and the XPASS system, which has been publicly released under the GNU GPL license [59, 31]. The Java-based implementation should work on any platform for which a Java virtual machine is available. In addition to serving as a testbed for further work on this topic, our system should be useful

to anyone building systems for languages that include XPath (e.g., XQuery, XSLT).

The rest of the thesis is organized as follows. Chapter 2 provides a comprehensive overview of the recent research work in related fields. Chapter 3 explains the XML data models and more details about XPath. In Chapter 4, we present the first version of XSQ system that handles closures, predicates, and aggregations. In Chapter 5, we describe a segment-based streaming evaluation scheme, which is used in the later chapters. In Chapter 6, we introduce the methods that handle complex subqueries. Chapter 7 introduces the methods that handle reverse axes. The XPASS system is introduced in Chapter 8. We then propose some future work in the area of streaming XML processing in Chapter 9. We conclude the work in Chapter 10.

Chapter 2

Related Work

The research of streaming XPath evaluation is connected with many related research areas. In this chapter, we first introduce a very similar topic of XPath filtering, in which we treat XPath queries as filters and use them to filter out the documents (instead of elements) that match the query from the stream. We then describe other available querying systems for XML streams, which are similar to ours, but use different query languages or pose different restrictions on the query language. We also compare our systems with them experimentally in Section 4.4, 6.5, 7.4, and 8.4. Other related work are then described: general XML query and transformation, theoretical results on XML query languages, and pattern matching. They are not in a streaming setting but should be able to provide some insights. Last, we introduce several general streaming management systems and streaming algorithms. Although not designed for XML data, some ideas in those areas should be useful in XML streaming processing as well.

2.1 XPath filtering

Many recent research on streaming XPath processing focus on filtering applications [3, 33, 25, 23, 47, 15, 35]. This problem has been referred to variously as *selective dissemination of information (SDI)*, *publisher-subscriber (pub-sub)*, and

query labeling. These filtering systems use XPath expressions as *filters* and focus on grouping large number of XPath filters to share the computation on common prefixes or suffixes. Unlike our querying system, the filtering systems do not have to handle the buffering problems since one item in the final result set satisfies the filter expression and the whole document (usually as a unique document identifier) is returned to the user.

Briefly, filtering systems assume that the input is a stream of segmented documents that are to be matched with a given set of queries. A query is said to match a document if evaluating the query on the document returns a non-empty result set. Since there is no output other than the identifiers of the documents matching each query, methods for filtering are simpler than those needed for querying. We may think of methods for filtering as starting points for the exploration of more general methods for querying. Filtering systems typically focus on supporting high throughput for a large number of queries using only a moderate amount of main memory.

The *XFilter* system [3] focuses on the problem of evaluating a large number of XPath filter expressions over every document in a stream of documents. Since the filter expressions are likely to have many common components, the automata are combined and indexed to yield an efficient filtering method. The *YFilter* system [25, 23] addresses a similar problem and uses a combined single automaton to evaluate all submitted filter expressions. It uses a run-time stack to track all the possible states for all the queries. Instead of the index used by XFilter, YFilter uses query identifiers in the states to denote the queries corresponding to the results. The

method described in [15] uses a data structure called *XTrie* instead of a flat table to index XPath queries based on common substrings among them.

Automaton-based methods spend a significant amount of time matching transitions to incoming events; as a result, deterministic automata typically yield higher throughput than their nondeterministic counterparts. However, as usual, the deterministic version of an automaton may require a large amount of memory. This problem is addressed in [33] by using a lazy deterministic finite state automaton. The main idea is to first build a naive finite-state automaton directly from the XPath expression. At run-time, the system adds new states as needed on the fly. Since it does not need to use a stack to keep track of all possible states, its throughput is improved. Although the deterministic automaton requires more memory than its nondeterministic counterparts, an upper bound on the size of DFA is provided.

The problem of *query labeling* is studied in [47]. The authors propose a *requirements index* as a dual to the traditional data index. A framework is provided to organize the index efficiently and to label the nodes in streaming XML documents with all the matched requirements in the index. The problem of validating XML streams using pushdown automata has been studied in [64]. Briefly, an XML document is said to be *valid* with respect to a given *Document Type Definition (DTD)* [71] if the document structure obeys the grammar specified in the DTD. This problem can also be considered as a filtering problem because the pushdown automaton can filter the documents that satisfy the DTD.

A filtering system based on alternating automata is proposed in [35]. It takes a bottom-up tree pattern matching algorithm to match the patterns specified by

XPath queries with the XML streams, which is efficient in the filtering systems, but not easy to be applied in the querying system that needs to return the required portion of the document. For example, XPush machine always matches an element with the node test in the last location step if its tag matches the node test, even if it does not match the pattern specified by the query.

The above filtering methods group queries using either common prefixes or common suffixes. In contrast, our method groups queries using common XPath segments (defined in Chapter 3), allowing us to better exploit common subexpressions across queries. (Our method always takes advantage of common prefixes and suffixes.) Further, in a filtering system, an XPath query is always boolean, meaning filter `A[B]/C` returns the same set of documents as filter `A[B and C]`. We only need to find one `A` element with a `C` child and a `B` child. In contrast, in a querying system that evaluates the first query, we need to, for the first query, keep all the `C` children of an `A` element before we see a `B` child of `A` and, for the second query, keep all the `A` elements until either we see its end tag or we see a `B` child *and* a `C` child. When the XPath query contains many predicates and closures axes, the evaluation requires even more book-keeping.

2.2 Streaming XML Queries

A transducer-based approach to evaluating *XQuery* queries on streaming data is presented in [51]. An XQuery query is decomposed into subexpressions and each subexpression is mapped to an *XML Stream Machine (XSM)*. An XSM consumes

the content of its input buffer and writes output to its output buffer, which may be the input buffer of another machine. This producer-consumer relationship of XSMs through their buffers results in a network of XSMs. This network is merged into a single XSM that can be optimized if the DTD for the input data is available. (In [55], a similar approach is used to evaluate regular path expressions with qualifiers over well-formed XML streams. It proposes a transducer network model called *SPEX*, in which each transducer is generated from a regular path expression construct. The output tape of one transducer forms the input tape of another.) The key differences between XSQ and XSM are as follows: First, XSQ supports XPath features such as aggregations, closures, and multiple predicates that are not supported by XSM. As described in earlier sections, these features, especially in combination, complicate query processing. Second, XSM supports constructors in XQuery expressions while XSQ supports only XPath (no constructors). XSQ uses this simplification to work with a simpler model of buffer interactions. Third, the combined, optimized XSM is quite complicated, making it difficult to group similar queries. In contrast, the method used in XSQ can be extended to support multiple queries, as illustrated in our XPASS systems. At the time of writing, the XSM system was not available for testing and it is therefore omitted from our study. However, we believe that XSQ and XSM are practical demonstrations of the trade-offs between query language expressiveness and system simplicity and efficiency (XPath vs. full XQuery).

In [7], the authors propose an *XAOS* system that handles reverse axes in XPath expressions in streaming environment. The XAOS system uses two data structures called *X-tree* and *X-dag* to filter out the irrelevant nodes in the document and

store only the relevant nodes in a *matching structure*. At the end of the document, 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()` functions) in the predicate. Our approach differs from the XAOS system since we focus on buffering least amount of data with small overhead. First, buffering least amount of data implies that an element is sent to output as soon as its membership in the result set is determined. Second, if we allow comparisons in the predicate, since we only need the result of the predicate, the XAOS approach that stores all the relevant elements becomes not suitable. Third, in evaluation of subqueries with boolean operators, we always shortcut the evaluation based on the boolean operator and the current results. 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 *TurboXPath* system [42] is a streaming XPath engine that supports FLWR expression introduced in XPath 2.0. An FLWR expression can specify multiple related path expressions and a RETURN phrase that constructs the result from the matched contents. TurboXPath builds a single query tree with multiple output nodes for all the specified patterns. Every output node in the query tree is associated with a buffer of potential result items. Every predicate node has a buffer of elements that are used to evaluate itself. A global *working array* stores the matched elements and maintains all the matching information. These data structures are updated when new data is streaming in.

A streaming XQuery query engine from BEA Systems [29] uses an iterator-based model for streaming evaluation. Every XPath expression in a query is decomposed into a sequence of XPath steps. The steps are evaluated against the stream based on the axis (`/` or `//`) used in the step. Optimizations based on type inference and schema information are applied in the evaluation. One of the optimizations, using the memoization technique [24], exploits the possibility of sharing common subqueries within and among XQuery queries. Currently, the system is not publicly available.

2.3 XML Querying and Transformation

Several systems provide methods for querying non-streaming XML data. *Galax* [28] is a full-fledged XQuery query engine. It implements almost all of the XML Query Data Model along with the type system and dynamic semantics of the XML Query Algebra. *XQEngine* [43] is a full-text search engine for XML documents that uses XQuery and XPath as its query language. XPath expressions and boolean combinations of keywords are used to query collections of XML documents. The engine creates a full-text index for every document before the document can be queried. It is difficult to adapt these systems for streaming data. Nevertheless, we use them in our experimental study for comparison purposes.

A topic closely related to XPath query processing is *XML transformation*. *XSLT* is a standard template-based language for transforming XML [44]. Since XSLT uses XPath to specify patterns in its rules, XSQ and other methods for

XPath processing have applications in XSLT processors. The popular implementation of XSLT in Saxon [45] is based on an in-memory materialization of the entire XML document and is therefore limited in the size of documents it can efficiently transform. By using a streaming XPath processor such as XSQ, we can design an XML transformation system that buffers only limited amount amounts of data.

The *STX* system takes a different, more procedural, approach to transforming streaming XML [9]. It uses templates to specify the operations that should be performed when data matching the template pattern is encountered. We may think of STX as a general-purpose event-driven programming environment that is not tailored to a specific query language. However, it may be used for XPath processing if we design a method for generating efficient STX templates from XPath queries. For example, if there are two predicates in an XPath query, we may create two variables in the program to store the current results of the predicates. When a predicate is evaluated, the corresponding variable is set to the result of the evaluation. We also need to specify explicitly when to reset the variables. We may then choose the right operation based on the current values of the variables. However, in this scheme, the positions of the elements have to satisfy the requirement that the predicate is evaluated before the target items. In general, it is not obvious how to generate STX templates equivalent to an XPath query in a systematic manner. However, this approach is an alternative to our automaton-based approach and would benefit from further attention.

2.4 Theoretical Researches on XPath

The streaming XPath evaluation problem we are considering in this proposal is essentially a tree pattern matching problem that is restricted to a single preorder traversal of the data tree (due to the streaming nature of the data). The pattern may include both parent-child edges and ancestor-descendant edges. The embedding of the pattern in the data tree does not need to preserve the ancestor-descendant relations in the data tree, which means two pattern nodes in different branches in the pattern tree may map to two nodes that one of them is an ancestor of the other one in the data.

XPath evaluation is a complex problem in its own right. Only recently a polynomial main memory algorithm to evaluate XPath queries is proposed in [32]. In [54], the authors point that XPath evaluation is essentially a different problem than the *classical tree pattern matching* [38] and *unordered tree inclusion* [46] problems. Most algorithms provided in these literatures are bottom-up algorithms which requires postorder traversal of the data tree. A top-down algorithm is provided in [38] for the classical tree pattern matching problem. The algorithm needs only preorder traversal of the data tree. However, since it allows only parent-child edges in the pattern and preserve the order of siblings in the pattern, the algorithm cannot be applied to the patterns that contain ancestor-descendant edges and do not imply orders between siblings in the patterns. The *query complexity* of XPath is addressed by [32], which provides a main-memory algorithm for evaluating XPath on non-streaming data that is polynomial in the size of the query (and data). The

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

Prior work [54] has noted that there are important differences between XPath evaluation and the classical problems of *tree pattern matching* [38, 18] and *unordered tree inclusion* [46]. In particular, the problem of unordered tree inclusion is NP-hard (by direct reduction from SAT), while XPath queries can be answered in polynomial time [32]. Intuitively, the reason the inclusion problem is harder than the XPath problem is that the former does not permit multiple nodes in the pattern tree to be mapped to the same node in the data tree.

Most of the algorithms for these problems require a postorder (bottom-up) traversal of the data trees and are thus unsuitable for streaming data that is provided in preorder. As an exception, an algorithm described for the classical tree pattern matching problem [38] needs only a preorder traversal of the data tree. However, it allows only parent-child (not descendant) edges in patterns and finds only matches for which the order of siblings in the data matches the their order in the pattern. In contrast, tree patterns corresponding to XPath queries include ancestor-descendant

edges (for the closure axis) and XPath semantics require that the sibling order in the pattern (order of nodes mentioned in predicates) be ignored.

2.5 Data Streams Management Systems

The *Aurora* system [14, 19, 72] is a data stream management system for monitoring applications, in which typical tasks include tracking the abnormalities among multiple streams, filtering specific target data for the user, and executing queries involving aggregations and joins. The Aurora system processes data streams using a large trigger network. The trigger, which is essentially a data-flow graph, is generated from the persistent queries provided by applications. The tuples in the results of these queries are created from the incoming streams and fed into the original application also in streaming form. The Aurora system provides a set of operators for an application to specify the persistent query and quality of service (QoS) requirements. At runtime, the Aurora system is optimized by using techniques such as load shedding (discarding data that requires a long time to process) and real-time scheduling.

The *Fjords* architecture [52] has been developed for managing multiple queries over the numerous data streams generated from sensors. Sensor data is generated in streaming form and the data rate is typically high and variable. The Fjords architecture is designed to maintain a high throughput for queries even when the data rate is unpredictable. It provides an efficient and adaptive infrastructure for more sophisticated query applications. The main components of the architecture

are the queuing system and the sensor proxies. The queues can function in either pull or push mode. They are the basic functional structures to route data between the operators in a query plan. Query operators may be adaptive, such as Eddies [5]. Each sensor has a sensor proxy that accepts queries and tries to simplify the queries for the sensor's processor. The proxy adjusts the sample rate of the sensor based on the queries and permits different users share data from the sensor. Such optimizations result in higher throughput and longer sensor battery life, since energy is conserved by avoiding unnecessary sampling.

The *NiagaraCQ* system is designed to efficiently support a large number of subscription queries expressed in *XML-QL* over distributed XML datasets [17]. It groups queries based on their signatures. Essentially, queries that have similar query structure by different constants are grouped and share the results of the subqueries representing the overlap among the queries. NiagaraCQ and XSQ work at different granularities of data. Although NiagaraCQ handles both change-based and timer-based continuous queries, the events it handles (such as changed remote XML file and activated timer) are at a high level. Therefore, it can use materialized data that is managed by a cache manager. In contrast, systems such as XSQ and XFilter respond to every event generated by a SAX-like parser. XSQ evaluates queries on streaming data, and the result is also in streaming form. These two granularities are complementary: One can combine the methods of NiagaraCQ for the larger granularity with the methods of XSQ for the finer granularity.

A related system, *WebCQ*, implements server-based Web page monitoring [50, 49]. Users use WebCQ's own query language to specify a sentinel, which is

essentially a request for monitoring the specified Web objects. The sentinel supports different kinds of objects, such as images and links in Web pages, different time intervals for change detection, and different kinds of notification mechanisms. Although both WebCQ and XSQ are event-driven systems, the events in WebCQ systems are specified by the user and are mostly timer-based. When a timer is activated, WebCQ visits the specified Web resource and pulls the content that will be compared with its stored version in the cache. XSQ, in contrast, is more like a push-based system that receives the data passively and returns the results continuously. Further, like NiagaraCQ, WebCQ also operates at a larger granularity than does XSQ.

Another system for processing data streams is *dQUOB* [61, 60]. It views the data streams as a relational database. Each event in the stream maps to a tuple in a relation that characterizes the stream. It uses SQL extended with *create-if-then* rules from *Starburst*'s active database query language [66]. The *create* clause specifies the name of the rule and the data source, the *if* clause contains a SQL query, and the *then* clause specifies an optional function that accepts the result of the SQL query for further processing (including serving as the input of another query). The dQUOB system can generate optimized query plans for the continuous queries presented in the system based on the relational model and allows user-specified adaptation for changes in data streams.

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.

2.6 Streaming Algorithms

Besides the new system architectures designed for streaming data management, current researches also address the requirement of developing more capable and efficient There algorithms for streaming data. Since it is usually not applicable to answer complex queries over streaming data precisely, techniques such as sampling and histograms are widely used [26, 30, 34].

The technique of sampling is widely used to solve problems such as *online aggregation* [37]. The purpose of online aggregation is to provide *enough* accurate result for aggregate queries where the precision is specified by the end users. The results are shown in a progressive manner along with the confidence interval of the changing result. A different and important performance metric of such algorithm is *minimum time to accuracy* since the user usually wants the result as accurate as specified in least amount of time. In [36], the authors extend the traditional blocking nested-loops and hash joins to a set of non-blocking *ripple join algorithms*, in which random tuples are fetched from both tables in the join and the inner and

outer relations in the join algorithm are interchanged continually. The technique of sampling is also studied in [16]. The approximation of the query result is considered as the *stratified sampling* optimization problem, in which the optimization goal is to select the samples to minimize the error for a given workload. Therefore, the known results and techniques for the *stratified sampling* problem can be utilized for query answer approximation. The methods proposed in [37, 36, 16], however, cannot be applied to streaming data directly since they need random access of the data. But we can see that the essence of the idea of sampling may lead to efficient online aggregation algorithms for streaming data.

Another approximation technique, the histograms, are also used in problems such computing *correlated aggregation* [30]. A correlated aggregation is an aggregation query that requires the result of another aggregation query. For example, query `count(price>avg(price))` asks for the number of `price` that is lower than the average price. It is clear that this category of query is difficult to evaluate over streams since the average of the `price` cannot be decided until all the data are processed. The authors classify the correlated aggregation based on the feature of the *independent aggregate* (i.e., the inner aggregate) and the semantics of the online aggregation (such as *landmark window* or *sliding window)*). The approach in [30] is based on histograms that can adapt to the interest of the query and variation of the statistics of the data.

The methods of using histograms to approximate streaming data for querying are further studied in [34], in which the authors provide an streaming algorithm that constructs histograms that support incremental maintenance and satisfy the

user-specified accuracy. In [69], the *position histograms* are used to estimate the size of the XML queries.

Most work on streaming data, including XSQ, assumes that the input consists of only the raw data. In this environment, certain limitations are unavoidable. For example, it is easy to devise XPath queries and sample inputs for which an unbounded amount of buffering is required for any XPath processor that produces exact results. An interesting alternative to this environment is one in which the input provides some assistance to the query processor by specifying constraints on forthcoming data or some other similar hints. For example, [65] describes a method for embedding *punctuations* in streaming data, facilitating the streaming evaluation of queries that include blocking operators such as *group by*. It should be interesting to use similar ideas for streaming XML to support XPath queries that include traversal axes such as *following*.

Chapter 3

Preliminaries

In this chapter, we provide brief descriptions of the DOM and SAX models for parsed XML, and of XPath. We focus on the features that are essential for understanding our methods presented in subsequent chapters and do not provide comprehensive descriptions, which may be found elsewhere [71, 39, 53, 21].

3.1 Data Model for XML

A static XML document is usually modeled as an edge-labeled or node-labeled tree [1]. In the commonly used *Document Object Model (DOM)* [39], an XML document is modeled as a node-labeled tree. Each element in the document is mapped to a subtree in the tree, whose root node is labeled with the tag of the element. Although an element E is mapped to a subtree of the DOM tree, it is convenient to refer to the root of this subtree as *node* E. The children of an element E are mapped to children of the node E that have *node type* of *element*. The attributes and text contents of element E are also mapped to children of node E, but with *node types Attribute* and *Text*, respectively. Figure 3.2 depicts the DOM tree of the XML document in Figure 3.1. In the figure, the nodes with dotted boxes are Attribute nodes and the nodes without boxes are Text nodes.

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

Figure 3.1: Sample XML Stream

3.2 Data Model for XML Streams

For streaming data, building a DOM tree in memory is not usually desirable because the data may be unbounded. Further, we may not need all of the DOM tree to process the given query. Therefore, we model the input XML stream as a sequence of **events**, modeled after SAX [53] events.

In our event-based model, each event e is a quadruple of the form (n, al, t, d): (1) The string n is the name of the element that generates the SAX event. (2) The list al contains pairs of the form (a, v), indicating that the element has attribute a with value v. Since elements are not permitted to have multiple attributes with the same name, the attribute name a uniquely identifies a pair in the list. We use the notation $e.a$ to refer to the value of the a attribute of element e; if e does not have an attribute a, $e.a$ is null. (3) The type t is B for a begin event, E for an end event, and T for a text event. Events of type E have an empty attribute list, while events of type T have an attribute list containing the single pair $(text(), v)$,

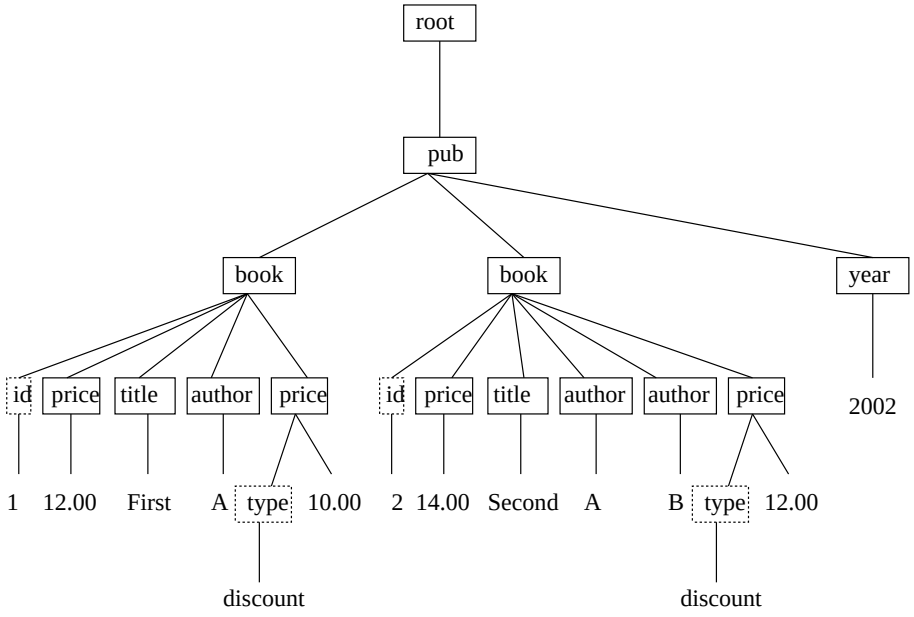


Figure 3.2: The DOM tree for the data in Figure 3.1

indicating that v is the text content of the element. (4) Finally, the integer d is the depth of the element in the document tree. The root of the document tree, also called *document root*, has depth 0. The attr and text nodes have the same depth as their parent nodes. Strictly speaking, SAX events do not include this depth component. Instead, this information is added by XSQ by wrapping SAX events and maintaining a depth counter internally.

A SAX parser generates a *start-document* event (S-DOC) when it begins parsing an XML document and an *end-document* event (E-DOC) when it finishes parsing the document. It is convenient to regard the S-DOC and E-DOC events as the begin and end events, respectively, of the document root.

Example 1 *Using the notation described above, we list below the first ten events generated by a SAX parser given the input of Figure 3.1.*

1. *(root, ϕ, B, 0): the begin event of root element.*

2. *(pub, ϕ, B, 1): the begin event of pub element.*

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

4. *(price, ϕ, B, 3): the begin event of price element.*

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

6. *(price, ϕ, E, 3): the end event of price element.*

7. *(name, ϕ, B, 3): the begin event of name element.*

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

9. *(name, ϕ, E, 3): the end event of name element.*

10. *(author, ϕ, B, 3): the begin event of author element.*

3.3 XPath

An XPath query is an expression of the form of $N_1N_2 \ldots N_k[/O]$, which consists of a **location path**, $N_1N_2 \ldots N_k$, and an optional **output function** O. Each **location step** N_i is of the form $/a_i::n_i[p_i]$ where a_i is an **axis**, n_i is a **node test** that specifies the name of elements N_i can match, and p_i is an optional **predicate** that is specified syntactically using square brackets.

3.3.1 XPath Semantics

In general, an XPath query is interpreted as follows: Each location step selects a set of nodes in the document tree. For every node x selected by N_{i-1}, N_i selects a set of nodes using x as the *context node*. The set of nodes selected by the last location step consists of the *result set* of the query. There is an implicit zeroth location step N_0 that always selects the document root. Thus, N_1 is always evaluated using the document root as the context node.

We now explain how a node y is selected by a location step N_i=$/a_i$::$n_i[p_i]$ from a context node x, which is selected by the location step N_{i-1} ($i > 0$).

Node test First, the tag of y has to match the node test n_i. A node test is either a string that specifies a matching element tag or a wildcard, denoted as *, that indicates any tag may match this node test.

Axis Next, the axis a_i specifies the relation between y and the context node x. In a simplified form, `/` is shorthand for the `/child::` axis, which specifies that y must be x's child. Similarly, `//` is shorthand for the `/descendant-or-self::node()/` axis, which specifies that y must be a descendant of x (not necessarily a proper descendant). If no axis is specified, the default axis is the child axis. However, if the axis before the first location step is omitted, the default axis is `//`, not the child axis. For example, expression `title/text()` returns the text content of all `title` *descendants* of the document root.

We process queries with the *parent* axis and the *ancestor* axis in Chapter 7. These axes are called **reverse axes** since they allow upward traverse in the docu-

ment tree.

Predicate If y satisfies the previous two conditions, y is selected by N_i if the predicates p_i evaluates to true *in the context of* y. In other words, if we treat y as the root, the XPath query p_i should evaluates to a non-empty result set. In many cases, the predicates are in simple forms such as `book[price<11]` and `book[@id]`. In the first query, the predicate `[price<11]` is true for a `book` element if it has a `price` child whose value is less than 11. In the second query, the predicate `[@id]` is true for a `book` element if it has an `id` attribute (the '@' symbol indicates that the following string specifies an attribute). We explain the form of the predicates in more detail in Section 3.3.2.

Output After the last location step N_k is evaluated, the output function O is applied to every node in the result set to produce the final output. The output function may specify an attribute or the text value of an element. It may also use an aggregation function such as `sum()` or `count()`. If no output expression is specified in the query, the elements in the result set are returned as the query result. Here are some examples:

- `title/text()` returns all the text nodes of the `title` elements.
- `book/@id` returns the `id` attribute of the `book` elements.
- `author/count()` returns the number of `author` elements.

3.3.2 Predicates in XPath

Each predicate P is also an XPath query, called a **subquery**. P is satisfied by an element e iff the subquery evaluates to a non-empty result set in the context of e, in which case we also say P returns true for e.

We can use boolean operators *AND* and *OR* to connect subqueries. The *NOT()* function can also be applied on a subquery. We can also *nested subqueries*, i.e. a predicate may itself contains predicates.

A top-level XPath query is evaluated in the context of the document root. A nested subquery, however, is evaluated in the context of the elements matched in the outer query, be it another subquery or the top-level query. Moreover, the result of a top-level query is a set of document nodes that match the last location step, while the result of a subquery is a boolean value that indicates whether its result set is empty.

If a predicate contains no value comparison, it tests the existence of specified object. For example, `book[price]` tests whether a `book` element has a `price` child. Predicates with value comparisons are evaluated as follows. First, when an element's attribute value or the text content a is compared with a literal v, XPath semantics specify that, if v is a number, a must be coerced to a numeric type. The comparison then proceeds with the usual numeric semantics. If the coercion fails, the predicate returns false. Second, a predicate such as `[price=10]` is interpreted as `[price/string()=10]`, where `price/string()` returns the aggregation of the text content within the `price` element. For ease of presentation, we

assume in this thesis that `string()` function be replaced by the `text()` function and that there is at most one text event for any element. (The `text()` function returns the set of text children of a node. For example, `string()` on a `price` element `<price>10<note>sale</note><price>` returns `10 sale`, while `text()` returns `10`.) However, our method easily supports `string()` and multiple text events within an event by buffering all the text events and delaying predicate-processing for an event to its end, after all text events have been encountered.

3.3.3 XPath Examples

The following queries, evaluated on the data of Figure 3.1, illustrate some of the key features of XPath.

- `//author/count()`: This query returns the number of `author` elements in the document. The first location step is `//author`, which consists of the closure axis `//`, and the node test `author`; it does not include a predicate. This location step matches all descendants of the document root that have tag `author`. The output expression, `count()` is applied to all qualifying elements to produce the query result. The result is 3 for the data of Figure 3.1. We note that this query may also be expressed as `author/count()` because a missing axis in the first location step defaults to closure.

- `//pub[book]//year`: This query returns the `year` elements that have `pub` ancestors that have at least one `book` subelement each. Here, the predicate of the first location step requires the existence of a `book` subelement. We note

that both location steps in this query use the closure axis. Further, there is no explicit output function, implying that the elements that match the location path constitute the query result. The result for the data of Figure 3.1 is `<year>2002</year>`.

- `//pub/book[@id > 1]/price[@type = "discount"]/text()`: This query returns the text contents of the price elements that have a type attribute with value `discount`. The price element must have a book parent, which in turn has a pub parent. The `id` attribute of the book element must be greater than 1. The result for the data of Figure 3.1 is `12.00`. Though the `id` attribute and the `discount` attribute are displayed both as strings in the document, the `id` attribute is compared using its numerical value since it is compared to a numerical value. If the value of an attribute cannot be converted to a number successfully, the operation returns false. (Such implicit type coercion semantics provide intuitive results on semistructured data and have been used in other languages, such as Lorel [2].)

Chapter 4

XPath Queries with Closures, Predicates, and Aggregations

In this chapter, we describe the first version of XSQ system, which is the first system that can evaluate XPath queries with closures, predicates, and aggregations (as a whole) over streaming XML data. Our implementation is based on using a hierarchical arrangement of pushdown transducers augmented with buffers. A notable feature of XSQ is that it buffers data for only as long as it must be buffered by any streaming XPath query engine. We present a detailed experimental study that characterizes the performance of XSQ and related systems, and illustrates the performance implications of XPath features such as closures.

4.1 Introduction

We focus in this chapter on XPath queries with (multiple) predicates, closures, and aggregations. The subset is described in Figure 4.2. We note that these XPath features are important usability advantages, especially if the data is semistructured or has a structure unknown to the query formulator. Closures, in particular, are indispensable in queries on data whose structure is partly unknown. For example, the query `//book[author = "Adams"]//price` returns the prices of books by Adams in a variety of likely structuring of bibliographic data, regardless of whether book occurs at the top level in the document or several levels deep and, similarly, regard-

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

Figure 4.1: Input Fragment 2

less of whether the price element is a child of the book element or a descendant separated by intervening `bookstore` elements. Similarly, predicates permit a more accurate delineation of the data of interest, leading to smaller, and more usable, results. The challenges posed by these features are exacerbated by data that has a recursive structure, as explained below. (A recent survey of 60 real datasets found 35 to be recursive [20].)

We then use some motivating examples to illustrate the complexities caused by closure axes and predicates in the streaming evaluation.

Example 2 *Consider the following query on the input fragment depicted in Figure 3.1:* `/pub[year > 2000]/book[price < 11]/author`*. Intuitively, it returns the* `authors` *of the* `books` *that have been published after* `year` *2000 and that have a* `price` *less than 11.*

When we encounter the first `author` *element on line 6 in the stream, it*

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

Figure 4.2: EBNF for an XPath Subset

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

When we encounter the `price` *element on line 7, we can check that it satisfies the predicate for its parent* `book` *element. However, we still cannot determine*

whether the `pub` *element on line 2 satisfies the predicate* `[year > 2000]`*. Consequently it is still unknown whether the* `author` *element on line 6 belongs to the result. Therefore, we must continue to buffer the* `author` *element and record the fact that the second predicate has been satisfied but not the first one. Similarly, the two* `author` *elements on lines 12 and 13, which belong to the second* `book` *element, have to be buffered as well. At this point in the stream (line 13) there are three* `author` *elements in the buffer: two with value A and one with value B.*

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

When we encounter the `year` *element on line 16, we may determine that the* `pub` *element on line 2 satisfies the predicate* `[year > 2000]`*. Recalling that this pub element is the ancestor of the* `author` *element remaining in the buffer, which has already satisfied the other predicate, we determine that this* `author` *should be sent to the output.*

The above example, although quite simple, illustrates some of the intricacies that we must handle. First, we may encounter items that are potentially in the result before we encounter the items required to evaluate their predicates. We need

to buffer such potential result items. Second, buffered items have to be distinguished so that, after the evaluation of a predicate, only the items that are affected by that predicate are processed. Third, in order to buffer items for the least amount of time possible, we need to check whether pending buffer items can be output as soon as some predicate is satisfied. Finally, predicates access different portions of the data. Some should be evaluated when the start-tag is encountered, while others may only be evaluated upon encountering the text content. (There are other forms of predicates, discussed later.)

Example 3 *Consider the query* `//pub[year>2000]//book[author]//name` *for the input fragment depicted in Figure 4.1. This example introduces some problems not seen in Example 2. Since the closure axis* `//` *is used in this query, an element and its descendants may match the same location step. For instance, the* `pub` *elements in lines 1 and 9 match the node test in the first location step. There are three ways in which the* `name` *in line 11 matches the pattern of the query (ignoring predicates). Each matching yields a different result for the predicates, as summarized in the following table.*

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

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

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

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

The XSQ system uses an automaton-based method to evaluate XPath queries over XML streams. The automaton, called an HPDT (Section 4.2.1), is a finite state automaton augmented with a buffer. For every input XPath query, we construct an HPDT hierarchically using a template-based method. Using the HPDT as a guide, a runtime engine (Section 4.3) responds to the incoming stream and emits the query result. The multiple matching problem (Example 3) is solved by associating with every buffer item its matching with the query and a flag the indicates the current predicate results. We note that the HPDT is used simply as convenient conceptual machinery to describe our methods. The expressiveness and theoretical complexity

of the automata are not our focus in this chapter.[1]

Organization: The rest of this chapter is organized as follows. Section 4.2 introduces how we compile an XPath query into an HPDT. In Section 4.3, we describe how the runtime engine processes the incoming stream using the HPDT as a guide. We also discuss the correctness and complexity of the method, along with a few key implementation details. Section 4.4 presents our experimental study of XSQ and related systems.

4.2 Compiling XPath Queries

In XSQ, an XPath query is first compiled into an *HPDT*, which is used by the runtime engine (Section 4.3) to evaluate the query on a streaming XML input. The HPDT is built in a layered manner with overlapping groups of states called *BPDT*s. We begin by describing HPDTs in Section 4.2.1. In Section 4.2.2, we describe the BPDT templates that form the basis of our method for building HPDTs. This method is described in detail in Section 4.2.3. Finally, Section 4.2.4 describes how aggregation functions are implemented in XSQ.

4.2.1 HPDT

The HPDT is a non-deterministic finite-state automaton augmented with a buffer. Its transitions are optionally associated with predicates and buffer opera-

[1] A brief description of our methods and the results of a preliminary experimental study of XSQ appear in [58].

tions. A transition is taken only if its predicate, if any, is satisfied. The buffer operation, if any, on a transition is executed when that transition is taken.

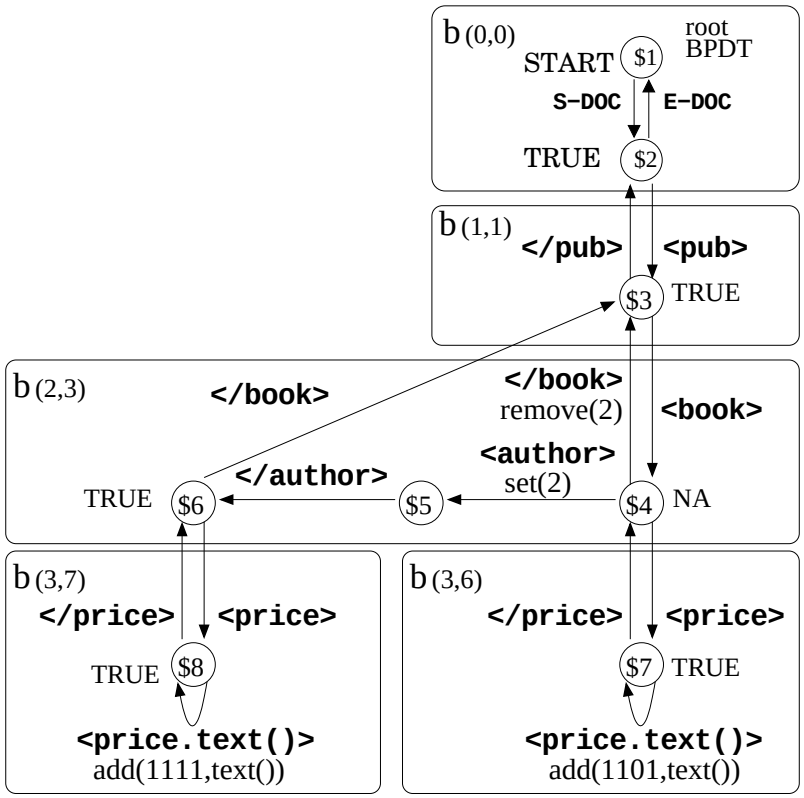
Transitions: The input to an HPDT is a sequence of SAX events, each of which takes the form (n, al, t, d), where n is the name, al is the attribute list, t is the type, and d is the depth. On transition arcs, we specify events as (n, t), where n specifies an element name and t specifies a SAX event type. Besides the three types (B, E, and T) introduced in Section 3.2, t can also be $\bar{*}$, a **catchall** type that matches all three types of events. A transition x with symbol (n, t) **matches** an input event e if n matches $e.n$ and t matches $e.t$. The attribute list al and depth d of an event are used in predicate evaluation and output composition, as described later. When a transition x emerging from a state matches the current event e in the input stream, we say this state **accepts** e. However, if x has a predicate then x is taken only if e satisfies the predicate, as described in Section 3.3. In the figures that depict state transition diagrams, we use an XML-like notation: `<n>` for (n, B), `</n>` for (n, E), and `<n.text()>` for (n, T). In our description, we use **Element(e)** to denote the XML element that generates event e. We use (e_i, B) to denote the begin event of element e_i, (e_i, E) to denote its end event, and (e_i, T) to denote its text event.

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

transitions, identified by the absence of special markings. These transitions differ in their effects on the runtime engine, as described in Section 4.3.

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

Buffer: The buffer of an HPDT is used to hold potential result items. We associate with each buffer item a $(k+1)$-bit flag, where k is the query length (number of location steps). The ith bit of the flag (counting from the left, starting with 0) denotes the current state of the predicate (perhaps trivial) of the ith location step: 1 for true and 0 for pending. Recall the zeroth location step always matches the document root and has no predicate. Thus, the zeroth bit of the flag is always 1 (We use $(k+1)$-bit flags instead of k-bit flags for better correspondence with BPDT identifiers, described in Section 4.2.3.) We use f_i to denote the ith bit of a flag f.



Note: HPDT for query `/pub/book[author]/price/text()`

Figure 4.3: A Sample HPDT

If all bits in a flag are 1, we say the flag is a *true flag*.

An HPDT uses buffer operations `set`, `remove`, and `add`. We describe these operations only informally here, deferring the details to our discussion of the runtime engine in Section 4.3. In that section, we also describe how the runtime engine applies `set` and `remove` operations selectively to only a subset of buffer items. However, for ease of presentation in the rest of this section, we assume that these operations apply to all items in the buffer. The *set(i)* operation sets (to 1) f_i for every buffer item. The *remove(i)* operation removes all buffer items having $f_i = 0$. The *add(f,a)* operation creates a buffer item with flag f using the *feature* a of the event e. The feature a may be an attribute name (including "`text()`"), in which

case $e.a$ is added. It may also be the catchall symbol $\bar{*}$, in which case the serialized (string) representation of e is appended, including all its attributes. For example, for the begin event (`book`, {(`id`,`"1"`)},B,1), the operation `add`($\bar{*}, f$) creates a buffer item that contains the string `<book id="1">` and has flag f. We do not use an explicit output operation. Rather, when the flag of a buffer item becomes a true flag (all 1s), the item is ready for output. The document order among the output items is preserved (as required by XPath) by using a global queue, as described in Section 4.3.4.

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

Example 4 *We can use the HPDT H depicted in Figure 4.3 to evaluate the query:* `/pub/book[author]/price/text()`. *We use rounded boxes to enclose the BPDTs, which are numbered using the scheme described in Section 4.2.3. All the transitions in H are regular transitions. Note that the* START *states of BPDTs $b(2,3)$, $b(3,6)$, and $b(3,7)$ are* TRUE *or* NA *states of other BPDTs (*TRUE *state of $b(1,1)$,* NA *state of $b(2,3)$, and* TRUE *state of $b(2,3)$, respectively). Such a shared state belongs to both the BPDT suggested by its enclosing box and the BPDT below it. Let us consider the first a few actions of H on the input fragment of Figure 3.1. After processing the begin event of the* `price` *element on line 4, state \$7 is active. The transition on the text event adds the text content,* 12.00*, of this* `price` *element to the buffer,*

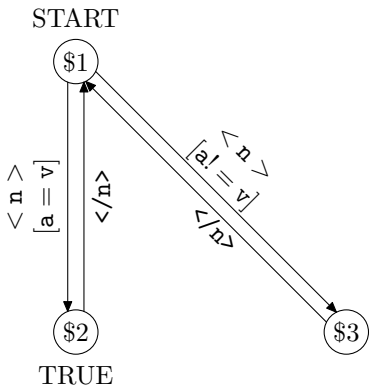


Figure 4.4: Template BPDT for: `/n[@a = v]`

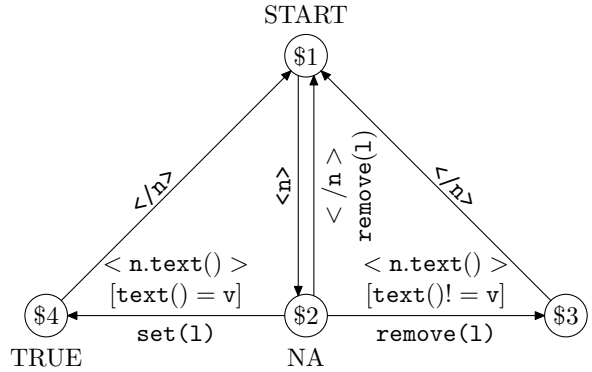


Figure 4.5: Template BPDT for: `/n[text() = v]`

with flag 1101. When H encounters the begin event of the **`author`** *element on line 6, it sets* f_2 *to 1 for the buffer items and transits to state $5 (and state $6 at the end event of this* **`author`** *element). Since the buffer item with value* 12.00 *now has its flag set to all 1's, it is emitted as output. When H encounters the next* **`price`** *element (line 7 of Figure 3.1), it transits to state $8. The transition from $8 on the text event results in the addition of* 10.00 *to the buffer, with flag 1111, which in turn causes* 10.00 *to be sent immediately to the output. (Since this* **`price`** *element's* **`book`** *parent has already satisfied the predicate* **`[author]`***, it should be immediately output.)*

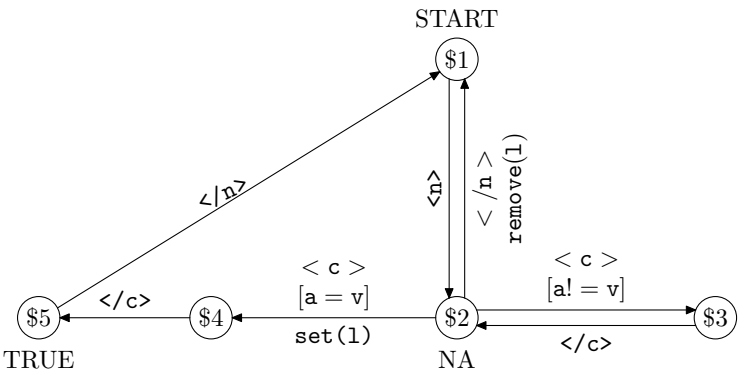


Figure 4.6: Template BPDT for: `/n[c@a = v]`

4.2.2 Templates for BPDT

We generalize the BPDT $b(2, 3)$ of Example 4 to the template depicted in Figure 4.7. We instantiate BPDTs from this template to evaluate location steps of the form of `/n[c]`. In general, we classify location steps into five categories for the purpose of template-based generation of BPDTs. In the following descriptions, we consider only the `/` axis. The modifications needed for the `//` axis are made separately after the templates are instantiated. During the instantiation of a template for a location step N_i, the parameter l used by the buffer operations in the template is replaced by i. The instantiation procedure is described further in Section 4.2.3. The design of these templates is guided by the existential semantics of XPath predicates. Once a predicate's result has been determined as true or false, the automata transit to states in which further data that could be used to evaluate the predicate is skipped. Buffered items are always processed, using the *set* or *remove* operations, at the earliest time that a predicate's result can be determined.

Template 1 Location steps of the form `/n`, `/n[@a]`, and `/n[@a op v]`, where `n` is an element name, `a` is an attribute name, `op` is one of the comparison operators

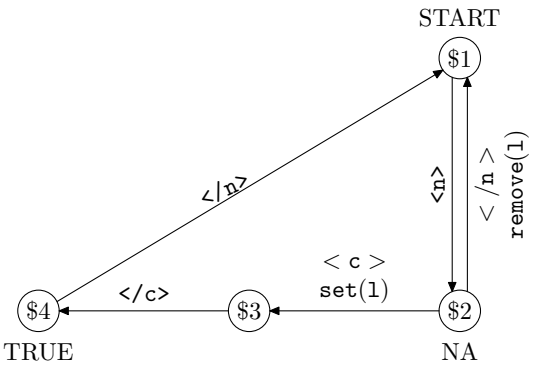


Figure 4.7: Template BPDT for: `/n[c]`

(Figure 4.2), and `v` is a literal: Figure 4.4 illustrates the template for `/n[@a = v]`. For `/n[@a]`, the test of the attribute value is replaced by a test for the existence of the attribute. For `/n`, state $3 and the transitions connected to it are not used and there is no test for the attribute. This template does not include an NA state because the result of the predicate is always known for each element as it is encountered. If the result is false, the BPDT enters state $3 that accepts nothing but the end event of the same element. Otherwise, the BPDT enters the TRUE state $2, which indicates that the predicate has been satisfied.

Template 2 Location steps of form `/n[text() op v]`, which include a predicate on the text content of matching elements: Figure 4.5 illustrates the template for `/n[text() = v]`. Since we assume that there is only one text event in each element, we compare the text event with the literal `v` only once. If the element `n` has no text contents (which can be determined only at the end of the element), the BPDT returns to the START state removing the buffer items that are waiting for this predicate. If the element `n` contains some text

content, the BPDT transits from state $2 to $4 if the content satisfies the condition, otherwise it transits from state $2 to $3. Once state $3 is active, it remains active until the end of this `n` element. State $2 is the NA state since the predicate is pending when it is active.

Template 3 Location steps of form `/n[c]`, which test the existence of `c`-children: Figure 4.7 illustrates the template for `/n[c]`. The template encodes the existential semantics of XPath predicates: After one `c`-child element of `n` satisfies the predicate, state $4 becomes active and no other `c`-child is tested. Only when the end of `n` is encountered and no `c`-child is encountered do we conclude that the predicate is false.

Template 4 Location steps of the form `/n[c@a]` and `/n[c@a op v]`, which include predicates that reference attributes of children. Figure 4.6 illustrates the template for `/n[c@a = v]`. For `/n[c@a]`, the test of the attribute value is replaced by a test for the existence of that attribute: This template encodes the existential semantics of predicates in a manner similar to that of the template for `/n[c]`. However, here a `c`-child may not satisfy the predicate, in which case state $3 becomes active and this `c`-child is ignored.

Template 5 Location steps of the form `/n[c op v]`, which include predicates that test the values of the child elements. Figure 4.8 illustrates the template for `/n[c = v]`. Recall, from Section 3.3, that the predicate `[c op v]` is interpreted as `[c/text() op v]`: This template is similar to that in Figure 4.7, but includes transitions to process the text events of `c`-children.

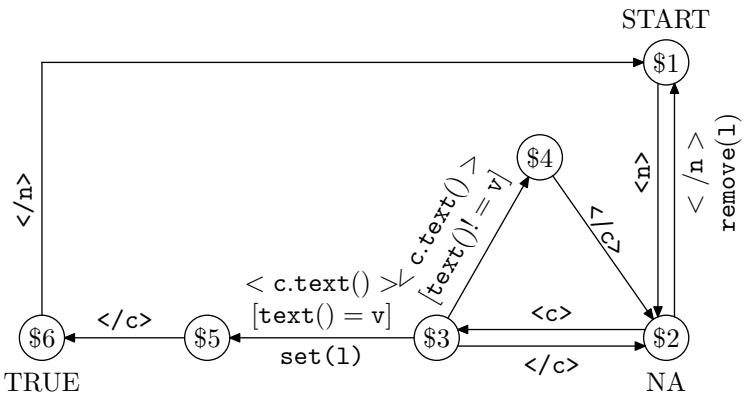


Figure 4.8: Template BPDT for: `/n[c=v]`

4.2.3 Building HPDTs from XPath Queries

Consider a query $Q = N_1 N_2 \ldots N_k$, where $N_i = /a_i :: n_i[p_i]$. The HPDT H for Q is is generated in a layered manner. Every BPDT is assigned a two-dimensional **identifier** (l, m) and is denoted as $b(l, m)$, where l is the layer and m is its position in the lth layer. We use the notation $b(x, y)$.START to denote the START state of BPDT $b(x, y)$ (and similarly for the TRUE and NA states). We first create the **root BPDT** $b(0, 0)$ (Figure 4.9) as the only BPDT in the zeroth layer. This BPDT does not depend on the XPath query and corresponds to the implicit zeroth location step of a query, which matches the document root. Its START state, denoted as s_0, is also the START state of the HPDT. Layer l, for $l \in [1, k]$ is generated as follows: For every BPDT $b(l-1, m)$ in the $(l-1)$th layer, we create a **child BPDT** $b(l, 2m+1)$, by instantiating the BPDT template that matches N_l. The TRUE state of $b(l-1, m)$ is merged with the START state of $b(l, 2m+1)$. If $b(l-1, m)$ has an NA state, we create another child BPDT, $b(l, 2m)$, by instantiating the template for N_l (again). The start STATE of $b(l, 2m)$ is merged with the NA state of $b(l-1, m)$. When instantiating a template, we set the parameter of the `set` and `remove` operations to

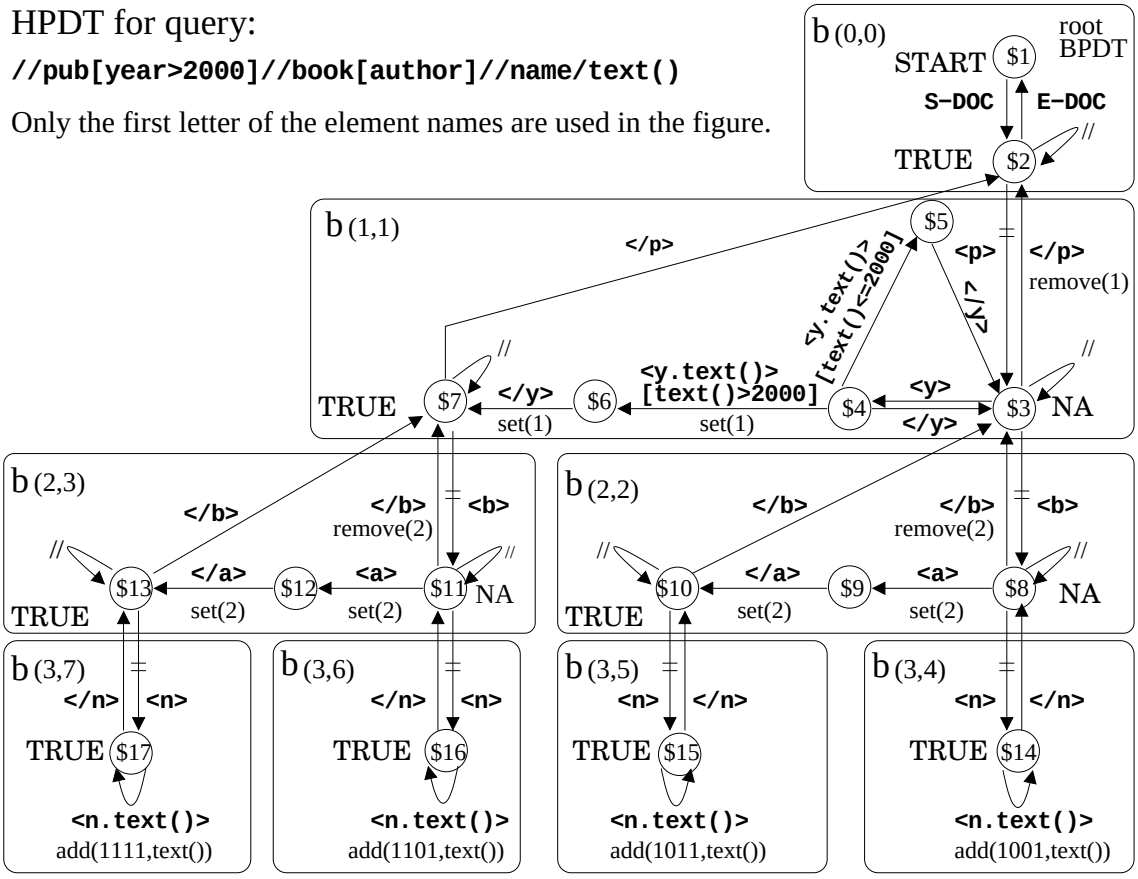


Figure 4.9: An HPDT Example

the layer number, l.

We summarize in Listing 1 the above procedure for creating an HPDT. The **AddBPDT(b,N,s)** procedure instantiates a BPDT using the template that matches location step N and sets the s state (either START or NA) of b as the START state of the new BPDT. After BPDT $b(l, m)$ is created, the **PostProcess** procedure is applied to it to perform the following three modifications. First, if a_l (the axis of location step N_l) is `//`, the procedure adds a *self-closure transition* from $b(l, m)$.START to itself, labeled `//`. We then use the **LocateTrans** function to locate all the transitions that emerge from the START state and match the begin event with name n_l

Algorithm 1: GenerateHPDT(Q)

```
   /* Build an HPDT from Q = N_1 N_2 ... N_k/O, where N_i = /a_i :: n_i[p_i]. */
 1 b(0,0) = CreateRootBPDT();
 2 for l ← 1 to k do
 3     for m ← 0 to 2^(l−1) − 1 do
 4         b ← b(l − 1, m);
 5         if b ≠ NULL then
 6             b(l, 2m + 1) ← AddBPDT(b, N_l, TRUE);
 7             PostProcess(b(l, 2m + 1), Q);
 8             if b.NA ≠ NULL then
 9                 b(l, 2m) ← AddBPDT(b, N_l, NA);
10                 PostProcess(b(l, 2m), Q );
```

(the node test of N_l). We mark them as *closure transitions*. (As discussed in Section 4.3, these newly marked transitions cause the HPDT to remain in $b(l, m)$.START in order to accept any descendants that also match N_l.)

Second, when a_{l+1} is `//` and p_l (the predicate of N_l) tests a child element, an extra `set` operation is added in $b(l, m)$ by the **AddExtraSet** procedure. This extra `set` operation is used to process descendants that are nested inside the child elements tested by p_l. This modification is needed only for BPDTs generated using the templates in the following Figures (with the affected transitions in parentheses): Figure 4.6 (\$4 → \$5), Figure 4.7 (\$3 → \$4), and Figure 4.8 (\$5 → \$6).

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

Algorithm 2: PostProcess(BPDT b, Query Q)

```
/*Modify b according to Q = N_1 N_2 ... N_k/O, where N_i = /a_i :: n_i[p_i] .*/
1 l ← b.layer;
2 if a_l = // then
3 |   NewTrans(b.START, b.START, "B", //, NULL, SELF-CLOSURE);
4 |   X ← LocateTrans(b.START, "B", n_l);
5 |   foreach x ∈ X do x.type ← CLOSURE ;
  /* Add an extra set operation if needed. */
6 if a_{l+1} = // then AddExtraSet(b);
  /* Add output to the lowest layer BPDTs. */;
7 if l = k then AddOutput(b, N_k, O);
```

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

The **initial flag** of the `add` operation in $b(k, m)$ is determined as follows. If p_k, the predicate of the n'th step N_k, is empty or p_k tests an attribute, the initial flag is always $2m + 1$. If p_k tests a child element or the text content then the initial flag is $2m + 1$ if the operation is on a transition whose source or target state is the TRUE state, and $2m$ otherwise. We see in Section 4.3.3 that such an initial flag correctly

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

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

encodes the current state of every predicate for the matching between the current element and the query.

4.2.4 Aggregations

In order to support aggregates in XPath queries, XSQ uses a statistics buffer called **stat**. This buffer is organized as a map and contains one entry for each aggregation function. The entry's key is the name of the aggregation function

and its initial value is *null*. There are two operations on this buffer: The first, `update(aggr)`, updates the entry for aggregation function *aggr* in *stat*. For example, `update(COUNT)` counts the number of buffer items with true flags and adds that number to *stat* entry for *count*; `update(SUM)` adds the numerical value of every buffer item with a true flag to the entry for *sum*. The second operation, `print(aggr)`, outputs the value of the *stat* entry for *aggr*.

For example, consider the following query, which differs from the query of Example 3 only in its use of output function `count()`:

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

To evaluate this query, we use an HPDT that is almost identical to the one depicted in Figure 4.9. We replace all occurrences of `set(`l`)` with `update(COUNT)`. The `add(`f, a`)` operation performs the update operation automatically if f is a true flag. We also place a `print(COUNT)` operation on the transition from \$2 to \$1 in the root BPDT.

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

4.3 Runtime Engine

The runtime engine maintains a set of *matching records*, which are described in Section 4.3.1 below. These records encode matching information and predicate re-

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

```
/* R is the set of matching records, each of the form (s, M).*/
1 for r ∈ R do
2     T ← LocateTrans(r.s, e.t, e.n);
3     for x ∈ T do
4         M' ← MatchDepth(r.M, e, x);
5         if M' ≠ NULL∧ Evaluate(x.predicate, e) = true then
6             R ← R + {(x.target, M')};
7             if x.type = REGULAR then R ← R − {r} ;
8             if x.op ≠ NULL then Execute(x.op, e, r.M) ;
```

sults for buffered items. Using the HPDT as a guide, the runtime engine responds to every input SAX event, updates the set of matching records, and executes the buffer operations. Buffer operations are described in Section 4.3.2. We discuss correctness in Section 4.3.3. We describe some implementation techniques in Section 4.3.4 and analyze our method's complexity in Section 4.3.5.

4.3.1 Matching Records

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

First, the engine uses the *LocateTrans* function to locate the set of transitions that emerge from state s and match event e. The engine performs no further op-

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

```
/* Returns the matching sequence M' of the target state. */
1  M' ← null;
2  switch x.type do
3      case SELF-CLOSURE: if e.t = B ∧ e.d > last(M).d then M' ← M;
4      case CLOSURE:
5          if e.t = B ∧ e.d > last(M).d then M' ← append(M, Element(e));
6      case CATCHALL:
7          if e.d > last(M).d then M' ← M;
8          if e.t = T ∧ e.d = last(M).d then M' ← M;
9      case REGULAR:
10         switch e.t do
11             case B: if e.d = last(M).d + 1 then M' ← append(M, Element(e));
12             case E: if e.d = last(M).d then M' ← removelast(M);
13             case T: if e.d = last(M).d then M' ← M;
14 return M';
```

eration for r if no such transitions exist. Next, for every matched transition x, the engine compares e with $r.M$ based on the type of the transition, $x.type$. The rules of the comparison are summarized in Listing 5. The `MatchDepth` function returns a new matching M'. If M' is empty, no further operation is performed for this transition, otherwise, the engine uses e to evaluate the predicate, if any, on transition x. If the predicate evaluates to false, no further operation is performed for x. Finally, a new matching record $r' = (s', M')$ is added to R, where s' is x's target state. If x is a regular transition, r is removed from R. If there is a buffer operation associated with transion x, it is executed as described in Section 4.3.2.

In the above scenario, when r' is added to R, we say that r **takes** the transition x on event e and **activates** r'. We also say that event e triggers the transition x and the runtime engine reaches state s'. We call a matching M **viable** if there is no

element e_i in M such that p_i is false before the `(`e_i`,B)` event. The method described in `EventHandler` and `MatchDepth` is motivated by the following two properties, which establish the relationship between matching records and matchings.

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

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

Property 2 For every $r = (s, M)$ in R with $M = (e_0, e_1, \ldots, e_i)$ $(i > 0)$, either

(1) s is a P-VALUE state of a BPDT in the ith layer and M is a matching between e_i and N_i or (2) s is a P-EVAL state of a BPDT in the $(i-1)$th layer and M is a matching between e_i and p_{i-1}. □

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

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

4.3.2 Buffer Operations

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

We now describe in more detail the buffer operations introduced in Section 4.2.1. Consider a buffer operation that is invoked when matching record $r = (s, M)$ activates matching record $r' = (s', M')$ on event e. Let $\texttt{max}(M, M')$ denote the longer one of M and M'. As before, the `last(M)` operation returns the last element in a matching M, while the `removelast(M)` function returns a new matching containing all but the last element of M.

- Operation **add(f,a)** creates a new buffer item whose content is the feature a of event e. The matching-flag pair associated with this item is $(\texttt{max}(M, M'), f)$.

- Operation **set(i)** sets f_i (the ith bit of the flag) for every buffer item whose matching contains the *target element* e_x. If s is an NA state, e_x is `last(`M`)`; otherwise, e_x is `last(removelast(`M`))`.

- Operation **remove(i)** removes all buffer items with $f_i = 0$ and a matching that contains the target element `last(`M`)`.

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

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

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

Example 5 *Consider the runtime engine, with the HPDT in Figure 4.9 for the query* `//pub[year > 2000]//book[author]//name/text()`*, operating on the stream of Figure 4.1. When the begin event of the* `name` *element on line 11 is encountered, there are three matching records with the state \$8, which accepts this begin event, and different matchings:*

M_1: document root, `pub` *on line 2,* `book` *on line 7*

M_2: document root, `pub` *on line 2,* `book` *on line 10*

M_3: document root, `pub` *on line 9,* `book` *on line 10*

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

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

On encountering the end event of the `book` *element on line 13, (\$10, M_2) and (\$10, M_3) both take the transition from \$10 to \$3. The two matching records (\$3, M_2^-) and (\$3, M_3^-) are already in R, since they activated (\$8, M_2) and (\$8, M_3)*

at the begin event of the `book` *element on line 10. They stayed in R because of the self-closure transition on $3.*

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

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

4.3.3 Correctness

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

flag). The following property is useful for this purpose.

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

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

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

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

Output ⇒ Result Suppose a buffered element e_k is associated with a true flag and a matching $M = (e_0, e_1, \ldots, e_k)$. We wish to show that (1) M is a matching between e_k and N_k and (2) e_i ($i \in [0,k]$) satisfies p_i, the predicate of N_i. Result (1) follows directly from Property 2 and an enumeration of the possible output functions and the corresponding translated operations. If the ith bit of the flag of e_k, f_i, is 1, either the ith bit of initial flag of the buffer item is already 1 or f_i is set by a `set(i)` operation. We have shown in Section 4.3.2 that `set(i)` set f_i for e_k (with matching M) only if M contains e_x and e_x has just satisfied p_i. Therefore, we only need to show that the ith bit of the initial flag for e_k (with matching M) is 1 only if e_i satisfies p_i. According to the definition of `add` operation, in BPDT $b(k,m)$, the initial flag is $2m+1$ if the transition on which the operation reside is connected to a TRUE state, or $2m$ otherwise. For the kth bit of the initial flag, it is 1 only if p_k evaluates to true for `last(`M`)` (otherwise the TRUE state would not be reached). For any other bit f_i, it is 1 if only if $m_i = 1$. According to Property 3, since the START state of $b(k,m)$ has been reached, m_i is 1 only if e_i satisfies p_i.

Result ⇒ Output Suppose an element e_k has a matching $M = (e_0, e_1, \ldots, e_k)$, e_i ($i \in [0,k]$) that satisfies p_i. We wish to show that there exists a buffer item created form e_k with matching M and a true flag. By Property 1, e_k is processed by the engine using (the transitions in) a kth layer BPDT and be added to the buffer. Let us consider the event e that evaluates p_i to true for e_i. If e_k is buffered before e, f_i for e will be set for e_k when e is processed. If e_k is buffered after e, there are two

cases. First, e_k is buffered before the TRUE state in the ith layer is reached (but e_i has satisfied p_i). This case could happen only if e_k is a descendant that is nested in c_i. The f_i is set for e_k by the extra `set` operation added by the `AddExtraSet` procedure. Second, e_k is buffered after the TRUE state in the ith layer is reached. By Property 3, upon encountering (`e`$_k$`,B`), the engine has a matching record with the START state of a BPDT $b(k, m)$, where $m_i = 1$. Therefore, e_K must be buffered by an `add` operation with an initial flag $2m$ or $2m + 1$, both of which have $f_i = 1$.

4.3.4 Implementation

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

the useful prefix of the matching. Thus, all the necessary operations on matchings can be performed on their depth-stack representations instead.

Depth stacks are stored as integers and operations on the depth stacks are implemented as bitwise operations on the integer representations. For example, if the depth stack is $(0, 1, 2, 5)$, the integer representation is 111001. That is, the i'th ($i >= 0$) bit is set if and only if the depth stack contains i. This representation is unambiguous because the depth stack consists of monotonically strictly increasing numbers (reading the stack bottom to top). Thus, the depth stacks use very little memory and operations on them incur very little overhead. We use long integers (64 bits) for this purpose. In order to support data with depth greater than 64, we can switch to using a pair of long integers.

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

Buffer Segmentation Consider buffer item b_1 with matching $(e_0, e_1, \ldots, e_i,$

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

4.3.5 Complexity

A detailed experimental study of the time and space efficiency of XSQ appears in Section 4.4. Below, we provide a very simple analysis of the construction-time and runtime complexity. For the construction-time complexity, we assume that the input query is in a parsed form and that string operations take unit time. The dominant factor in the construction is the number of BPDTs in the HPDT. For the runtime complexity, we assume that each depth-stack operation takes unit time. (See Section 4.3.4.) The function `remain`(ds, f) can be computed in constant time as follows. Since the right-most non-zero bit of every flag f can be computed in advance, given the depth stack ds stored as an integer, the `remain(`ds`, `f`)` function is a simple bitwise operation. Target groups in the buffer can be located using the hash table in constant expected time. Thus buffer operations, such as appending an item, deleting a group, and appending a group to another group, can be performed in constant expected time. Strictly speaking, some depth-stack operations, which are implemented using bitwise operations, and some buffer operations, which operate on only pointers in groups, may require non-constant time given arbitrary inputs. (For example, XSQ's implementation of depth stacks using constant-size integers does not work if the input XML has truly unbounded depth.)

Construction-Time Recall the construction of HPDTs summarized in Listing 1. The worst case occurs when every location step has a predicate. In this case, the construction creates 2^{k-1} BPDTs for an XPath query with k location steps. Creating a BPDT requires constant time for the tasks of finding the matching template,

initializing a constant number of states and transitions, and adding and changing a constant number of operations. (The number of templates, states, transitions, and the number of items to check for template matching, are all bounded by a small constant.) Therefore, the space and time cost of construction is bounded by $O(2^k)$. Although the exponential dependence on query length may seem problematic at first, the space cost of the HPDT is typically completely dwarfed by the space cost of buffering data at runtime.

Runtime Recall the runtime actions summarized in Listing 4. First, by examining the templates, we note that given a source state and an event, the `LocateTrans` function returns at most two transitions. Listing 5 suggests that the `MatchDepth` function also requires only constant time. The main determinant of the complexity is the number of matching records that need to be processed in the outer for-loop of Listing 4. If the query contains no `//` axis, the HPDT is free of closure and self-closure transitions. In this case, there is only one matching between a location path and an element. There can be only one or two (in the case of catchall transitions) matching records at any time. Therefore, each event can be processed in constant time. If the query contains `//` axes, there will be multiple matching records because of the closure and self-closure transitions. Since there are at most 2^{i-1} BPDTs generated to process the ith location step N_i, we have at most 2^i states (2 states in each BPDT) associated with a matching of length $i+1$. The number of ways that an element at depth d can match N_i is bounded by $\binom{d}{i}$. Therefore, the number of matching records (and thus the processing time per event) is bounded by

$\sum_{i=1}^{k} 2^i \binom{d}{i}$, where k is the query length and d is the maximum depth of an element. However, typical query-data combinations do not permit all the combinations for matchings assumed by the above calculation; thus the bound is not likely to be reached in practice. (See Section 4.4.)

4.4 Experimental Evaluation

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

We begin by describing our experimental setup in Section 4.4.1. We describe results on throughput in Section 4.4.2, latency in Section 4.4.3, and memory usage in Section 4.4.4. Section 4.4.5 presents a broader study of a set of query engines aimed at characterizing their features and performance. Section 4.4.6 presents an experimental characterization of XSQ.

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

Figure 4.10: System Features

4.4.1 Experimental Setup

In order to facilitate our experimental evaluation of the effects of different XPath features, we have implemented two versions of XSQ: **XSQ-NC** supports multiple predicates and aggregations, but not closures; **XSQ-F** supports closures in addition to multiple predicates and aggregations. The former implementation uses a deterministic automaton leading to performance benefits. We conducted our experiments on a PC-class machine with an Intel Pentium III 900 MHz processor with 1 GB of main memory running the Redhat 7.2 distribution of GNU/Linux (kernel 2.4.9). To ensure the evaluation is performed only in the main memory, the maximum amount of memory the Java Virtual Machine (JVM) could use was set to 512 MB. For the purpose of comparison, we selected a set of **systems** that process XPath or XPath-like queries. These systems are outlined in Figure 4.10. As the figure suggests, these systems vary considerably in their design goals and features. Many do not support streaming evaluation, and many are main memory systems that evaluate the query on the document tree of the data built in memory. We have

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

Figure 4.11: Dataset Descriptions

discussed XQEngine [43] (version 0.56) and XMLTK [4] (version 0.9) in Section 2.1. An implementation of STX, Joost [8] (version 20020828), and an implementation of XSLT, Saxon [45] (version 6.5.2), are also studied here. Some systems use query languages that are supersets or variations of XPath. For such systems, we issued queries that are equivalent to the XPath queries in our experiments. In many cases, the results are enclosed by different container elements but the contents are the same.

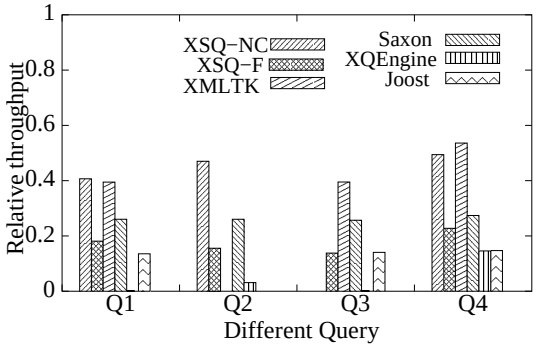
In our experiments, we use both real and synthetic **datasets** that differ in size and characteristics. We use four real datasets [4]: an XML-ized version of Shakespeare's plays (SHAKE); the NASA ADC XML dataset (NASA) [11], bibliographic records from the DBLP site (DBLP) [48], and the PIR-International Protein Sequence Database (PSD) [67]. Since these datasets have relatively shallow structures, we generated two synthetic datasets, *RECURS* and *RECURB*, using IBM's XML

Generator with deeper document structure to explore features related to such data. Some characteristics of these datasets are listed in Figure 4.11. In Section 4.4.5, We also use Toxgene [6] to generate synthetic datasets that contain specified number of designated elements.

To the best of our knowledge, there are no standard or widely used benchmarks for XPath queries. Therefore, following other work on this topic (e.g., [51, 7]), we conduct our experimental study using **queries** that vary in a variety of features that are likely to influence performance, such as query length, number of predicates, and types of axes. The queries used for each experiment are listed near the figures summarizing the results.

For a text-based data format such as XML, **parsing** the input typically accounts for a substantial fraction of the running time. The last two columns of Figure 4.11 list the parsing times for our sample datasets. We also note that parsing times vary widely across systems, depending on the parser and programming environment. In order to prevent these differences from masking the effects of query processing, we normalize the running time of each system using its parsing time.

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



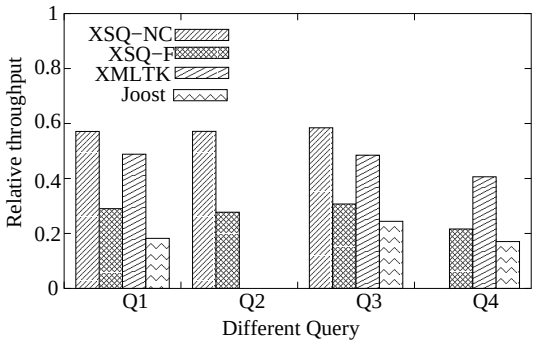
Q1: `/PLAY/ACT/SCENE/SPEECH/SPEAKER/text()`

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

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

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

Figure 4.12: Relative throughputs for different queries on the SHAKE dataset



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

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

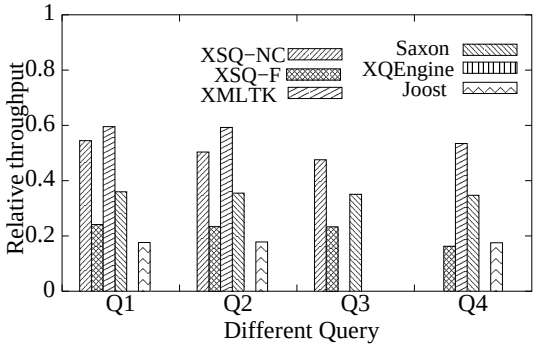
Q3: `/article@key`

Q4: `//title`

Figure 4.13: Relative throughputs for different queries on the DBLP dataset

4.4.2 Throughput

We measure throughput as the rate at which a streaming query engine consumes input data (megabytes per second). Since this rate may vary over time (perhaps depending on the structure of the data, or as a result of periodic reorganization of data structures in a streaming system), we measure the average throughput as the size of the input divided by the time required to process it. (For infinite streams, the average throughput at a point in the stream is obtained by dividing the amount



Q1:`/dataset/reference/source/other/title/text()`
Q2:`/dataset/title/text()`
Q3:`/dataset[altname@type="ADC"]/title/text()`
Q4:`//dataset//title/text()`

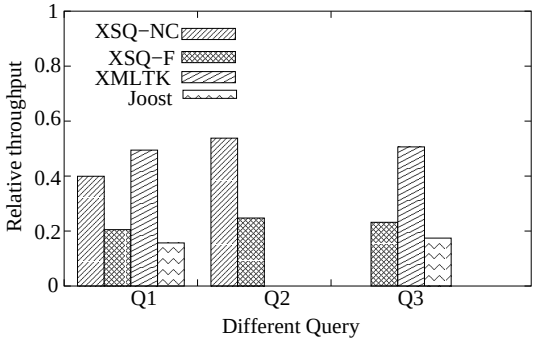
Figure 4.14: Relative throughputs for different queries on the NASA dataset

of data processed up to that point by the amount of processing time expended up to that point.)

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

Figures 4.12, 4.13, 4.14, 4.15, 4.16, and 4.17 summarize our experiments comparing the relative throughputs of the systems over different datasets and queries. Results for several combinations of queries and datasets are missing for one or more systems because either the system does not support queries with certain features (e.g., closures, predicates) or the dataset is too large for the implementation.

We observe that, in general, XMLTK and XSQ-NC are the two fastest systems when we use simple queries that they support. XMLTK supports only predicates that can be evaluated at the time a potential result element is encountered, such as a predicate on that element's attribute. Therefore, XMLTK can always output

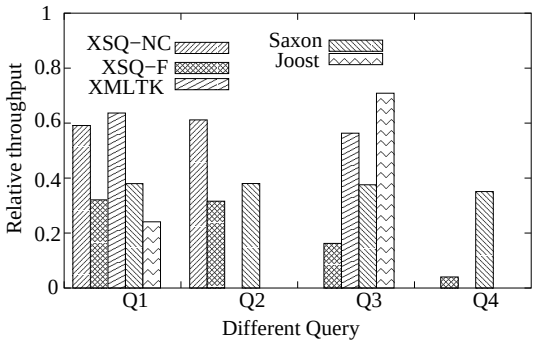


Q1:`/ProteinEntry/reference/refinfo/authors/author/text()`
Q2:`/ProteinEntry[sequence]/protein/name/text()`
Q3:`//sequence`

Figure 4.15: Relative throughputs for different queries on the PSD dataset

a result item at the time it appears in the stream. The HPDT used in XSQ-NC is deterministic, which means there is only one active matching record and at most one matching transition for the incoming event. It is one reason that XSQ-NC is faster than XSQ-F since XSQ-F may have multiple active matching records and multiple matching transitions for an incoming event.

However, even for the same query without closure, XSQ-NC is faster than XSQ-F although XSQ-F also has only one active matching record in this case. One reason is that XSQ-NC does not need mechanisms such as depth stacks to keep track of possible multiple matchings. Moreover, XSQ-F always buffers a potential result item b first even if b is known to be in the result when it comes in. XSQ-F then marks b as output, checks the queue, and outputs b if b is at the head of the queue. This mechanism is used only in XSQ-F since, due to the existence of closure axes, there may be other undecided items in the buffer before the current buffer item. Without closure axes, if we can determine b is in the result, we can always output b right away. In this case, we conclude that every b's ancestor matches a



Q1:`/pub/book/title/text()`
Q2:`/pub/book[year]/author[email]/name/firstname/text()`
Q3:`//proceedings/@category`
Q4:`//pub[year=14]//paper[@id=13]/title`

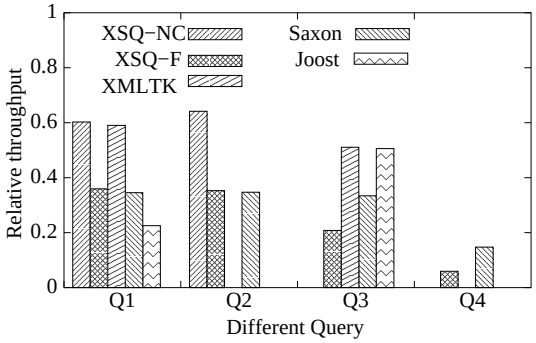
Figure 4.16: Relative throughputs for different queries on the RECURS dataset

location step in the query and satisfies the predicate. Therefore, there cannot be any undecided buffered items, since a buffered item can only wait for an open element whose predicate is pending. We study these issues further in Section 4.4.5.

Figures 4.12, 4.14, and 4.16 suggest that Saxon is faster than XSQ-F when they process XML data that can fit into main memory. Saxon loads all the data into the memory to build the DOM-tree before it evaluates the query. After parsing the data, Saxon performs all the processing in main memory. Such in-memory processing is efficient and can support more complex features such as the whole set of XPath axes. However, the main memory approach is not suitable for streaming data in general.

4.4.3 Latency

Output latency is an important property of streaming systems, and we measure it as follows. We let every system output the result to standard output. For a query that returns an element with name `N`, we monitor the standard output to detect



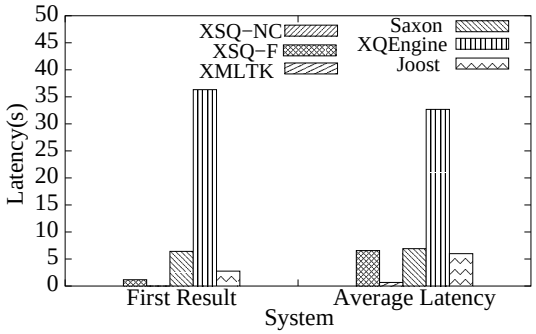
Q1:`/pub/book/title/text()`

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

Q3:`//proceedings/@category`

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

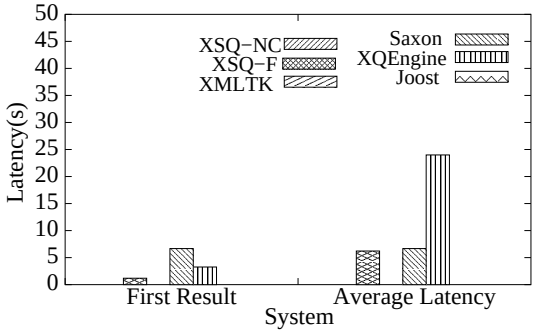
Figure 4.17: Relative throughputs for different queries on the RECURB dataset



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

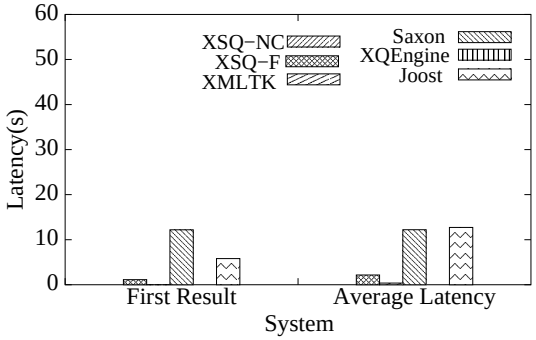
Note: For XQEngine, the average latency bar has been scaled down by a factor of 10.

Figure 4.18: Latency on the SHAKE dataset



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

Figure 4.19: Latency on the SHAKE dataset

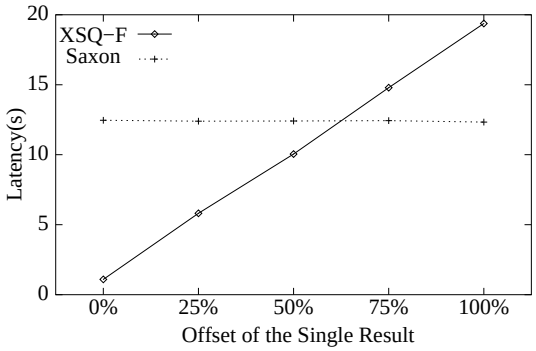


Query: `/datasets/dataset/reference/source/other/title`

Figure 4.20: Latency on the NASA dataset

the start-tag `<N>` and record the elapsed time when we receive each such tag. (The clock is started at the time the system begins evaluation.) For each result item, we refer to this time as its latency and define the latency of the query result to be the average latency for all items in the result.

Figure 4.18, 4.19, 4.20, and 4.21 summarize our results on the output latency. In the first three figures, the left parts illustrate the time when the first result elements are returned. The right parts of the figures illustrate the average output latency of result elements. We note from Figure 4.18 that the streaming systems usually output the first result item earlier than the non-streaming systems. (XSQ-NC and XMLTK returned the first element immediately after the systems were invoked.) This result is as expected since the non-streaming systems need to load all the data and build the document tree in memory before actual query evaluation begins. The average latency for the non-streaming system Saxon is very close to its latencies for the first returned element. The reason is that it always evaluates the whole query first and then returns the result when the whole result set is available. Since the XQEngine version we tested cannot handle documents with more than



Query:`/datasets/dataset[altname=x]/title`

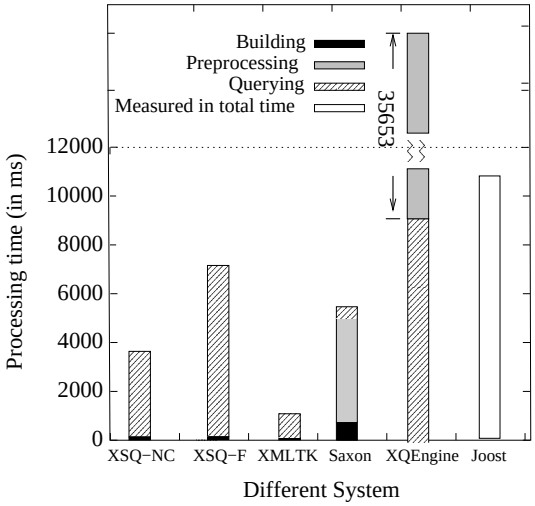
Note: The query returns a single result item at the offqset in the document that we selected using different value of `x`.

Figure 4.21: Latency on the NASA dataset

32,767 elements, we divided datasets into a sequence of smaller documents as needed to satisfy this constraint. Therefore, XQEngine returned the first result item after it finished processing the first small document. Also, we note that the bar depicting average latency for XQEngine has been scaled down 10-fold in order to fit in the chart.

In Figure 4.19, we used a query that contains a predicate testing whether the text content contains a string. Besides results similar to those in Figure 4.18, we notice that XQEngine returns the first result very quickly and that its average latency is also lower than that in Figure 4.18. This result is explained by recalling that XQEngine builds a full-text index for the XML document, and can therefore efficiently evaluate queries that require string lookups of this kind.

We used the NASA dataset for the next set of experiments. Figure 4.20 illustrates that for this larger (23MB) dataset, the latencies of the first result items in the streaming systems are much smaller than those in the non-streaming systems. This result is as expected since the non-streaming systems now need to load a larger



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

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

Figure 4.22: Preprocessing time, query processing time, and total querying time

dataset before they output the first result item. We also observe that the average latencies of XSQ and XMLTK are much smaller than those of the non-streaming systems, while the average latency of Joost is still almost the same as that of Saxon. After examining the result, we discovered that Joost uses buffered output. Since the result size of this query is twice the buffer size, the result items are emitted in two groups.

We note that the non-streaming systems may return results faster. In Figure 4.21, we used a query that returns a single element. By selecting the element at different positions in the stream, we observe that the latency for XSQ is almost proportional to the size of data before the result element. In contrast, Saxon's latency is almost constant since the position of the element is not important for its main-memory query evaluation.

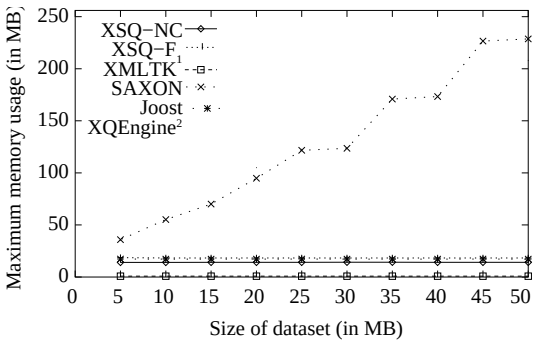
Figure 4.22 illustrates the result of measuring the components of the overall query-processing time. Although the figure depicts the result for one query, the

results are similar for other queries we used. The dark bar represents the query compilation time, which usually includes parsing the query and building the data structures used by the runtime query engine. The gray bar represents the preprocessing time. For example, the preprocessing stage of Saxon loads all the data into memory to build the DOM-tree before it can evaluate the queries. Similarly, XQEngine preprocesses data by building a full-text index on the data before evaluating any queries.

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

4.4.4 Memory Usage

The main memory required by a streaming query engine is an important metric and often determines its feasibility for an application. Figures 4.23, 4.24, 4.25, 4.26, 4.27, and 4.28 summarize the results of our experiments comparing the memory usage. We observe that, as expected, the streaming systems typically use much less memory than the non-streaming systems. We also note that, for different datasets, the streaming systems use almost the same amount of memory. This fact suggests that the amount of memory used by the streaming systems is only weakly dependent on the size of the datasets. For systems such as XMLTK and Joost, this observation is always true since no data is buffered during the evaluation. However, systems that support predicates, such as XSQ-NC and XSQ-F must buffer data and the

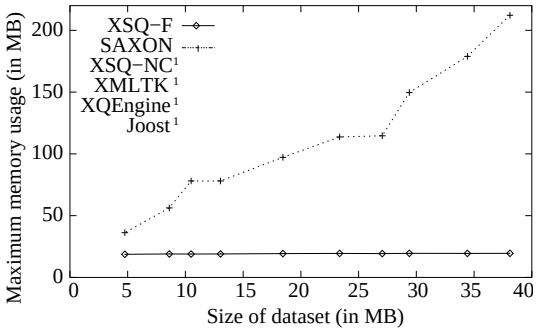


Query:`/dblp/inproceedings[author]/title/text()`

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

2. XQEngine could not be tested because it currently supports only 32K elements per file.

Figure 4.23: Memory usage for DBLP-based datasets of different sizes



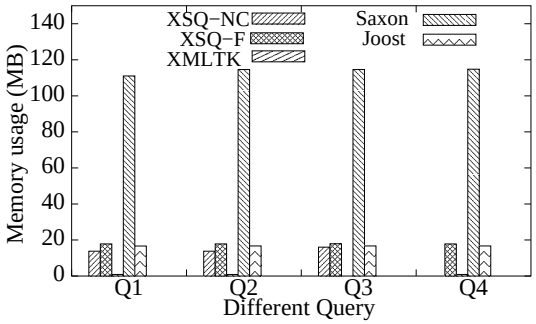
Query:`//pub[year]//book[@id]/title/text()`

Note: The system cannot handle the query in the dataset.

Figure 4.24: Memory usage for synthetic datasets of different sizes

amount of buffered data may be large, depending on the dataset and query. Further experiments studying this aspect of XSQ are described in Section 4.4.6.

We also used the XML Generator program to generate datasets of varying size and recursiveness. For example, for the dataset of size 13 MB, the nested level parameter of the XML Generator program is set to 15 and the maximum repeats parameter is set to 20. From Figure 4.24 we note that even with highly recursive data and queries with closures, the memory used by XSQ-F is almost constant. Since all the items in the buffers can be determined when we encounter the end event of the element matching the first location step, the maximum amount of memory that



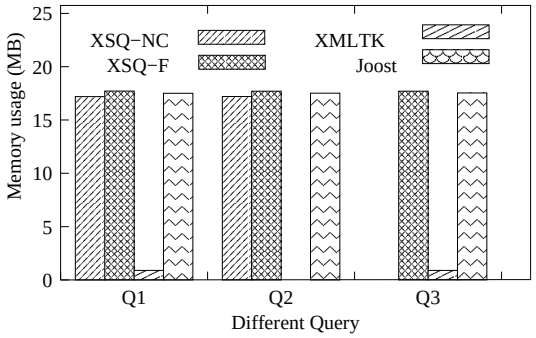
Q1:`/dataset/reference/source/other/title/text()`

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

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

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

Figure 4.25: Memory usage for different queries on the NASA dataset

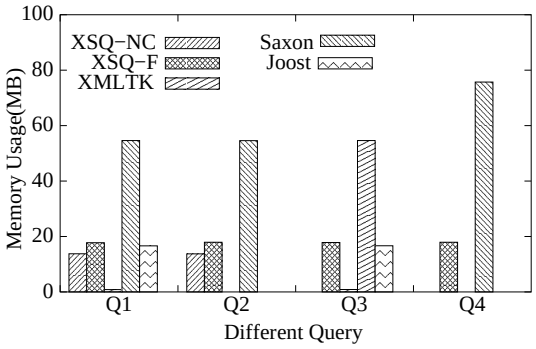


Q1:`/ProteinEntry/reference/refinfo/authors/author/text()`

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

Q3:`//sequence`

Figure 4.26: Memory usage for different queries on the PSD dataset



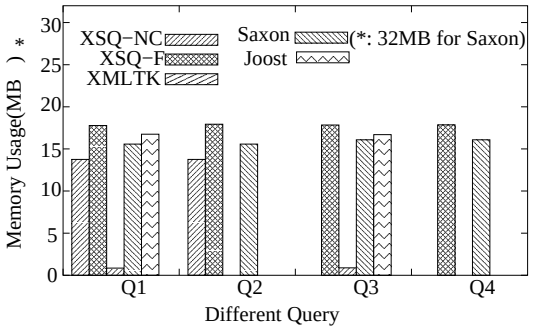
Q1:`/pub/book/title/text()`

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

Q3:`//proceedings/@category`

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

Figure 4.27: Memory usage for different queries on the RECURS dataset



Q1:`/pub/book/title/text()`
Q2:`/pub/book[year]/author[email]/name/firstname/text()`
Q3:`//proceedings/@category`
Q4:`//pub[year=14]//paper[@id=13]/title`

Figure 4.28: Memory for different queries on the RECURB dataset

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

Figure 4.29: Toxgene template

XSQ needs does not exceed the size of the largest element in the stream.

4.4.5 Characterizing the XPath Processors

Since streaming query engines need to buffer potential results items, the relative ordering of XML elements in a dataset may influence the amount of buffer space needed. To study the effect of element order, we generated a 10 MB dataset using Toxgene, by applying the template of Figure 4.29 repeatedly to generate new elements with successive `id` attributes. The result dataset contains 128 `A` elements, each of which has a non-zero `id` attribute, a `prior` child with value 1, a `posterior`

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

Figure 4.30: Synthetic queries

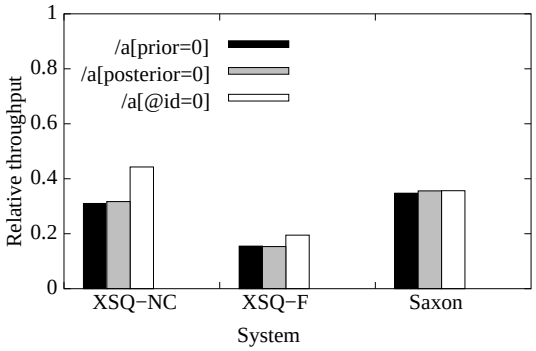


Figure 4.31: Effect of data ordering on throughput

child with value 1, and up to 10,000 `foo` children. There are 700,771 `foo` elements in total. We used the three queries in Figure 4.30. All three produce empty results on the dataset. However, the data items that are used to evaluate the predicates come from different locations in the element.

Figure 4.31 summarizes the results of running XSQ-NC, XSQ-F, and Saxon on these queries. Saxon's throughput is essentially the same for all three queries since it always builds the whole DOM tree before the evaluation. When it traverses the DOM tree to evaluate the query, the document order of the elements is not important. However, the throughput of XSQ-NC is about 30% higher for Q3 than for Q1 and Q2. For Q3, XSQ-NC can determine at the beginning of an `A` element that all the contents in it should be ignored. For Q1 and Q2, on the other hand, the content of every `A` element must be buffered because the `prior` and `posterior` child elements may occur anywhere before the `</A>` tag. We also observe that XSQ-F is not as sensitive as XSQ-NC to the element order. Even if XSQ-F determines that an incoming item is in the result set, XSQ-F cannot output it right away since there may exist undecided queue items. Thus, XSQ-F must first mark the item as "output" and then check the queue, which reduces its sensitivity to the order of the

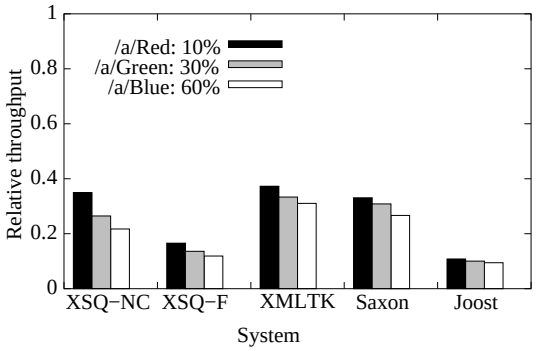


Figure 4.32: Effect of the result size on throughput

elements.

We also studied the sensitivity of throughput to the result size, which varies across the systems. For example, XQEngine is slower than the other systems in Figure 4.22 where the query returns a large portion of the dataset. However, if a node test in the query is not in the data, XQEngine returns the empty result set very quickly because it builds an inverted-file index on all the strings in the data. The other systems, lacking such an index, spend similar amount of time on the query irrespective of whether the node tests in the query appear in the data.

We used Toxgene to generate a 10 MB dataset consisting of a mix of three types of elements (besides a few top level elements): 10% of the elements have name `red`, 30% `green`, and 60% `blue`. The content of each such element is a single character. We used this dataset with three queries: `/a/red`, `/a/green`, and `/a/blue`, generating query results that are roughly 1 MB, 3 MB, and 6 MB in size, respectively. Figure 4.32 illustrates the relative throughputs of the systems on these queries. (XQEngine is not tested for the same reason as described in the previous experiment.)

We observe that XSQ-NC is quite sensitive to the result size. The different

performance is due to the different handling of data items based on whether they are in the result. Items that are not in the result can be ignored by XSQ-NC. If there are more items in the result set, XSQ-NC performs more matching record activations and output operations, which constitute a large portion of the running time of XSQ-NC. We also note that XSQ-F is not as sensitive as XSQ-NC. XSQ-F always keeps the item first since there may be multiple transitions that process the item. Even if the item is not in the result, only when all the transitions finish can we throw it away. The difference between the treatment of elements in and not in the result is therefore not as large as the difference in XSQ-NC. Saxon's throughput is not very sensitive to the result size because, after it loads all data into main memory, all query evaluation is performed in main memory except for the output process, which constitutes only a small amount of the total execution time. Similarly, the low sensitivity of XMLTK's throughput to the result size is because the difference is only in the time required to output the result. However, it is not clear why Joost's throughput is not sensitive to the result size.

4.4.6 Characterizing XSQ-F

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

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

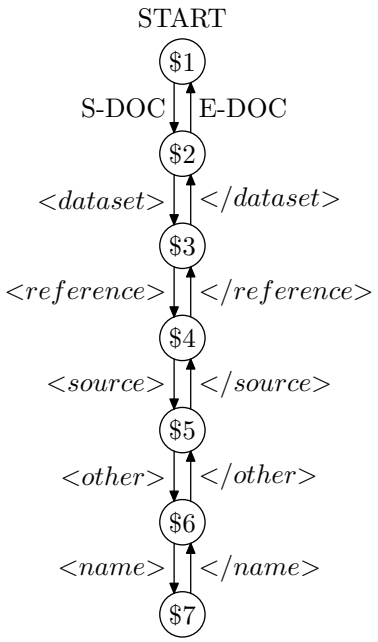
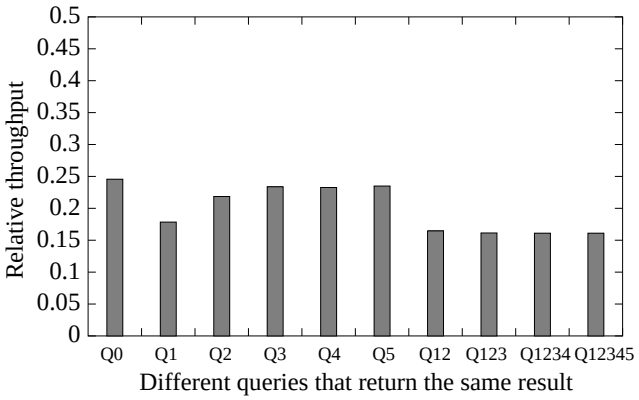


Figure 4.33: HPDT generated for query `/dataset/reference/source/other/name`



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

Figure 4.34: Effect of closure axes in the queries on NASA dataset

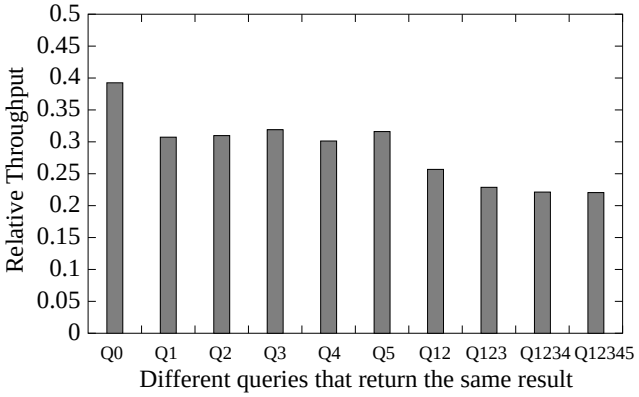


Figure 4.35: Experiment of Figure 4.34 using a modified NASA dataset

result set but have different number of closure axes. In Figure 4.34, Q_S, where $S \subseteq \{1, 2, 3, 4, 5\}$, is the query in which the ith location step has a closure axis for all $i \in S$. For example, the query Q_{123} has closure axes in the 1st, 2nd, and 3rd location steps. (The remaining location steps have the child axis.) The memory usage of XSQ-F when evaluating these queries is summarized in Figure 4.36. The HPDT generated for the query `/dataset/reference/source/other/name` is depicted in Figure 4.33. The HPDTs for other queries have a similar structure, with self-closure transitions and closure transitions in the appropriate places, following the scheme of Section 4.2.1.

Figure 4.36 indicates that the memory used for the different queries does not vary much. This insensitivity is due to the fact that the memory used for storing the HPDT and matching records is only a very small amount in the total memory used by the system. The buffers are responsible for most of the memory usage. Therefore, although different number and position of closure axes lead to different number of matching records at runtime, the difference in overall memory usage is very small.

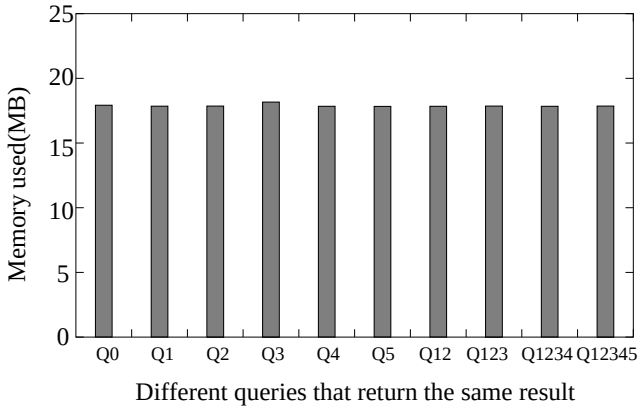
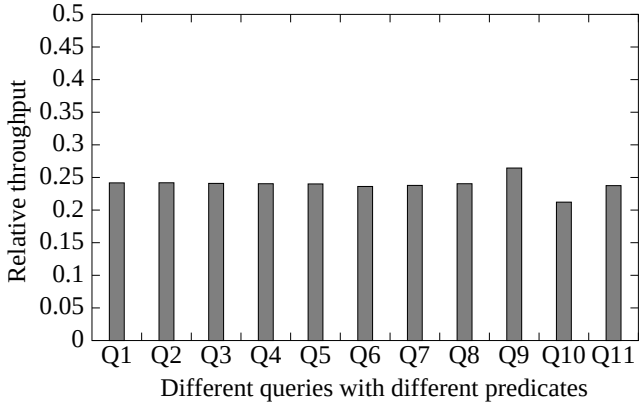


Figure 4.36: Memory usage of queries with closure axes on NASA dataset

Figure 4.34 summarizes the throughput on the above queries. We observe that the throughput is lower for queries with a starting closure axis than for queries with a starting child axis. The DTD of the dataset [11] suggests that all the top level element are `dataset` elements. (The `datasets` element in the DTD is treated as the document root.) qIf the first location step has a closure axis, after the runtime engine (Figure 4.33) makes the transition from state \$1 (with depth stacks omitted here) to \$2, \$1 keeps active. Then, the engine needs to check for every incoming element whether it is a `dataset` element, which involves string comparisons. In contrast, if the first location step uses a child axis, \$1 does not remain active after the transition. Therefore, only for all the child elements of the `dataset` elements does the engine check their names. Any element that is not a descendant of both `dataset` and `reference` is ignored after the engine checks its depth, which is much faster than string comparison.

It is not the position of the closure axes in the query alone that determines the throughput. On examining the dataset closely, we note that the evaluation time is significantly affected by the *selectivities* of each location step. Let S be the set of



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

Figure 4.37: Effect of predicates in the queries on NASA dataset

elements that match the $(i-1)$th location step and S' the set of children of nodes in S. We define the selectivity of the ith location step (for a given dataset) to be the fraction of the nodes in S' that match the ith location step. If the ith location step uses the closure axis, we use descendants instead of children in identifying S' . For the query and dataset of this experiment, each `dataset` element contains one `reference` child, which corresponds to 10%–20% of the total number of events for one `dataset` element.

We also ran these queries on a dataset obtained by removing all child elements of `dataset` elements other than `reference` (which means the selectivity of the second location step changed from around 20% to 100%). The result is summarized in Figure 4.35. We observe that the closure axis in the first location step no longer

has a significant impact on the throughput. (The throughput of query Q1 is not significantly smaller than throughputs of queries Q2, Q2, Q3, and Q5, all of which contain one closure axes but in different location steps.) The reason is that the extra work done for Q1 (checking descendants of child elements other than `reference`) on the original dataset no longer exists since the `dataset` elements in the new dataset have only `reference` child elements. In general, when the selectivity of a location step is small, closure axes preceding this step result in a performance penalty because the descendants that are not in the result set cannot be eliminated by depth comparisons and incur the cost of more expensive string comparisons.

In the previous experiment, we used queries with only closure axes but without predicates. In the next experiment, we used queries on the NASA dataset with predicates of different types and in different positions in the query. The results are summarized in Figure 4.37. We abbreviate the node test `dataset` as `d` in the queries. Similarly, we abbreviate other node tests by their first letter. The first eight queries have the same result although they have different types and numbers of predicates. The last three queries have empty results. We note that the throughputs for the first eight queries are similar because the number of comparisons needed to determine the results of their predicates does not vary much across these queries. For example, although the `dataset` elements typically have several `altname` child elements, the first `altname` child element usually has the attribute `type` that has value *ADC*. Therefore, the queries Q3 and Q4 both check the first `altname` child element and ignore the remaining `altname` elements. However, for query Q10, although the result set is empty, resulting in less time spent on output operations, all the `altname` child

elements of `dataset` elements must be checked. Therefore, its throughput is lower than those of queries Q3 and Q4. We also observe that the query Q9 has the largest throughput among all the queries used in the experiment. The reason is that the predicate in this query `[@subject=test]` can be evaluated to false at the beginning of the `dataset` elements. Thus, all the descendants of the `dataset` elements can be ignored. This experiment demonstrates that XSQ is able to save on comparisons for predicates that have already been evaluated.

Chapter 5

Segment-based Streaming XPath Evaluation

We introduce in Chapter 4 how we evaluate XPath queries with closures, multiple locations, and aggregations. In this chapter, we introduce our general methodology for streaming XPath evaluation, the segment-based evaluation. Its applications are illustrated in Chapter 6, Chapter 7, and Chapter 8.

As we describe in Section 3.3, in a streaming environment where no DOM tree [39] is built, we cannot use the traditional step-by-step evaluation method. In this chapter, we describe an equivalent interpretation of the XPath queries based on the SAX model [53]. First we describe a new concept of *segments* in Section 5.1. The model of the XPath queries we use is described in Section 5.2. In Section 5.3, we define how an element matches a query tree node in a streaming environment. All the example queries in this chapter are evaluated on the sample input stream listed in Figure 5.1. At the right-most end of each line, we number the elements according to the order in which they are encountered in the stream.

5.1 Segments

We use the term segment to refer to a pair of consecutive node tests along with the connecting axis. In the following query:

```
//store[name='BN']//book[//price=10]//title
```

1.	`<store>`	e_1
2.	`<location>NY</location>`	e_2
3.	`<book>`	e_3
4.	`<title>XML</title>`	e_4
5.	`<price>10</price>`	e_5
6.	`<author>Mike</author>`	e_6
7.	`</book>`	e_3
8.	`<name>BN</name>`	e_7
9.	`<book>`	e_8
10.	`<title>Java</title>`	e_9
11.	`<price>15</price>`	e_{10}
12.	`<author>John</author>`	e_{11}
13.	`</book>`	e_8
14.	`</store>`	e_1

Figure 5.1: Sample XML stream

the `store` node test participates in the following two segments: `store/name` and `store//book`. It also participates in a hidden segment `ROOT//store`, where `ROOT` is the hidden zeroth step that alway matches the document root (please see page 42). In general, a segment has the form `M/a::N`, where `M` is the *parent* node test, `a` is the axis, and `N` is the *child* node test.

Recall from Section 3.3 that we evaluate a location step in a *context*, which is a set of nodes in the DOM tree selected by its previous location step. For example,

for query `/store/book/name`, we first select all `store` children of the document root; for every `store` selected, we select all its `book` children; then for every `book` selected, we select all its `name` children, which consist the result set of the query.

We can also evaluate an XPath queries by evaluating its segments. For example, for the above query, we can first evaluate segment `store/book`, whose result consists of the set of `book` elements with a `store` parent, and segment `book/name`, whose result consists of the set of `name` elements with a `book` parent. The two result sets are combined (like a join operation) to get the final result set: the `book` parent of the `name` elements must appear in the result set of the first segment.

A segment is in general easier to evaluate than a location step, especially in streaming environment, since it has no recursive structure such as nested subqueries. Moreover, even if the axis in the segment is a reverse axes, the segment is still easy to evaluate: since we are essentially only finding pairs of elements *matching* certain relations (parent/child or ancestor/descendant), a stack is enough to hold the information we need. In the contrary, the evaluation of a location step may be as complex as that of a full XPath query.

To evaluate XPath queries using segments, the important tasks are to organize results of the segment evaluation and to compute the final result using those intermediate results. The following Example 6 illustrates the basic idea of the segment-based evaluation.

Example 6 *In Figure 5.1, element e_4 on line 4 has two ancestors besides the document root: element e_1 on line 1 and element e_3 on line 3. The ancestor-descendant*

pair (e_1, e_3) matches segment `store//book`*, (e_3,e_4) matches* `book//title`*, and (e_1,e_4) matches* `store//title`*. Therefore, we can conclude from the above facts that e_3 is the result of query* `store//book[//title]` *and also the result of query* `store[//title]//book`*. Meanwhile, e_4 is the result of* `store//book//title` *and of* `store[//book]//title`*. Note that an ancestor-descendant pair may match more than one segments: (e_1, e_3) also matches* `store/book`*.*

We call such an evaluation scheme a **segment-based evaluation**. In Chapter 6 and 7, we illustrate how the segment-based evaluation efficiently processes XPath queries with subqueries and reverse axes.

The above example also suggests that evaluation of segments can be shared among queries. In Chapter 8, we use this feature in our XPASS system to improve the evaluation performance when we need to evaluate multiple XPath queries simultaneously over XML streams. We describe the basic idea of segment-based evaluation in this chapter and details in the remaining chapters of this thesis.

5.2 XPath Query Tree

We model an XPath query as an XPath query-tree (XQT). An example XPath query and its XQT are illustrated in Figure 5.2. An XQT is a node- and edge-labeled tree. Simply put, each XQT node corresponds to a node test in the query and the label of an edge is the axis that connects the two node test. A node may also be associated with a **validation expression** that corresponds to the predicate of the node test.

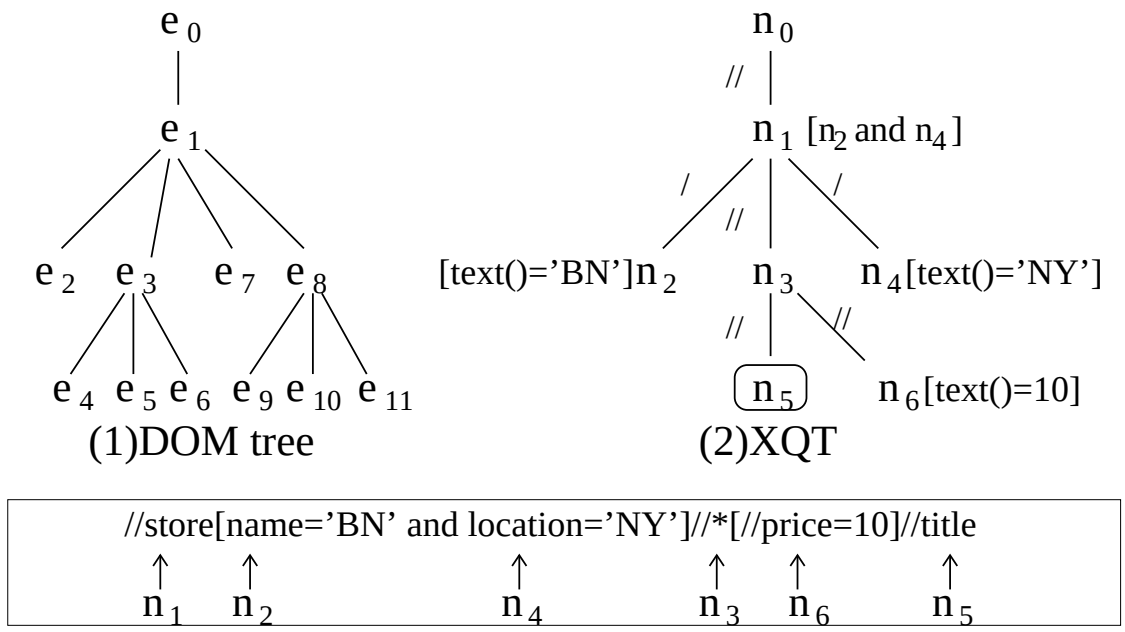


Figure 5.2: DOM-tree and Query-Tree

The XQT is created from the query as follows. For every node test N in the query, we create an XQT node (node for short) with label N. For the first location step $A_1::N_1[P_1]$, the node created for N_1 is connected via an edge with label A_1 to a special root node n_0, who always matches the document root e_0. For every segment `M/a::N` in the query, we make the node created for `N` the child of the node created for `M`; the edge between them is labeled `a`. In the query and its XQT depicted in Figure 5.2, we assign each node a unique name and show it under the corresponding node test in the query. In the figures, the output node is identified by a surrounding box.

Validation expression An XQT node may be associated with a validation expression. The validation expression of node n, denoted by $V(n)$, is a boolean expression obtained from n's predicate by retaining only node tests that are directly

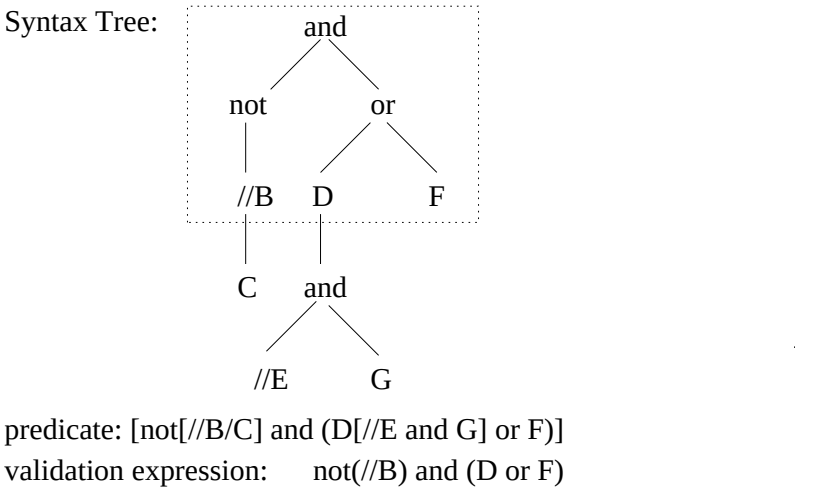


Figure 5.3: Syntax Tree and Validation Expression

related to n. In more detail, we map the syntax tree of n's predicate to the syntax tree of n's validation expression by removing all the nodes that has a non-operator ancestor. For example, the syntax tree of predicate `[not[//B/C] and (D[//E and G] or F)]` is illustrated in Figure 5.3 and the validation expression is `[not(//B) and (D or F)]`.

Trunk and Branch We refer to the path from the XQT root to the output node as the *trunk* of the XQT. Nodes in the trunk are called *trunk nodes*. Paths from trunk nodes to non-trunk leaf nodes, which does not include other trunk nodes, are called *branches*. The *branch root* is the trunk node from which the branch starts. Nodes in a branch, except the branch root, are called *branch nodes*. In Figure 5.2(2), nodes n_0, n_1, n_3, and n_5 are trunk nodes and the rest nodes are branch nodes.

We distinguish the trunk nodes from branch nodes since subqueries are evaluated differently then the original query. For a trunk node, we may need find all the elements that match it. Meanwhile, for a branch node, we may only need find one element that matches it to conclude that it satisfies a predicate for its ancestors.

5.3 Matchings

We evaluate an XPath query by *matching* the elements in the documents with XQT nodes. Intuitively, an element e matches an XQT node n if e satisfies both the pattern specified by n and the predicate of n. We define the concept of matching in more detail in this section.

We define that an element e **matches** an XQT node n iff there exists a sequence of (element, node) pair $((e_0, n_0), \ldots, (e_k, n_k))$, where $e_k = e$ and $n_k = n$ such that the following three conditions hold. In our discussion, e_0 always denotes the document root and always matches n_0, the XQT root.

1. $\forall i \in [1, k]$, the tag of e_i *matches the label* of n_i: The tag and the label are identical strings or the label is $*$;

2. $\forall i \in [1, k]$, e_i *matches the pattern* of n_i: The relation between e_{i-1} and e_i matches the axis labeled on the edge between n_{i-1} and n_i.

3. e *satisfies the predicates* of n: Either n has no validation expression or its validation expression $V(n)$ evaluates to true given the following assignment: the variable x in $V(n)$ is assigned true iff there exists an element e' such that e' matches x and the relation between e and e' matches the axis labeled on the edge between n and x.

Note that the definition is recursive since in the third condition we need to determine whether the element e' matches x. In other words, we need to compute elements that match the leaf XQT nodes first since their validation expressions either

are empty or can be computed instantly (e.g., testing the text value).

A sequence $((e_0, n_0), \ldots, (e_k, n_k))$ is called a **matching** between element $e = e_k$ and node $n = n_k$ if it satisfies the first two conditions. If $\forall i \in [1, k]$, e_i matches n_i, we call it a **full-matching**. Iff there exists a total-matching between e and n do we say that e fully-matches n. The result set of query Q over document D consists of all element e in D that fully-matches the output node of Q.

For example, element e_4 on line 4 in Figure 5.1 is the result of the query in Figure 5.2 because of there exists a full-matching ((e_0,n_0), (e_1,n_1), (e_3,n_3), (e_4,n_5)) between e_4 and the output node n_5: e_0 always fully-matches n_0 by definition; e_1 matches the label of node n_1 and satisfies the predicates `[name='BN']` and `[location='NY']`; e_3 matches the label of n_3 and satisfies the predicate `[//price=10]`; and e_4 matches the label of n_5, which has no predicate.

Equivalence to XPath semantics It is easy to prove that the result set defined by the matching is the same as the result set selected by the traditional step-by-step XPath semantics. First, if there exists a full-matching $((e_0, n_0), \ldots, (e_k, n_k))$ between an element e and the output node O, by mathematical induction we know that every e_i is selected when we evaluate the ith location step. On the other hand, if e is selected after the last location is evaluated, by tracing back the evaluation process we can easily construct a full-matching.

Evaluation by matching has several important features. First, instead of the step-by-step evaluation that requires DOM tree in the main memory, we can evaluate the segments simultaneously when the data streams in as long as we can combine

the results of the segments dynamically and efficiently. Second, the result of a segment can be shared among different matchings, which provides the opportunity to efficiently evaluate groups of queries, as we illustrate in Chapter 8. Moreover, note that the definition of matchings does not exclude any specific axes. Reverse axes, such as "parent" and "ancestor", as well as "following" and "previous", can be used in the definition. Therefore, it is possible to extend the methods to support those axes, as we illustrate in Chapter 7.

Matching in streams In streaming evaluation where only a portion of the data is available, the above definition cannot be used to evaluating the query directly. The existence of a matching can be verified since we maintain a stack of currently open elements. However, it is not straightforward to verify a full-matching since some predicates that are not satisfied by the current data may be satisfied by future data. For example, when we encounter element e_4 on line 4, we only know that there exists a matching $((e_0,n_0), (e_1,n_1), (e_3,n_3), (e_4,n_5))$. However, whether it is a full-matching depends on whether e_1 and e_3 satisfy the corresponding predicates, which may be satisfied by future data in the stream. In this case, we say that these elements are still *pending* on their predicates.

In the following chapters, we are going to illustrate how we organize the partial predicate results and potential result items (*candidates*). In Chapter 4, we use the states in HPDT to encode the predicate results and the hierarchical structured buffers to store the candidates. We extend those ideas and support queries with more complex features such as subqueries and reverse axes.

Chapter 6

XPath Query with Subqueries

In this chapter, we describe a method for streaming evaluation of XPath queries whose predicates may contain subqueries. Our method marks elements in the stream with partial evaluation results. The marking process is guided by the query tree of the XPath query. Meanwhile, the potential result items and their partial predicate results are also efficiently maintained. To the best of our knowledge, this method is the first to support XPath subqueries in a streaming environment. It also provides features such as 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.

6.1 Introduction

In this chapter, we focus on XPath queries with subqueries in their predicates. Since each subquery may itself be a complex XPath expression, stream processing of such XPath queries poses significant additional challenges. First, it becomes more difficult to keep track of the relations between the buffer items and their pending predicates since components of the subqueries may be evaluated upon different por-

```
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 6.1: Example XML Data

tions of the stream. Second, since the boolean operators AND, OR, and NOT are permitted in the subqueries, we not only need to keep track of different portions of the subqueries and the relations between them, but also need to process the new universal semantics of the predicates when the `not()` function is present. Moreover, since the subqueries have different semantics than the original query, we have to use this fact to evaluate the subqueries efficiently. We need also to differently address problems such as multiple matchings in subqueries. We illustrate some of the difficulties in Section 6.2.

Figure 6.1 depicts an example XML file that is used through out this chapter.

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

Figure 6.2: Combination of predicate results

6.2 Motivating Examples

The following examples highlight some of the difficulties of supporting subqueries in streaming evaluation. Example 7 illustrates why multiple matchings are difficult to handle in streaming evaluation.

Example 7 ***[Multiple Matchings]*** *When we encounter the* `title` *element in line 18 it has three matchings with the query, as shown in Figure 6.2. Multiple matchings require no extra handling in the DOM-based evaluation as illustrated in Section 6.3.1, since each step is always evaluated after the previous step is evaluated.*

In the streaming environment, upon encountering this `title` *element, we need to keep all the information in Figure 6.2 in order to determine in the future the membership of the* `title` *element in the result set. Moreover, since subqueries such as* `//quantity=1` *may be replaced by more complex XPath queries, such a table could expand to a fair complex degree. It is obvious that we cannot treat the multiple matchings in this straightforward manner.*

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 is 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 run-time. Following example illustrates some of the difficulties.

Example 8 ***[Scopes of Elements]*** *Consider the query used in Example 7. 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. Essentially, for predicate p of an element e, there may be multiple partial results of p at a certain moment in stream. The following example illustrates the complexities.

Example 9 ***[Complex Subqueries as Predicates]*** *We modify Q_1 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 match 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 used in the evaluation of the subqueries.

6.3 Evaluation Algorithm

We first describe a main-memory evaluation method. The method matches the patterns specified by the query top-down in the query tree while the predicates are evaluated bottom-up. Although the method evaluates the XPath query by traversing the DOM-tree, it does not specify the order of the traversal and the number of times it visits each node. We later adapt the method to limit it to a single preorder traversal of the DOM tree and thus can be applied in the streaming

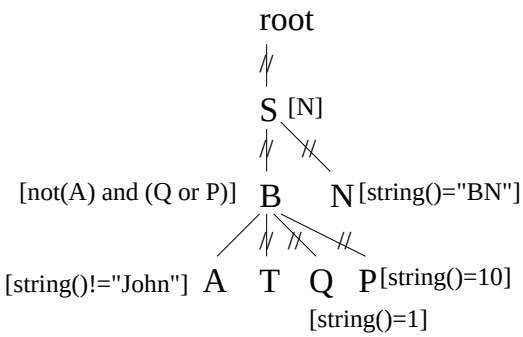


Figure 6.3: The Query Tree of Q_1

environment.

6.3.1 A Main-memory Method

Query Tree To evaluate an XPath query $q = A_1N_1[P_1]\ldots A_kN_k[P_k]O$, we first transform q into a query tree P (please see Section 5.2) that represents all and only the *patterns* in q. We briefly restate the concept of query tree here. For every node test `X` in q, we create a node in P with label `X`. For each segment `X/a::Y` (or `X[a::Y]`) in q, we create an parent-child edge with label "a" from `X` to `Y` in P. The root node of the query tree always fully-matches the document root. It always is labeled "`root`" and connects to node created for N_1 via an edge with label A_1. The leaf node N_k is called the **output node**, denoted by O_P. The query tree of Q_1 is depicted in Figure 6.3. Unlabeled edges in the tree represent the child axis, which is the default. In a query tree, the **trunk nodes** are generated from node tests in the main trunk. The other nodes are called **branch nodes**.

The main-memory XPath evaluation algorithm described here matches the patterns specified in the XPath query with the document tree using a top-down

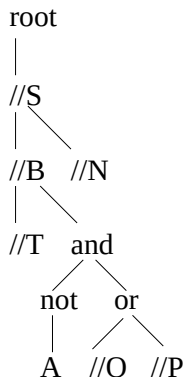


Figure 6.4: The Syntax Tree of Q_1

matching method; for every node matched in the document tree, its predicate is evaluated bottom-up.

Algorithm The main-memory algorithm is described in Listing 6, where $P_D(e)$ denotes the path from the root of the document tree to element e and $P_P(p)$ denotes the path from the root of the query tree to node p.

In this algorithm, for every element e, we first determine whether its name matches a query tree node p using the **LocatePatternNode** function. If there is no such p, we conclude that the element is not used in the evaluation and can be ignored. Otherwise, we use $P_P(p)$ as a regular expression obtained by concatenating node labels along the path and inserting Kleene stars where `//` axes are used. The function **MatchPath(**$P_D(e)$**,** $P_P(p)$**)** returns true iff the string obtained by concatenating element names in $P_D(e)$ matches the regular expression specified by $P_P(p)$. If the string matches the pattern, we mark e with p; otherwise no operation is performed.

After all the elements are marked, we evaluate the predicates for each of them. The function **Evaluate(**e**,** p**)** described in Listing 7 evaluates the validation expres-

Algorithm 6: Main-memory Evaluation

```
Input: an XPath query q, an XML document d
1  Build the query tree P for q;
2  Build the DOM tree D for d;
3  for ∀e ∈ D do
4      p ← LocatePatternNode(e);
5      if p ≠ null then
6          if MatchPath(P_D(e), P_P(p)) then
7              mark e with p;
8  for ∀e ∈ D do
9      if e is marked with p then
10         Evaluate(e, p);
   /*O_P is the output node of P*/
11 for ∀e ∈ D marked with O_P+ do
12     if ExistTruePath(e) = false then
13         unmark e;
14 return all nodes marked as O_P+;
```

sion $E(p)$ of pattern node p for element e. If $E(p)$ is empty, the function always returns true. For non-empty $E(p)$, each variable in the expression is assigned values as follows: A variable x is set to true if there is a descendant (child if x needs to be a child of e) of e that has mark $x+$; otherwise x is set to false. The boolean expression $E(p)$ is then evaluated. If the result is true, e is marked with $p+$ to denote that the predicate of p evaluates to true if we use a e as the context node. If the result is false, we **unmark** e since it has failed the predicate.

The `Evaluate(`e`,`p`)` function is recursive since it calls itself for e's descendants to determine the assignment of the variables in p. The recursive call terminates when the descendant is a leaf node in the query tree, which has no predicate and therefore has an empty validation expression. Essentially, the evaluation is performed from

Algorithm 7: Function Evaluate(e, p)

```
   parameter: an element e, its mark p
 1 if E(p) = null then
 2 |   mark e with p+;
 3 |   return;
 4 for ∀x used in E(p) do
 5 |   for ∀e' that is e's descendant and marked with x do
 6 |   |   Evaluate(e', x);
 7 |   if ∃e' that is e's descendant and marked with x+ then
 8 |   |   x ← true;
   |   else
 9 |   |   x ← false;
10 evaluate E(p) using the assignment;
11 if result is true then
12 |   mark e with p+;
   else
13 |   unmark e;
```

bottom-up: the predicates for the leaf nodes are evaluated first and the results are then propagated to their ancestors.

After the predicates for all elements are evaluated, we locate the elements that are marked with O_P+, i.e., the target result elements. For each element e located, the **ExistTruePath** function tests whether there exists a matching between $P_D(e)$ and $P_P(O_P)$ such that every element in the matching is marked with "+". (Recall from Section 3.3 that a matching between a sequence of elements and a sequence of node tests is a subsequence of elements that matches the node tests in order.) When there are multiple such matchings, e is included in the result only once.

Example 10 ***[Evaluation by marking]*** *The process of applying the algorithm to evaluate Q_1 over the stream in Figure 6.1 is illustrated in Figure 6.5. Matching*

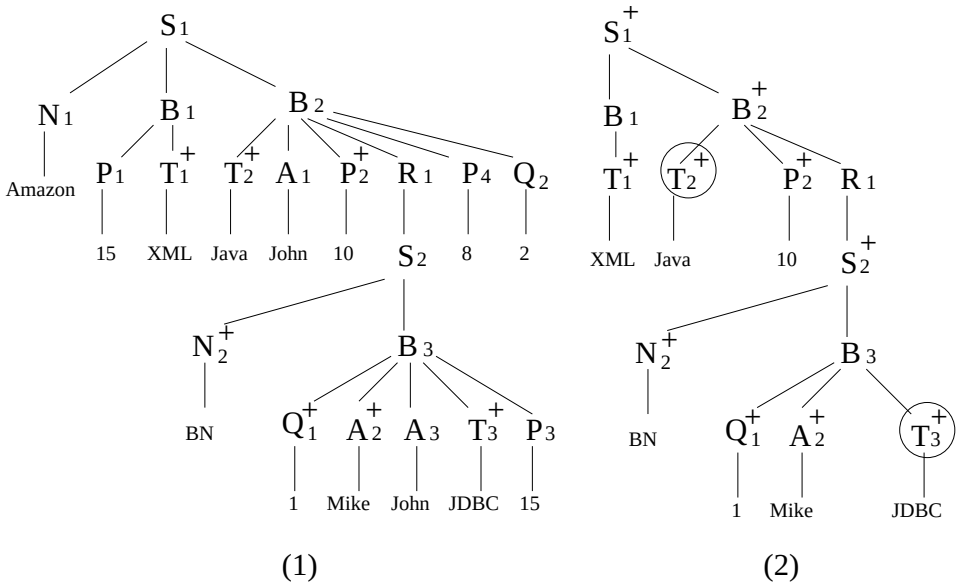


Figure 6.5: Main Memory Evaluation of Q_1

the query patterns with the paths in the document tree is straightforward and the marks are omitted in the figures. In Figure (1), every element marked with a leaf node in the query tree has its predicate evaluated. If the result is true, the element is marked with a + sign. In Figure (2), the validation expression of each element marked with a non-leaf query tree node is evaluated. For example, for B_2, `not(A) and (P or Q)` *evaluates to true since it has no child* `A` *with +, a child* `P` *with +, and a descendant* `Q` *with +. We do not consider the* A_2 *descendant of* B_2 *since the query tree specifies that A must be a child of B. After all the elements are marked, we can see that* `ExistTruePath` *returns true for both* T_2 *and* T_3 *and therefore they are returned as the results.*

Algorithm 8: Begin Event Handler

```
   Parameter: the begin event of element e
1  Push(e,S);
2  p ← LocatePatternNode(e);
3  if p = null then return;
4  p' ← p's parent node;
5  if ∃e' ∈ S → (e' is marked with p') ∧IsNecessary(e', p) then
6  |  Evaluate(e, p);
7  |  if e is marked with p+ then
8  |  |  ProcessTrueNode(e, p);
   |  else
9  |  |  if p = O_P then Enqueue(e);
```

Algorithm 9: End Event Handler

```
   Parameter: the end event of element e
1  Pop(S);
2  p ← e's current mark;
3  if p ≠ null then
4  |  Evaluate(e, p);
5  |  if e is marked with p+ then
6  |  |  ProcessTrueNode(e, p);
   |  else
7  |  |  unmark e;
8  |  |  if p is a trunk node then Clear(e);
```

6.3.2 Streaming Evaluation Algorithm

We now modify the main memory algorithm into a streaming algorithm that limits to one preorder traversal of the tree.

Data Storage In streaming evaluation, we use only a *stack* and a *queue* as the data storage. The stack, denoted as S in the pseudocodes, stores the currently open elements (whose start tags, but not end tags, have been encountered) by pushing and popping elements in response to their start and end tags, respectively. The queue stores the potential result items for which some predicates are still pending,

Algorithm 10: ProcessTrueNode(e,p)

```
   Parameter: an element e marked with p+
1  if p is a trunk node then
2      if ExistTruePath(e) then
3          Output(e);
       else
4          Upload(e);
   else
5      CheckAncestor(e);
```

i.e., all the *candidates*. When we encounter an element that has been justified to be in the result, if the queue is not empty, we enqueue the element in order to maintain the document order in the output. A one-bit label is attached to every queue item to distinguish the justified result items and the candidates.

If stack item e matches a trunk node p and has its predicate pending, we attach a **buffer** with it. The buffer of e is the set of pointers to the candidates in the queue that are e's descendants. Since e's predicate is currently pending, these candidates should be notified when e's predicate is determined. Although each candidate may have several predicates pending, as we describe later, its pointer is always stored with the nearest ancestor whose predicate is pending. We may think of the buffers as a method to group candidates based on the predicates they are pending on.

Partial predicate result In streaming evaluation, not all the predicate results are available at the same time. Therefore, to adapt the previous main-memory method into a streaming environment, we need to main the partial predicate result for every element.

First, due to the existential semantics of the XPath predicates, we do not need

to locate all elements that justify the same subquery for the same element. For an element e, when we see a descendant that satisfies a subquery in its predicate, we can simply associate a flag with e to denote that this subquery has been satisfied.

For an element e in the stack that matches pattern node p, we mark e with p and associate a k-bit flag with e in which the ith bit denotes the *current* value of the kth variables in p's validation expression $E(p)$. If $E(p)$ is empty, i.e., e has no predicate, a 1-bit true flag is associated with e to denote that e satisfies its (null) predicate. If $E(p)$ is not empty, when we see a descendant e' of e in the stream that matches a child node p' of p and is marked with $p'+$, we set the corresponding bit, also referred to as **bit** p' in the flag.

Predicate evaluation Based on the flag associated with e, the function **Evaluate(e,p)** evaluates $E(p)$ in two manners. The first is called *tvl-evaluation* (three-valued-logic evaluation), which is used whenever a subquery is satisfied. Variable x in $E(p)$ is assigned to TRUE if bit x is set and NA otherwise. $E(p)$ is then evaluated using the usual three-valued logic. If the result is TRUE, mark e with $p+$; if the result is FALSE, unmark e; otherwise e is still marked with p.

We use *boolean-evaluation* upon encountering the end tag of the element. Variable x is assigned to TRUE if bit x is set and FALSE otherwise. Since we allow only descendant be used in the predicates, pending subqueries can not be satisfied after the end tag of the element is encountered. If the evaluation result is TRUE, mark e with $p+$, otherwise unmark e.

Event Handlers The algorithm is presented in the form of event handlers in

Listing 8 and Listing 10. It depicts how each incoming event is processed.

Upon the begin event of an element e, the `LocatepatternNode` function first checks whether its name matches any pattern node. If it matches a pattern node p, we then check whether the path $P_D(e)$ matches $P_P(p)$. Since all the elements in $P_D(e)$ is in the stack, the matching can be performed using only the stack. If there exists such a matching, the validation expression $E(p)$ for e is then evaluated. However, unless $E(p)$ is empty or tests e's attributes, $E(p)$ usually is pending at e's begin event and e is marked with p. In the case where $E(p)$ evaluates to true, the `ProcessTrueNode` procedure is executed, which is described later. If p is the output node O_P, the contents of e specified by the output function is enqueued in the queue and the pointers to them are put into the buffer associated with e.

At the end of element e with mark p, function `Evaluate(e,p)` returns either TRUE or FALSE since every variable in $E(p)$ can be now assigned to either true or false. If `Evaluate(e,p)` returns false, we can unmark e to denote that it fails the predicate. Otherwise, e is marked with $p+$, and the function `ProcessTrueNode` is called.

ProcessTrueNode If p is a branch node, e must be used by at least one of its ancestors for predicate evaluation. The **CheckAncestor** function (described in Listing 11) applies this result to ancestors that are marked with the parent node of p, i.e., who use e in their predicate evaluation. In the `CheckAncestor` function, we set bit p in the flag of the ancestor element e' and then evaluate the predicate of e'. If the result is true, e' is marked with "+" and the `CheckAncestor` function

is called recursively until a predicate evaluates to NA or no ancestor is available to mark. If the result is false, we unmark e', but no further operation is needed. If the result is NA, no operation is performed.

If the element e is marked with O_P+ (recall that O_P is the output node of the query tree), the `ExistTruePath` function is called to check whether e has been justified to be in the result. If `ExistTruePath` returns true, we output e's contents right away (either emit e if it is at the head of the queue or label it as output otherwise). If `ExistTruePath` returns false, for every matching between e and O_P+, there exists some ancestors in the matching that have their predicates pending. In this case, for every matching m, `Upload` function puts e's contents (whose pointers are stored in e's buffer) to the nearest ancestor of e in m that is not marked with $+$. By uploading we move the pointers to the queue items from the current buffer to the buffer of the ancestor.

Note that, to solve the problem of multiple matchings, of the pointer to a queue item may be duplicated during uploading. A reference count is created for the queue item to keep track of the number of copies created. Essentially every copy of the pointer corresponds to a matching between the candidate and the output node. As soon as the queue item is labeled as output, no further operation (following other copies of the pointer) can alter the fact. If a queue item is not marked as output, it is removed from the queue when the reference count is decreased to zero. Therefore, a candidate is determined to be in the result if one matching satisfies all the predicates, i.e., a full-matching exists; it is removed from the queue if and only if all the matchings have failed their predicates, which implies that we only delete a

queue item when the reference is decreased to zero.

If element e is marked with $p+$, where p is a trunk node other than the output node, the operation for the end event of e is similar to the operation described for the case where p is O_P. The only difference is that we do not enqueue anything in this case. In both cases, we clear the buffer if e is unmarked.

One feature of the method is that we only evaluate subqueries that is currently *necessary*. A necessary subquery is a pending subquery whose result may change the result of the predicate. For example, in predicate `[A or B]`, if subquery `A` has evaluated to true, `B` is not necessary even it is pending. Since subqueries are denoted as subtrees in the query tree, we discuss the necessities of subqueries in terms of necessities of variables in the validation expressions.

If element e is current marked with pattern node p, the function **IsNecessary(**e**,** X**)**, where pattern node X is a child of p, returns true if and only if X is a trunk node or variable X is necessary to evaluate $E(p)$ for e.

If X is a trunk node, p must be a trunk node as well. In this case, since e is still pending, any node with mark X may contain potential results and thus needs to be processed.

If X is a branch node and a variable in $E(p)$, only if bit X is current pending in e's flag and a value change of X may change the result of $E(p)$ for e, X is necessary. We use an `IsNecessary` function to determine the necessity and describe it in Section .

We now describe two mechanisms used in the XSQ system that supports the three features very efficiently. First, we make the stack linked by threads to facilitate

Algorithm 11: Function CheckAncestor

```
   Parameter: element e marked with "p+"
1  p' ← p's parent node;
2  for ∀e' ∈ S marked with p' do
3      Set bit p in the flag of e';
4      Evaluate(e', p');
5      if e' is marked with p+ then
6          ProcessTrueNode( e', p' );
       else
7          if e' is unmarked then
8              if p' is a trunk node then
9                  Clear(e);
```

the pattern matching task and the `Upload`, `ExistTruePath` and `CheckAncestor` functions. Second, instead of using one flag and evaluate the expression each time a bit is set, we introduce a two-flag marking scheme such that the current predicate result and the result of `IsNecessary` function can be obtained directly from the flags.

6.3.3 A Threaded Stack

An element e in the stack with mark p (or $p+$) is threaded (linked) to its ancestor e' that is marked as the parent of p, p'. Since there may be multiple ancestors that match p', e may have multiple outgoing threads.

We keep a set of **active elements** in the stack. An element e is active iff it is marked with a pattern node p (or $p+$) and either e is the top element in the stack or pattern node p has a outgoing edge with label "//". Intuitively, the active elements serve similar purposes as the active states in an automaton.

For every incoming element e matching the name of a pattern node p, we first

check for every active element e' (with mark p' or $p'+$) whether p' is the parent of p and p' is necessary for e'. If yes, e is marked with p and set as active. A thread is created from e to e'. Note that e' becomes inactive if e' is previously the top element and all outgoing edges from p' are labeled with child axes. It is easy to prove by induction that an element e is marked with p iff the path $P_D(e)$ matches the pattern specified by $P_P(p)$.

CheckAncestor Consider an element e that matches a non-trunk node p. When the predicate of e evaluates to true, we follow the outgoing threads from e to check ancestors that is marked with p's parent p' . Therefore, in the pseudocode of the `CheckAncestor` function, instead of recursively scanning the stack for ancestors marked with p', we can now only check the actual ancestors through the links.

It is true that using the threaded stack does not reduce the asymptotic time complexity of the `CheckAncestor` function, which is $O(ld)$ where l is the depth of the stack and d is depth of the pattern node p. However, the amortized time complexity is reduced to $O(l)$ since every node in the stack can be updated at most once.

ExistTruePath Another benefit of the threaded stack is that, each matching between a candidate and the output node is represented by a threaded path from the document root to the candidate (we can get every such path by following the threads going out from the candidate). Although it seems that the `ExistTruePath` function may need to check the whole stack following the threads whether such a path exists, it can be implemented as a single-step operation.

We associate a true-path flag to every element that is marked with a trunk node. The flag denotes the current result of the `ExistTruePath` function. When the `ExistTruePath` function is called for element e, we just check all ancestors that are directly linked by e. If any of them has the true-path flag with value true, the function returns true and the true-path flag of e is set to true.

Upload Recall that, for every element e in the stack with mark p that is a trunk node, we associate with e a buffer that contains pointers to all the queue items are pending on e. The `Upload` function is used to move the pointers in e's buffer when we encounter e's end tag and e's predicate evaluates to true. For every matching m between $P_D(e)$ and $P_P(p)$, those pointers are moved to the buffer of e's nearest pending ancestor in m. Note that if every ancestor in m satisfies its predicate, every candidate in e's buffer are justified to be in the result, and therefore should be processed by `Output` function instead of `upload`. Given the threaded stack, the `Upload` function just follows the threaded paths and move the pointers into the buffer of the first element in every path that is not marked with +.

6.3.4 A Two-Flag marking Scheme

Keeping track of partial predicate result for every element is another important operation in the evaluation. As described in Section 6.3.2, we use a bitmap flag that records the current variable values in the validation expression.

Such a single-bit marking scheme, however, has two problems. First, since we allow *not()* function in the subquery, a variable in the validation expression

may be assigned to one of these three values: TRUE, FALSE, and NA. When `not()` function is not allowed, a subquery can be falsified only at the end of an element. Therefore, a single-flag scheme will do: a set bit stands for TRUE; an unset bit stands for NA before and FALSE at the end event of the element. Second, to support optimal predicate evaluation, very often we need to determine whether a variable in a validation expression is *necessary* for a given element. If the validation expression has a complex structure, we have to use its syntax tree every time to determine whether a variable is necessary.

Example 11 *Why not single flag? Consider a predicate with two simple subqueries in the query* `//X[A and B]`*. We associate a two-bit flag with every* `X` *element and initially set the flag to* `00`*. When the pattern* `//X/A` *(*`//X/B`*) is matched, we set the first (second) bit of the flag of the element* `X` *matched. 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, a single-flag scheme is not enough when there are `not()` *functions in the predicate. Consider the query* `//X[A and not(B)]`*. When the pattern* `//X/B` *is matched in the stream, we should mark the* `X` *element to denote that it is not in the result. Since all possible values of the flag have already been used to denote variable values, e.g.,* `00` *denotes both subqueries are pending, we cannot denote such fact using only one flag.*

We can associate a three-valued bit for each variable. However, it is slower to compute than a boolean bit flag. More importantly, it also has the same drawback

as a boolean bit flag, i.e., the relations among the subqueries are lost. For example, for query `[((A and B) or C) and ( D or E )]`*, we cannot determine whether the subqueries are necessary based solely on the flag, and thus a syntax tree of the predicate has to be maintained to determine the necessity.*

Therefore, we introduce a **two-flag marking scheme** in our method. For every stack item, we associate it with two flags (as bitmaps): a TRUE-flag that records which clauses in the disjunctive normal form (DNF) of its validation expression have been evaluated to true, and a FALSE-flag that records which clauses in the conjunctive normal form (CNF) of the validation expression have been evaluated false.

To operate these two flags, every branch node p in the query tree is associated with a TRUE-map that indicates which clauses it appears in the DNF of its parent's validation expression and a FALSE-map that indicates which clauses it appears in the CNF. Example 12 illustrates the basic ideas.

Example 12 *Two-flag Marking Scheme 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` *for an element e, we first check the current flags of e. If the* TRUE*-flag's first two bits are 1's, there are only two possible cases: (1) both* `A` *and* `B` *have been satisfied; (2)* `C` *has been satisfied. In neither case is* `C` *necessary. If the* FALSE*-flag's last two bits are 1's, there are also only two possible cases: (1)* `C` *has been failed; (2) both* `D` *and* `E` *have been failed. Again, in neither case is* `C` *necessary.*

We now describe the marking process in more detail. When an element e is newly marked with $p+$ where p is a branch node, following the thread going out of e, we locate all the stack items that is marked with p', where p' is p's parent. For the TRUE-flag of every located element e', we perform a boolean "or" operation with the TRUE-map of p. The result TRUE-flag of e' denotes that every clauses in the DNF of $E(p')$ that p appears in have been satisfied. If there is a `NOT` operator before p in the normal forms, we perform a boolean "or" operation on p's FALSE-map and e''s FALSE-flag in order to denote that those clauses in the CNF of $E(p')$ have been failed.

At the end event of an element e with mark p, if variable x in $E(p)$ is still NA, x is assigned FALSE. Accordingly, we set all the set bits in x's FALSE-map in e's FALSE-flag. For a stack item without a predicate, its TRUE-flag is always all set and the FALSE-flag is always all zeros.

Given the above method of marking the two flags, the predicate of element e with mark p evaluates to TRUE iff all the bits in e's TRUE-flag are set and FALSE iff all the bits in e's FALSE-flag are set.

In the `IsNecessary(`e`,` x`)` function, we can determine whether variable x is necessary for e by checking the two flags: for e's TRUE-flag, we check whether all bits in x's TRUE-map are set; for e's FALSE-flag, we check whether all bits in x's FALSE-map are set. The result of checking are obtained by performing "xor" operations on the two bitmaps. If *either* test returns true, the `IsNecessary(`e`,` x`)` returns false, otherwise returns true.

Therefore, given the above marking scheme, both the `IsNecessary` function and `Evaluate` function can be implemented efficiently using bitmap operations.

6.4 A Comprehensive Example

In this section, we apply the technique described in the previous sections to evaluate Q_1 over the XML stream depicted in Figure 6.1. Some highlights of the evaluation process are depicted in Figure 6.6. In each stack item, instead of the name of the element, we display the DOM node numbered as in Figure 6.5, with the TRUE- and FALSE-flags shown to the right. The active elements are enclosed in bold

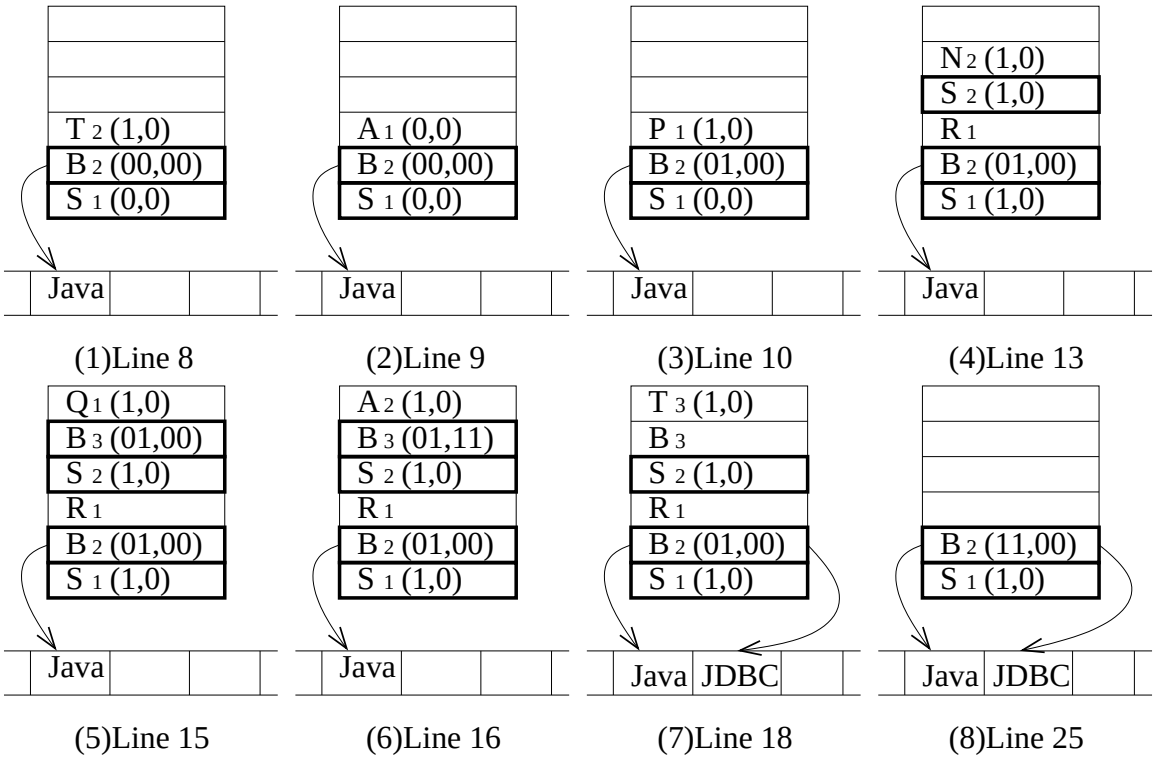


Figure 6.6: Streaming Evaluation of Q_1

boxes. Since the threads between stack items are easy to derive from the figure, we do not show them explicitly. The queue is shown below the stack and the pointers to queue items are depicted using arrows. The start and end tags are not shown in the queue.

In step (1), when we encounter the T_1 element in line 8, the begin event handler is called. Since there is no predicate for the `title` node test, the `ProcessTrueNode` function is called. The `ExistTruePath` returns false and the `Upload` function is called. In this case, the pointer to the begin tag "¡title¿" is put directly to the buffer of the parent node B_2. This pointer is not shown in the figure, but we can

see the following text event will put the pointer to the text direct to the buffer of B_2. (Recall that we may consider the text event as the begin event of a text child.)

In step (2), we encounter the A_1 element in line 9. At the begin event, A_1 is marked with `author` (not `author+`), i.e., the flags are (0,0). At the text event, the predicate `[string()!="John"]` evaluates to false and no operation is performed. (Strictly speaking, the string() function returns the aggregation of all the text contents of the `author` element. This predicate can be evaluated only at the end of the element. We simplify the case here by treating the string() function as the text() function since there is only one text event in the element.)

In step (3), the P_1 element in line 10 evaluates the subquery `[//price=10]` to true. Since the TRUE-map for the `price` in the predicate of the `book` is 01 (i.e., it appears in the second phrase of the CNF of the predicate), we set the second bit TRUE-flag of B_2.

In step (4), the N_2 element in line 13 evaluates the predicate `//name="BN"` to true. Note that at this time, both S_1 and S_2 are active and accept this `name` descendant. Therefore, there are threads from the stack item N_2 to both and S_1 and S_2. When the predicate evaluates to true, the TRUE-flags of both S_1 and S_2 are updated. (They are both marked with `store+` now.)

In step (5), the Q_1 element in line 15 evaluates the subquery `//quantity=1` to true. However, since `IsNecessary` function only returns true for B_3, only B_3 accepts this Q_1 descendant and has its TRUE-flag updated. For B_2, `IsNecessary` function returns false since the TRUE-map of the `quantity` node is 01, which is already set in the TRUE-flag of B_2.

In step (6), the A_2 element in line 16 evaluates the subquery `author!="John"` to true. However, since there is a `not` operator between the `author` node and the `book` node in the syntax tree, we update the FALSE-flag of B_3 element using `author`'s FALSE-map. Since the DNF of the predicate is `[(not(A) and //P) or (not(A) and //Q]`, the FALSE-map is 11. The FALSE-flag of B_3 is updated to 11, which means B_3 fails its predicate and therefore is being unmarked. Note that only B_3 is updated since the `author` element is specified to be the child of the `book` and thus no thread is created between B_2 and A_2.

In step (7), when we encounter the T_3 element in line 18, B_3 is no longer active. However, since the `title` element is specified to the descendant of the `book` element, B_2 accepts T_3 and the content of T_3 is only uploaded to B_2.

In step (8), when we encounter the end event of B_2 in line 25, since the subquery `[author!="John"]` is still NA, it now evaluates to FALSE. Since there is the `not` operator, we update the TRUE-flag of B_2 using `author`'s TRUE-map, which is 10. We can see that the TRUE-flag of B_2 is now 11, which mean B_2 is marked with `book+` now. The `ProcessTrueNode` function is then called. Since the `ExistTruePath` function returns true, the contents in the buffer are sent to output.

In conclusion, we can see from the process that: the method only buffers the potential result items that cannot be determined; only subqueries that may change the result of the predicate are evaluated; the result items are sent to output at the earliest moment that their membership in the result set can be determined.

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 6.7: Systems

6.5 Performance Evaluation

6.5.1 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 6.7. For each system, we used the latest version available at the time of experimentation. (At that time, we were unable to obtain the XAOS and BEA systems, discussed in Section 2.2.) Here, we highlight only the features relevant to streaming XPath

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 6.8: Datasets

processing. **XML toolkit (XMLTK)** [4] 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 [9], 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, we have to manually program the stylesheet to evaluate predicates that require buffering, which we do not use in the experiments. **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 [32]. 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 6.8. We used three real datasets that vary in size and other characteristics: (1) the **ADC** astronomy research dataset from NASA [11], (2) the **DBLP** bibliographic dataset [48], and PIR-International Protein Sequence Database (**PSD**) [67]. We also used three synthetic datasets, generated using the XMark benchmark [63]. 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.

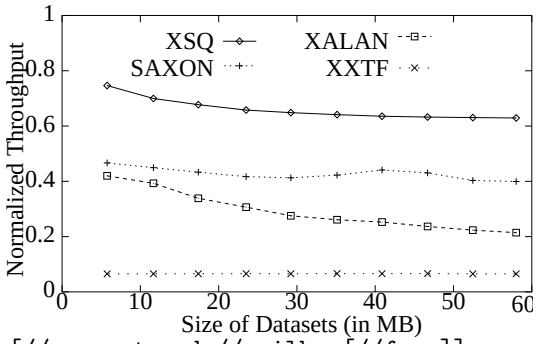
Metrics

We measure the **normalized throughput** of a system as the ratio of its raw throughput (rate of input consumption) to the raw throughput of 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.

We also 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 result. 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.

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



Q1: `//regions/samerica[//payment and //mailbox[//from]]`
`//item [quantity>=2 or shipping]/name`

Figure 6.9: Throughput of Query on XMark Datasets

them, the same system may report substantially different performance on the same dataset on different queries (as illustrated by our results described later). Therefore, instead of using a random set of queries in the experiments and reporting the average performance, we varied the features of the tested queries for every system on every dataset and analyzed the results case by case.

Two important features we studied are the query's element-selectivity and event-selectivity. The **element-selectivity** stands for the number of elements that are used in the evaluation of a 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, in which 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.

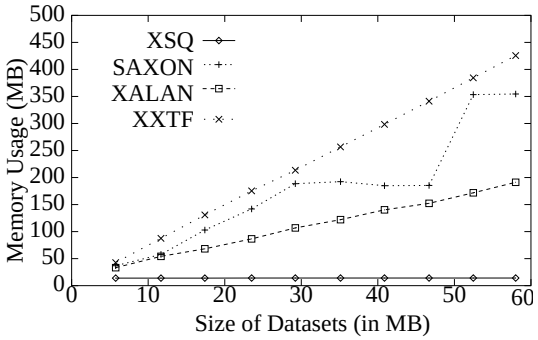
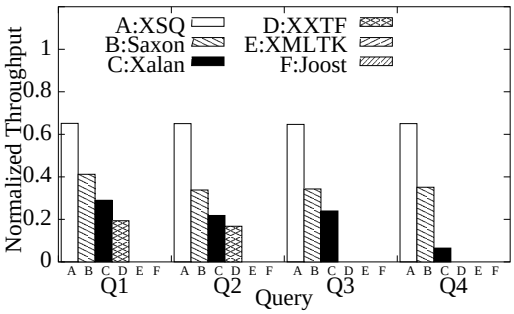
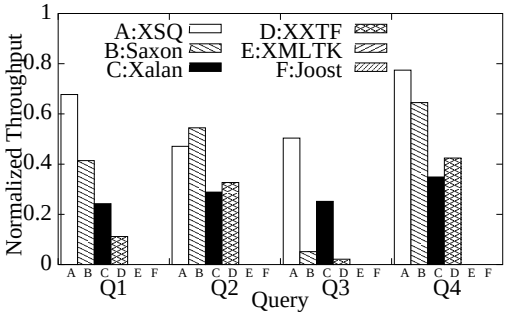


Figure 6.10: Memory Usage of Experiments in Figure 6.9




```
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 6.11: Complex Queries on XMark-s Dataset




```
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
```

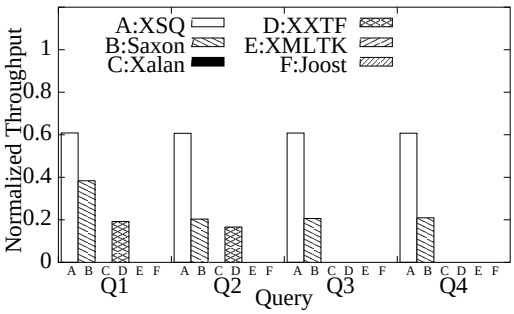
Figure 6.12: Complex Queries on NASA Dataset

6.5.2 Evaluating Subqueries

We first compare the performance of the systems when they evaluate *XPath queries with subqueries*. We do not include XMLTK and Joost in this set of experiments since the former does not support subqueries and the latter requires programming to support subqueries.

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). **Figure 6.9** depicts the normalized throughput of the systems when querying the datasets, and **Figure 6.10** depicts the maximum memory usage during the evaluation. The normalized throughputs of Saxon, XSQ, and XXTF are almost constants, while Xalan's performance degrades when the size of the dataset increases. 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-S does not need store all the data in main memory, its memory usage is almost constant.

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



Note: Queries are the same as in Figure 6.11.

Figure 6.13: Complex Queries on XMark-m Dataset

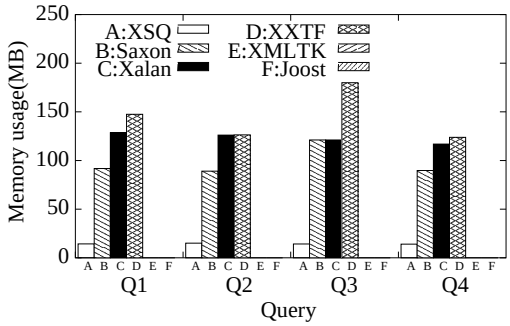


Figure 6.14: Memory Usage for Experiments in Figure 6.12

not clear, but should not be insufficient memory since the memory usage was stable at around 360MB during the three hours' running.

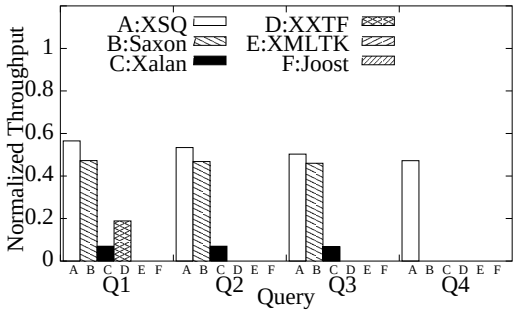
We also tested other complex queries on the XMark datasets. For the same set of queries, **Figure 6.11** and **Figure 6.13** depict the throughputs of the systems on XMark-s(29MB) and XMark-m(117MB) datasets. We chose the queries so that `Q1` contains only child axes in the main trunk while `Q2` replaces 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 moving 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 by the value comparisons required for predicate evaluations as much as the join-based systems. Since navigation-based systems compare the values during the traversal of the data (either in main memory or streaming form), usually the intermediate result set is small. **Figure 6.14** depicts the memory usage for the queries in **Figure 6.12**. 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 6.15**. The previous conclusions are supported by this set of experiments as well.

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)



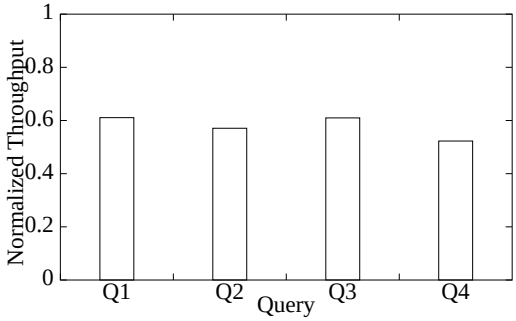
Q1: `/dblp[inproceedings]/article/title`

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 6.15: Complex Queries on DBLP Dataset



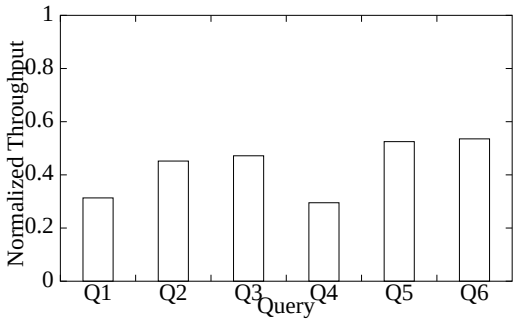
Q1:`//regions/samerica[//payment and //mailbox[//from]]`
`//item[quantity>=2 or shipping]/name`

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 6.16: Complex Queries on XMark-l Dataset




```
Q1: //reference[//refinfo[year=1981 and citation]]//author
Q2: //ProteinEntry[organism/source="human" and header/accession="A31764"]
        //protein//name
Q3: //ProteinDatabase[//reference[refinfo[//volumn=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[volumn and year]]]
        //ProteinEntry[header/uid]//genetics
```

Figure 6.17: Complex Queries on PSD Dataset

and the XMark-l dataset(1172MB). The results are illustrated in **Figure 6.16** and **Figure 6.17**.

For queries on the XMark-l dataset, we used several queries different than the queries in Figure 6.11 and 6.13. 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) are listed below.

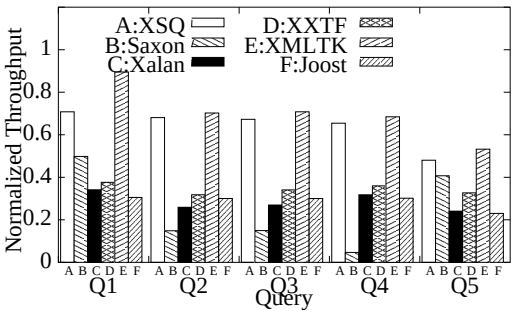
Q1:	98,229	Q4:	161,058,001
Q2:	37	Q5:	12,936,858
Q3:	44	Q6:	16,106,474

As Figure 6.17 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.

6.5.3 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

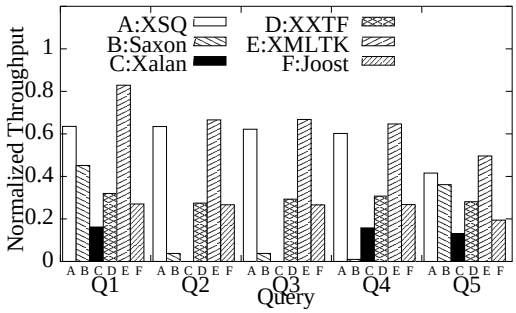


Q1:`/site/regions/samerica/item/name`
Q2:`//regions//item//name`
Q3:`//regions//name`
Q4:`//name`
Q5:`//regions`

Figure 6.18: Simple Queries on XMark-s Datasets

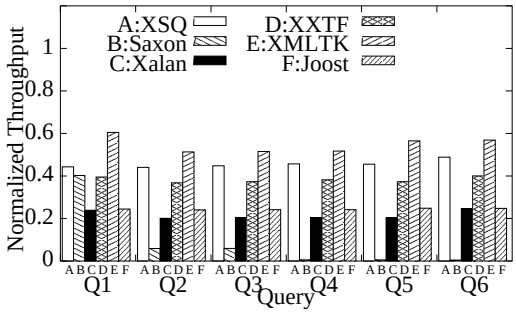
number of location steps and the number of closure axes.

We first tested a set of queries with different number of location steps on XMark-s. The results are depicted in **Figure 6.18**. 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 queries since there are only a few `regions` elements but thousands of `name` elements in the dataset.



Note: Queries are the same as in Figure 6.18.

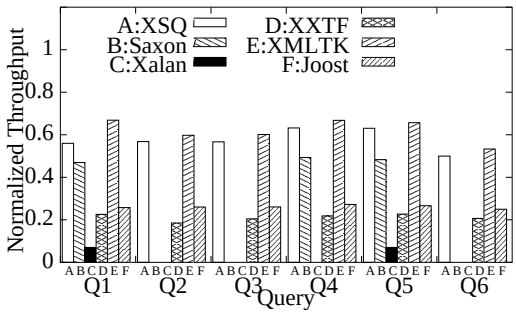
Figure 6.19: Simple Queries on XMark-m Datasets



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 6.20: Simple Queries on NASA Datasets

We tested the same queries in Figure 6.18 on the XMark-m dataset. The results are illustrated in **Figure 6.19**, which are similar to those illustrated in Figure 6.18. 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 6.19 with the queries in Figure 6.13 and Figure 6.15, it seems that Xalan's method to handle multiple closure axes cannot scale up to large datasets.

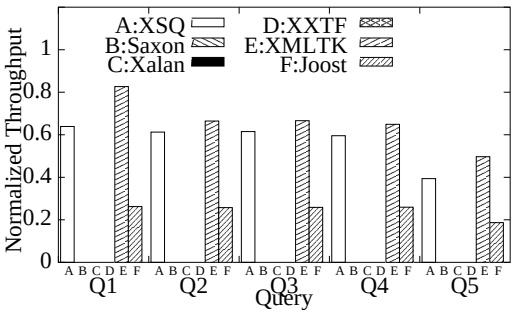


Q1:`/dblp/article/title`
Q2:`//dblp//article//title`
Q3:`//article//title`
Q4:`//phdthesis//title`
Q5:`//phdthesis`
Q6:`//title`

Figure 6.21: Simple Queries on DBLP Dataset

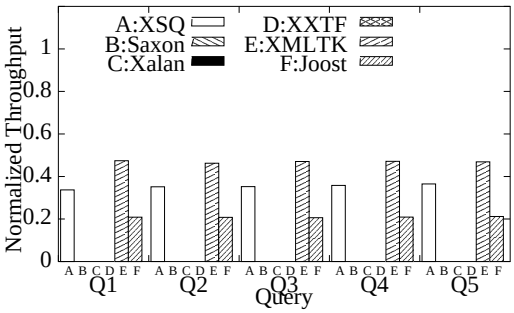
Simple queries with different number of location steps and closure axes are also tested in the NASA dataset and DBLP dataset. The results are depicted in **Figure 6.20** and **Figure 6.21**. Our previous conclusions, e.g., XXTF and the streaming systems are not sensitive to the number of location steps, 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 6.8, 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 6.8. Therefore, streaming systems need to process more events in a denser dataset, and the throughput is in general smaller in the denser datasets than in the XMark datasets.



Note: Queries are the same as in Figure 6.18.

Figure 6.22: Simple Queries on XMark-l Datasets



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 6.23: Simple Queries for PSD Dataset

6.5.4 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 6.18, 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 6.22**, illustrate similar pattern as Figure 6.18 and Figure 6.19. For example, XMLTK performs best for queries

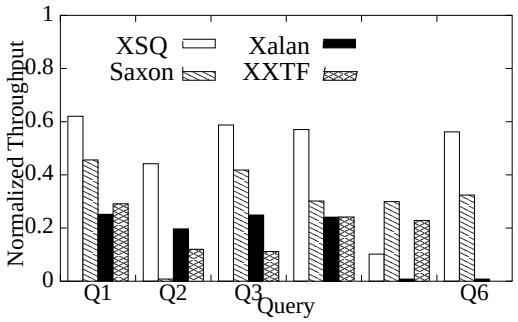
without closure axes.

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 6.23. 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 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.

6.5.5 Processing Boolean Operators

We use the next set of experiments to explore how boolean operators 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 6.24 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.

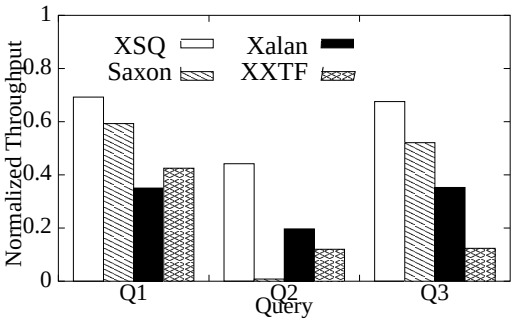



```
Q1://dataset[//initial]//reference[source//publisher]//name
Q2://dataset[//initial and //date[year>=1950]]//name
Q3://dataset[//initial and //date[year>=1950]]
         //reference[source//publisher]//name
Q4://dataset[//initial and //date[year]]
         //reference[source//publisher and //related]//name
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 6.24: Queries with AND Operators

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`.

XXTF, a join-based system, is sensitive to the number of subqueries, since it needs to generate more intermediate results. However, for subqueries that does not

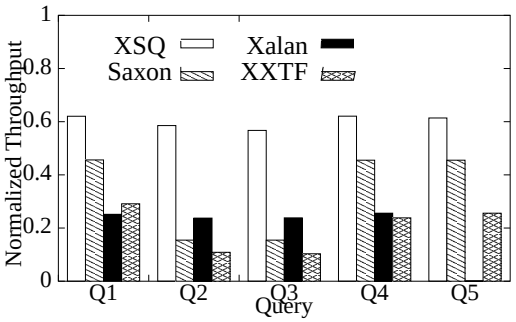


Q1:`/datasets[not(dataset)]//name`
Q2:`/datasets[//initial and //date[year>=1950]]//name`
Q3:`/datasets[//initial and //date[year>=1950] and not(dataset)]//name`

Figure 6.25: Queries with NOT Functions

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 6.25 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 expected that some predicates that can be decided earlier affects 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 6.24), but its throughput increases significantly in `Q3` where the additional subquery `not(dataset)` can

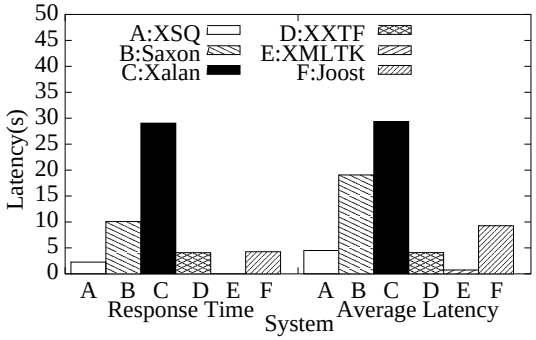



```
Q1://dataset[//initial]//reference[source//publisher]//name
Q2://dataset[//initial or //date[year>=1950]]
         //reference[source//publisher or //related]//name
Q3://dataset[//initial or //date[year>=1950] or //altname]
         //reference[source//publisher or //related or //author]//name
Q4://dataset[//initial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

Figure 6.26: Queries with OR Operators

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 6.26**. 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



Query:`//dataset//reference//name`

Figure 6.27: Output Latency

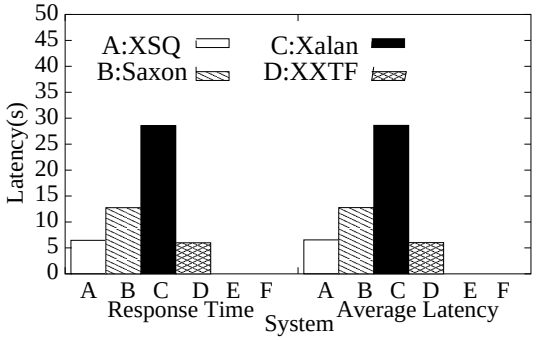
system.

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.

6.5.6 Output Latency

We also tested the output latency for the systems on the NASA dataset. The results are illustrated in **Figure 6.27** and **Figure 6.28**. 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 both metrics for the Java-based systems.) The query used in Figure 6.27 has no predicate. The query used in Figure 6.28 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



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

Figure 6.28: Output Latency

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.

Chapter 7

XPath Queries with Reverse Axes

We describe in this chapter our method to evaluate XPath queries with reverse axes. Rewriting any XPath query into an equivalent *single-step* form and encoding the pending predicates in a dependency graph, the new method is efficient for both queries with and without reverse axes. Moreover, it needs only one pass of the data, even if there are arbitrary number of reverse axes in the query. Another desirable feature of the method is that the results are emitted as soon as they are available. We give a detailed performance evaluation of our implementation of the method in the XSQ system and other XPath processors.

7.1 Introduction

Current navigation-based evaluation methods are very inefficient when evaluating queries with reverse axes such as **parent** and **ancestor**, as we illustrate in the experimental results. The fundamental difficulty caused by the reverse axes is that they incur bottom-up traversal in the document tree while the tree is usually traversed in pre-order. In streaming environment, since seeking-back in the stream is usually not allowed or very expensive, this difference seems to be irreconcilable.

Nevertheless, reverse axes are important for the users. Firstly, it empower the user to specify more complex patterns. Without reverse axes, XPath queries can

only specify tree patterns, while with reverse axes the pattern could be a graph instead of a tree. The details are explained further in this chapter and a simple example heres shows the difference.

Example 13 *Some queries cannot be easily specified without reverse axes. Suppose we have a book dataset in which books are grouped by the publishers, which means that the book elements are children of the publisher elements. If we want to find a book that is either about XML or is published by O'Reilly, we have to use the* `parent` *axis:*

`//book[subject="XML" or parent::publisher="O'Reilly"].`

For such queries that the ancestor is optional, reverse axes are needed.

The reverse axes are also very convenient for user to specify queries that can fit data of various DTDs or schema. For example, in the scenario of information dissemination, it is very likely a query issued by the user will be applied to heterogeneous data sources.

Example 14 *It is very likely that different book datasets organize the books in different ways. Suppose we want to find books that are written by W3C and published by O'Reilly. However, the books may be grouped by the publisher, by the author, by both (in either order), or not grouped at all (e.g., the author and the publisher are children of the book instead of ancestors). With reverse axes, we can issue this query as:* `//book[publisher="O'Reilly" or author="W3C" or ancestor::publisher="O'Reilly" or ancestor::author="W3C"].`

Moreover, some queries are more naturally composed using reverse axes. Since

the last location step always specifies the elements desired by the user, it is sometime more natural to think the ancestors as the prerequisites of the desired elements, which can be specified as a predicate instead of in the location path.

Example 15 *In natural language processing, it is usually more straightforward to specify a child element in a parse tree of a sentence and refer to its ancestors. For example, the following query asks for the noun phrase that has a noun "book" and appears in a top level verb phrase (i.e., its parent roots the parse tree) and contains a verb "read" (N is for noun, NP for noun phrase, and etc.):*

```
//NP[ancestor::VP[parent::root and //V=read] and //N=book]]
```

This query can be written without reverse axes, but its semantics are retained more explicitly using this form.

If the users cannot use reverse axes directly in the XPath query, they have to either specify equivalent queries without these features or program on some intermediate result to get the final result. These approaches usually cause more management overhead and hard to maintain afterward. There are also methods proposed [56] to rewrite the XPath queries with reverse axes into equivalent queries without reverse queries. However, the method either generates equivalent queries with join operations, which are expensive and difficult to evaluate (especially in streaming environment), or generates exponential number of queries (connected by disjunctions) with respect to the size of the query.

To process the reverse axes in streams, one solution is to buffer all the data needed to evaluate the query and postpone the evaluation, which is used in the

XAOS system [7]. There are three main limitations of this approach. First, in the case of infinite stream, the evaluation may be unnecessarily postponed infinitely. Second, even if the predicate for an element could be evaluated, the method still unnecessarily buffers the contents used to evaluate that predicate. Third, even if the results can be determined based on currently available data, the correct result is not sent to output and the once potential results (that can be proved not in the result) are still buffered. Because of these limitations, this type of solution is inefficient and not viable for environments that need to process infinite streams or desire lower output latency.

We introduce our method to process XPath queries with reverse axes in this chapter. In Section 7.2, we introduce the method to rewrite every XPath query into an equivalent query that has only one location step. In Section 7.3, we build an XPath query tree from the rewritten query and process the pending predicates using a hierarchical index. Section 7.4 gives the performance study of our implementation and other popular XPath processors.

7.2 Single-step Normal Form

We first rewrite any XPath query into an equivalent *single step normal form* in which the query and all the subqueries have only one location step. The single-step normal form works as an internal universal representation of the query. Therefore, our algorithms process a standard query format while can handle queries with complex structures.

The rewriting method is based on two facts that follow the XPath semantics:

- Query `A/B` is equivalent to query `B[parent::A]`.
- Query `A[B/C]` is equivalent to query `A[B[C]]`.

Two queries are equivalent if and only if they have the same result on any given data.

In general, we apply the following two rules repetitively to the query until the rules are not applicable. In the first rule, a_i, n_i, and p_i is the axis, node test, and predicate for the ith location step, respectively. a_i^r is the reverse counterpart of axis a_i, e.g., $descendant^r$ is $ancestor$. The `ROOT` stands for the document root of the XML data.

$$a_1\texttt{::}n_1\texttt{[}p_1\texttt{]/}a_2\texttt{::}n_2\texttt{[}p_2\texttt{]}$$
$$\Longleftrightarrow\ n_2\texttt{[}p_2\ \texttt{and}\ a_2^r\texttt{::}n_1\texttt{[}p_1\ \texttt{and}\ a_1^r\texttt{::ROOT]]}$$

It is obvious that the two queries return the same set of nodes. We can apply the above rule repetitively on the left-most two location steps of the XPath query until there is only one location step left.

In the second rule, we rewrite every subquery in the predicates into a single location step expression. Since a predicate may contain several subqueries that are connected by boolean operators, e.g., `A[`Q_1 `and` Q_2`]`, we apply this rule to every subquery separately.

$$n_0\texttt{[}a_1\texttt{::}n_1\texttt{[}p_1\texttt{]/}a_2\texttt{::}n_2\texttt{[}p_2\texttt{]/...]}$$
$$\Longleftrightarrow\ n_0\texttt{[}a_1\texttt{::}n_1\texttt{[}p_1\ \texttt{and}\ a_2\texttt{::}n_2\texttt{[}p_2\texttt{]/...]]}$$

`/A[a:B[C]//D]/a:E[//F]`

$\Longleftrightarrow$ `//E[//F and //A[a:B[C]//D and p:`$ROOT$`]]`

$\longleftrightarrow$ `//E[//F and //A[a:B[C and //D] and p:`$ROOT$`]]`

Figure 7.1: An Example of the Rewriting Process

We can apply this rule on every subquery recursively so that every subquery connected by the boolean operators has only one location step.

After the above rewriting processes, the XPath expression is transformed into the single-step normal form a`::`n`[`p`]`. The predicate p is also composed by only single-step subqueries that are connected using boolean operators and parentheses. Figure 7.2 illustrates an example of the rewriting process. The first rule is used in the first step and the second rule is used in the second step. In the example and future discussion, for the clarity of the presentation, we do not distinguish "descendant::" from "//", which is the abbreviation of "/descendant-or-self::node()/". Similarly, we consider the reverse axes of "//" as "/ancestor::" but not "/ancestor-or-self::node()/". Moreover, we abbreviate "parent::" as "p:" and "ancestor::" as "a:".

Absolute subqueries The subquery used in the predicate of an XPath expression may be an absolute subquery, whose evaluation uses the document root as the context rather than the element selected by the node test of the expression. For example, query `A[/descendant::B]` returns all the `A` element if the document root has a `B` descendant. For absolute subqueries such as `A[/p]`, we rewrite it into the

form `A[root::ROOT/p]` so that above rewriting process can be applied. The *root* axis is a special reverse axis that always return the document root, which always has the name *ROOT*. The reverse of the root axis is the descendant axis. (We may name the reverse as root-descendant axis so that the $(a^r)^r == a$ relation holds.)

7.3 Streaming Evaluation Algorithm

After an XPath query is rewritten into a single-step normal form, we then create the **XPath Query Tree (XQT)** for the query, as described in Section 5.2. The child of the `ROOT` node is always the output node since we create the tree from a single-step normal form, i.e. there is only one *trunk* node.

The main-memory evaluation algorithm described in Section 6.3.1 can be revised to process queries with reverse axes. First, in the `Evaluate(`e`,`p`)` function that evaluates the predicate p for element e (please see Algorithm 7), we need to check ancestors as well instead of only descendants. Second, the `MatchPath` and `ExistTruePath` functions need to consider the cases where the path may contain backward segments. These revisions are easy to make when the DOM tree is built in the main memory. In this section, we focus on how to evaluate queries with reverse axes over streams.

7.3.1 Dependency Between Elements

Similar to the method used in Chapter 6, we evaluate the queries by marking the elements with the XQT nodes that they may match. However, since we allow

reverse axes in the query, the method used in Chapter 6 cannot be applied here directly. The reason is that, without reverse axes, the predicate of an element can always be determined when the end tag of the element is encountered. This condition does not hold with reverse axes in the predicates. The following example illustrates the difference.

Example 16 *Consider query* `A[//B[a:C[//D]]]`*, when can we determine an element* `A` *in the stream does not match the XQT node* `A`*? We want to determine that there cannot be any* `B` *descendant of this* `A` *element that can match the* `B` *node. There may be following possible situations:*

- *At the end of* `A`*, we have not seen any* `B` *descendant.*
- *We have seen some* `B` *descendants, but none of them has a* `C` *ancestor.*
- *There are some* `B` *descendants whose* `C` *ancestor is also an ancestor of the* `A` *element. We have not seen any* `D` *descendant of that* `C` *ancestor yet.*

For the first two cases, we can determine that there cannot be such `B` *elements at the end event of the* `A` *element. For the third case, the decision can be made only at the end of the* `C` *ancestor.*

With reverse axes in the query, to evaluate the *validation expressions* in the streaming environment, we firstly define the notion of **dependency** between the elements.

An element e **depends** on element e' if and only if e is marked with XQT node n and e' is marked with the child of n, n' (i.e., e' is used to evaluate the predicate

of e). Recall that e being marked with n implies that e satisfies the pattern of n but have not satisfied the predicate of n yet (please see Section 6.3.2). In the above example, in the third situation, the `A` element depends on the `B` descendants, which in turn depends on the `C` ancestor. The `C` element will not depend on its `D` descendants: as soon as we encounter a `D` descendant in the stream, the `C` element can be marked with a "+", i.e., matches with the `C` XQT node.

7.3.2 Hierarchical Index

We now introduce a **hierarchical index (HIndex)** to store the dependencies among the elements and to facilitate the evaluation process. Without reverse axes in the query, the predicate of an element can always be determined when the element ends. Therefore, a stack that holds only the open elements suffices to keep the undecided elements. With reverse axes in queries, we use the HIndex to hold those undecided elements, i.e., those with a mark but no "+" sign.

The HIndex is essentially a directed dependency graph. The nodes in the graph are the undecided elements. An element e depends on another element e' if and only if e' is a parent of e in the dependency graph. A naive solution is to create a node in the graph for every element in the document. However, due to the existential semantics of XPath predicates, we can design a more efficient index.

Consider the query `//A[//B[a:C[//D] and a:E]]`. An element `A` depends on all its `B` descendants, which in turn depend on their `C` ancestors. Note that the `B` elements does not depend on their `E` ancestors since whether they have an `E` ancestor

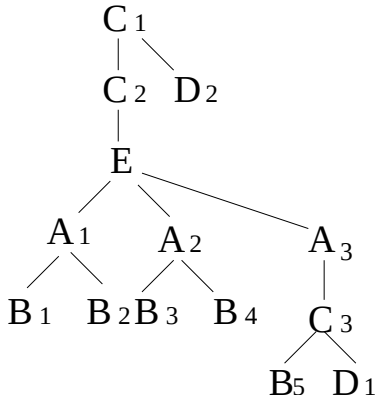


Figure 7.2: A DOM tree of an XML document

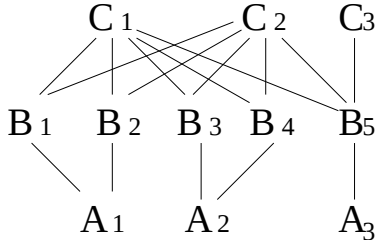


Figure 7.3: The dependency graph

can be determined at the time we encounter them. Figure 7.2 illustrates the DOM tree of an example document. The streaming form of this document is essentially a preorder traversal of its DOM tree. We distinguish elements with same name by subscriptions. The dependency graph when we process the begin event of the B_5 element is shown in Figure 7.3.

We modify the above dependency graph into a more compact and efficient HIndex as follows. Firstly, for every element e with mark n, we create a **socket** for

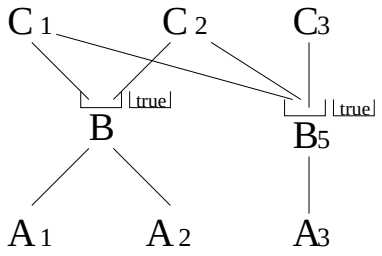


Figure 7.4: The HIndex after the B nodes are combined

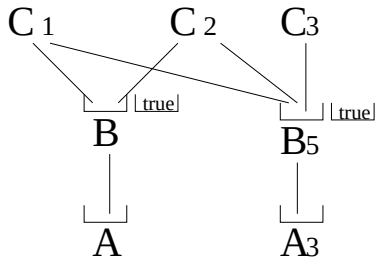


Figure 7.5: The final HIndex

e in the index for every variable $a_i :: n_i$ in the the validation expression of n. The $socket_i$ is labeled with the current value of $a_i::n_i$ and contains all the edges between e and element e' with mark n_i. We group these edges using the socket so that when one element connected from these edges is marked n_i+, we label this socket with *true* and remove all the edges from this socket. The $socket_i$ of element e created for $a_i::n_i$ is labeled *false* in the following cases:

- If a_i is a reverse axis, $socket_i$ is labeled as *false* after the begin event of e when it is empty and the current value is na.
- If a_i is the attribute axis, $socket_i$ is labeled as *false* after the begin event of e when the predicate involves the attribute evaluates to false.
- If a_i is a forward axis, $socket_i$ is labeled as *false* after the end event of e when it is empty and the current value is na.

It is easy to see that $socket_i$ labeled with *true* do not need accept new edges because of the existential semantics of XPath predicates. If $socket_i$ is labeled with *false*, it cannot accept new edges as well since it is labeled as *false* if and only if the corresponding query has already been falsified.

Two index nodes can be combined when their corresponding sockets either

have the same label or contain the edges from the same set of parents, which implies that they depend on the same set of elements. The following example shows how this combination shrinks the index drastically.

Example 17 *We have seen the dependency graph for the following query:*

```
//A[//B[ancestor::C[//D] and ancestor::E]]
```

When we evaluate it over the document shown in Figure 7.2. The corresponding HIndex, which is shown in Figure 7.5, is much smaller than the dependency graph.

Firstly, since the first four `B` *nodes depend on the same* `C` *ancestors, they can be combined together. The result is shown in Figure 7.4. Since then the first two* `A` *nodes are now depending on the same node, they are combined as well. The final HIndex is shown in Figure 7.5.*

The sockets of each index node are also shown in Figure 7.4 and Figure 7.5 using the symbol $\sqcup$*. Edges are grouped by sockets. The socket labeled with true is created for the* `ancestor::E` *item in the validation expression of the* `B` *nodes. Since we have seen an* `E` *ancestor when we encounter the* `B` *elements, these sockets are labeled with true when we create them.*

After the combination, the propagation of the result is very efficient. When we encounter the D_2 *child of the* C_1 *ancestor that the combined* `B` *node depend on, the* C_1 *node is marked true and the corresponding socket of the combined* `B` *node is labeled* `true`*. Since the other socket of the* `B` *node has already been labeled true, the validation expression of the* `B` *node now evaluates to true. The only socket of the combined* `A` *node will then be labeled true and the contents will be sent to output.*

Similar process also happens earlier when we encounter the D_1 child of C_3 element and the A_3 node is marked true. However, since there are potential results that cannot be determined before this A_3 element, we have to wait until later and output the three `A` *elements together.*

Note that we cannot simply connect those `C` *ancestors to the combined* `A` *node although essentially the* `A` *elements are now depending on those* `C` *ancestors. Since the predicate may be in the form of* `B[not(ancetor::C//D] and ancestor::E]`*, the evaluation of the validation expressions at node* `B` *cannot be omitted. Optimizations can be done here if there is no* **NOT** *operators in the predicate.*

The combining process shown in above example is not the actual process during runtime. We do not combine the index node after we have created all of them. When we want create a new index node, if we find that there is an index node that depends on the same sets of parent and has the same label for the empty sockets, we do not create the new index node but use the existing index nodes instead.

7.3.3 Create HIndex Dynamically

We create the HIndex dynamically during runtime. There are three operations on the HIndex: *create, update*, and *remove*.

The **create** operation creates a new node in the HIndex. When we mark an element e with node n in the XQT, a new index node for e is created in the HIndex. If e depends on an ancestor element e', since e' must be already in the HIndex, we connect e' to e's corresponding socket (and e' becomes a parent of e in the index).

If e depends on its descendants, which is currently not in the HIndex, we just label the corresponding sockets as *na* (and no edge is created).

If n is a branch node, e may also be depended on by other nodes. If the edge between n and its parent in the XQT is a reverse axis, no edge is created since e may be depended on only by its descendants. Those edges are created when we create the index nodes for the descendants. If the edge is a forward axis, which means one of e's ancestors depends on e, we connect e to e''s corresponding socket. Therefore e becomes a parent of e' in the HIndex (note that e' is an ancestor of e in the document).

The matching process using XQT is modified accordingly so that if the socket of node e that corresponds to $a_i::n_i$ is labeled *true* or *false*, we no longer match elements with tag n_i for element e.

The **update** operation updates the marks of the HIndex nodes and propagate the update as needed. When a HIndex node e is being marked with $n+$ or being unmarked, for all the HIndex children of e, the socket of all the children that connects e is labeled TRUE or FALSE accordingly and all the edges in that socket is removed. The validation expression of those children is then evaluated. If the result of the re-evaluation is either *true* or *false*, the child is marked with the new value and its children is updated accordingly.

The **remove** operation is executed when any node is marked *false* after an update operation. (Recall from Section 7.3.2 that an index node is marked false in three cases.)

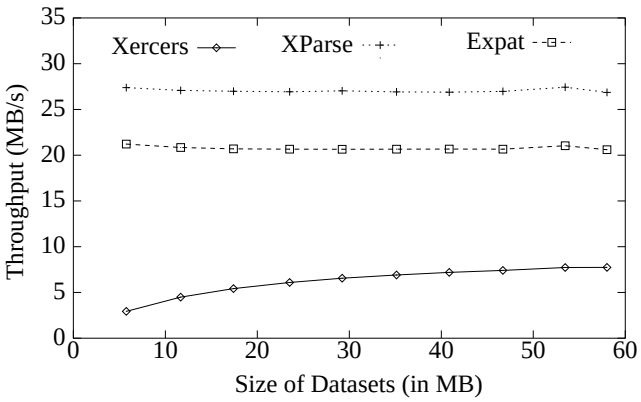


Figure 7.6: Throughput of the pure parsers

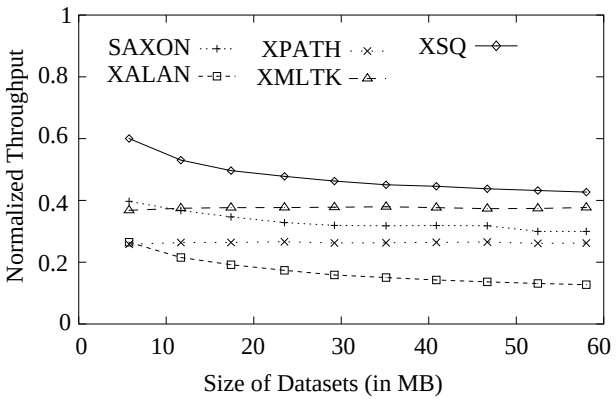
7.4 Performance Evaluation

We present in this section the results of the performance study we conducted for the algorithm described in this chapter. The study is based on our implementation of the algorithm using Java SDK 1.4.0_01. As part of the XSQ project [59], the new system is named as XSQ-R.

7.4.1 Experiment Setup

Our goal of the performance study is to examine that whether XSQ-R has high throughput and low memory footprint for data and queries of different sizes. We also study the relation between the throughput of the system and the characteristics of the queries.

We also compare the performance of XSQ-R with other systems that can process XPath queries: Saxon (`http://saxon.sourceforge.net/`), Xalan (`http://xml.apache.org/xalan-j`), XPATH from XMLTaskForce (`http://www.xmltaskforce.com`), and XMLTK [4]. XMLTK (version 1.01) is a set of XML tools developed at



Q0:`/site`

Figure 7.7: Throughput of Q0 that returns the whole dataset

University of Washington. The xrun program in the toolkit can evaluate XPath queries on large XML datasets. It uses a DFA generated from the XPath query that takes the SAX events as input and return the results of the query. Since it does not support XPath queries that need buffering, we use it only in experiments using queries without predicates and reverse axes. Saxon (version 6.5.2) and Xalan (version 2.4.0) are two broadly used high performance XSLT processor. XPATH is the implementation from XMLTaskForce of the polynomial algorithms introduced in [32]. All three systems are main memory systems that need to build the DOM tree in the main memory before evaluation.

The main metric we use is the normalized throughput of the systems, which is the raw throughput of the system normalized by the throughput of the corresponding *pure parser* that parses the data but does nothing else. We wrote a pure parser in Java using Xerces2 Java Parser 2.4.0 Release, which is used as the XML parser for XSQ-R, Saxon, and Xalan in the experiments. We also wrote another pure parser in C using Expat parser, which is used by the XPATH program. XMLTK 1.01 provides

a parser program named XParse that parses the document and count the number of elements in the document. Since the counting process only needs one statement at the begin event of every element, which should be very fast operation compared to the parsing process, we use the XParse program as the pure parser of the XMLTK system.

The throughput of the pure parsers are shown in Figure 7.6. We can see that the throughput of the C parsers are roughly constants while the throughput of Xerces has a converging process. Since the preprocessing time of the Xerces parser, including Java Virtual Machine (JVM) loading and system initialization, becomes less significant as the size of the data increases and the parsing time becomes the dominant factor in the throughput. In the contrast, the running time of the C parsers are always dominated by the parsing time, which is proportional to the size of the data.

We generate ten datasets using XMark benchmark program with the scale factor set to 0.5 to 5, step by 0.5. The sizes of result dataset range from 5.7MB to 58MB. For most queries, we show the normalized throughput for all the datasets. We use the dataset of size 58MB (of scale factor 5) when we compare the throughputs of different queries on the same dataset.

We ran all the experiments in a PIII 900MHz PC with 1GB of main memory running Redhat 7.2 distribution of GNU/Linux (kernel 2.4.9-34). The maximum memory the Java virtual machine can use was set to 512MB. The running time is obtained using the GNU TIME(1) tool. Every data point presented is obtained from the average of the results of ten runs.

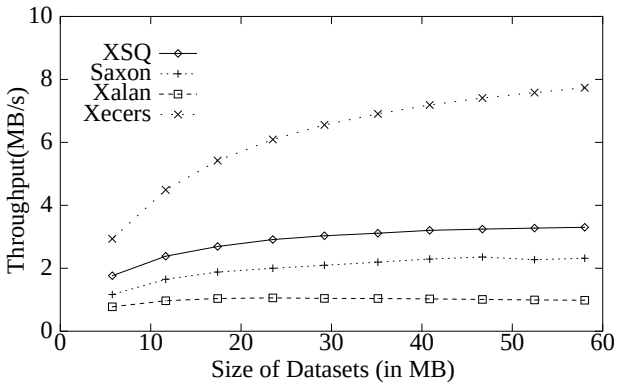


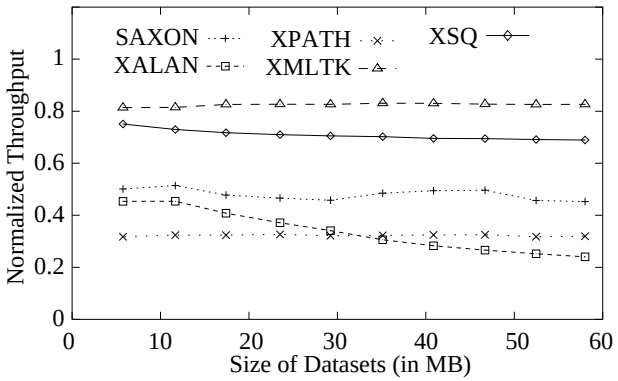
Figure 7.8: Absolute throughput of Q0

7.4.2 Simple Queries

In this chapter, we claimed that although we rewrite the query into a single-step normal form, the performance of the queries that have simple structures are not compromised. The first set of experiments proves this claim using simple queries without predicates and reverse axes. Such queries are essentially regular expressions, which can be evaluated most efficiently using automaton-based approaches.

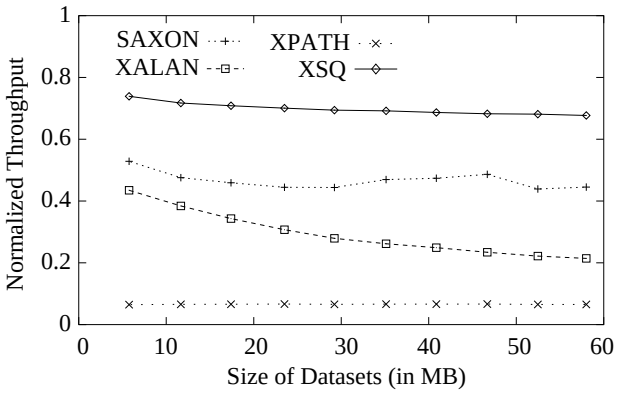
The query `/site` returns all the items in an XMARK dataset. The normalized throughputs are shown in Figure 7.7. We can see that XSQ-R is very efficient when handling this simple query that is requires a large mount of disk IO.

The systems using the Xerces as the parser show similar converge curves, while the normalized throughput of the systems written in C are almost constant. As we show in Figure 7.6, the throughput of the C parsers are almost constant, which implies that the XPATH and XMLTK also have constant throughput for this query. For the Xerces-based systems, we show the throughput in Figure 7.8 together with the throughput of the pure parser using Xerces. If the actual throughput of the query engine after preprocessing is T (MB/s), the preprocessing time is P (s), and



Q1: `/site/regions/samerica/item/name`

Figure 7.9: Throughput of a query with only child axes

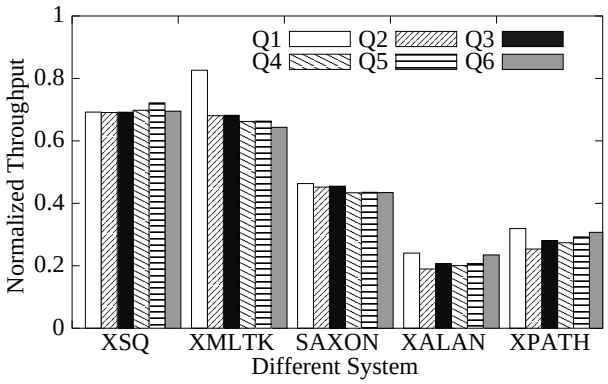


Q7:
```
//regions/samerica[d::payment and d::mailbox[//from]]
                //item[ quantity>=2 or shipping]/name
```

Figure 7.10: Throughput of a query without reverse axes.

the size of the dataset is S (MB), the throughput we compute is $\frac{S}{P+T\times S}$, which converges to the actual throughput as S increases. However, since the throughputs converge at different pace and to different values, the normalized throughputs also show similar converging curves. It is also the reason we use the throughput of the largest when we compare different queries, since the normalized throughput has almost converged at the point.

XSQ-R is very efficient when evaluating queries with very simple structures. Figure 7.9 shows the performance of the systems for a simple query with five lo-

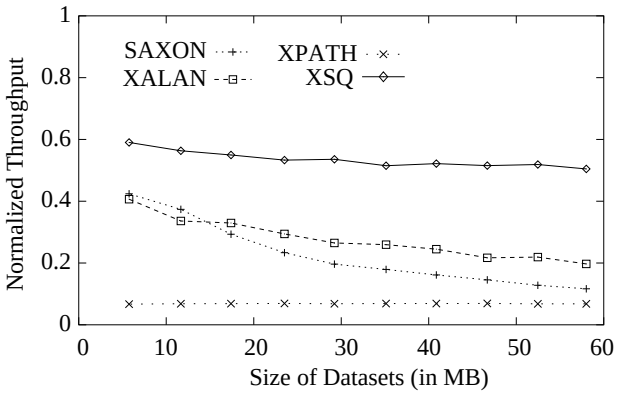


Q1: /site/regions/samerica/item/name
Q2: //site//regions//samerica//item//name
Q3: //regions//samerica//item//name
Q4: //regions//item//name
Q5: //regions//name
Q6: //name

Figure 7.11: Throughput of queries with closures of different length

cation steps without predicates and closure axes. Since the query can be processed by a very simple Deterministic Finite Automata (DFA), we expect that the automaton-based system (such as XMLTK) is most efficient for this query. In our method, we rewrite this query into the form `/descendant::name[parent::item[parent::samerica[parent::regions[parent::site[ parent::root]]]]]`, which means that whenever we encounter a `name` element, we have to check the matching stack of the HIndexNode that corresponds to `item` to see if it is empty. In an automaton-based system, if this `name` does not match the pattern, it is ignored since the state that accepts the start event of the name element is not the current state of the DFA at the time. Nonetheless, as we can see in Figure 7.9, XSQ-R is still very efficient when evaluating this query.

We then show closure axes are processed differently in different systems. We show in Figure 7.11 several queries of different length on the XMark dataset of size 58 MB (scale factor 5.0). Although `Q1` and `Q2` has the same result set for this dataset,



Q8: `//item[quantity=1 and d::listitem[a::regions/samerica]]/name`

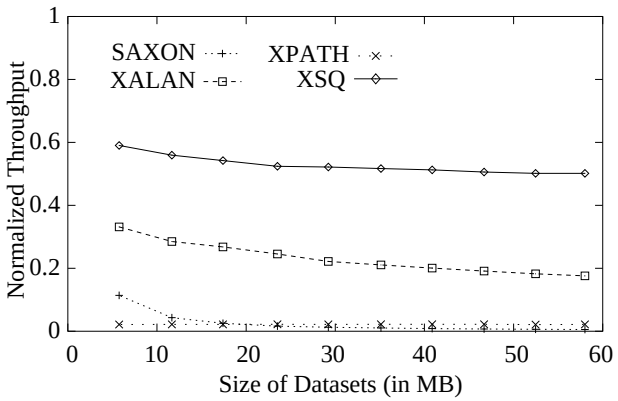
Figure 7.12: Throughput of a query with a reverse axis in the predicate

we can see that XMLTK, Xalan, and XPATH has significant performance difference between these queries. XSQ-R and Saxon, in the contrast, are not very sensitive for the closure axes. In the case of XSQ-R, more closure axes only means larger current node set since a closure axes keeps the parent index node in the current node set when a child node is added while the a non-closure axis removes the parent node from the index. However, the size of the current node set makes almost no difference in the performance since the current node set is organized as a hash table and the only operation we have is hashed lookup.

7.4.3 Complex Queries

In this set of experiments, we show how XSQ-R is efficient when evaluating queries of more complex structures and with reverse axes. Since XMLTK does not support such queries, we do not include it in the experiments.

Figure 7.10 shows a query that has nine location steps in total from the query and predicates and does not use reverse axes. Comparing to the previous queries



Q9: `//listitem[a::item[d::price>10 or d::quantity=1]`
`or a::annotation[happiness>8]]`

Figure 7.13: Throughput of a complex query with two reverse axes in the predicate

with less location steps and simpler structures (cf. Figure 7.9), XPATH performance worse since its approach generates more intermediate result and perform more join operations. In the contrast, the number of location steps does not cause much difference in the performance of the Xerces-based systems since they usually evaluate the query by traversing the document tree. Since there is no reverse axes in the query, which means the predicates and the query can be evaluated naturally in one traversal of the document tree, the throughput of this query is similar to query `Q1` which has a much simpler structure.

The next complex query has one reverse axes in the predicate. We can see from Figure 7.12 that the Saxon's performance degrades comparing the previous query without reverse axes. XPATH perform OK since it treats reverse axes almost the same as forward axes. However, it seems that Xalan has some optimization for reverse axes so that its performance is almost the same as the previous query.

Next query with two reverse axes in the predicates proves our speculation. Saxon seems to be evaluating the reverse axes in a very straightforward manner

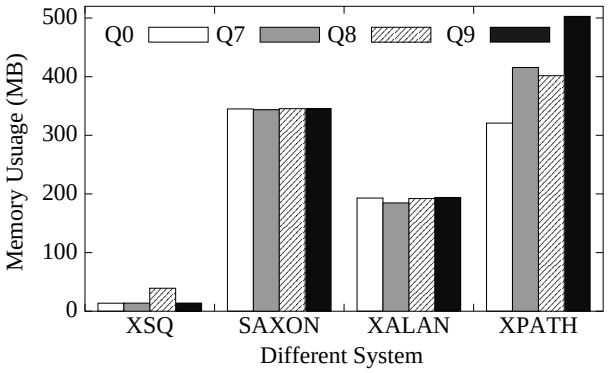
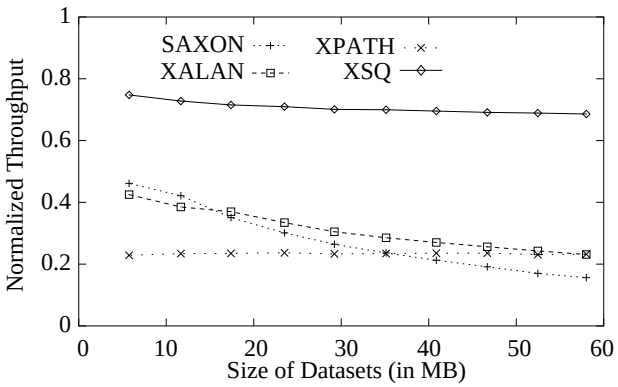


Figure 7.14: Maximum memory allocated for queries over the 58MB dataset

since more reverse axes incurs more traversal of the document tree. Xalan still shows similar performance as the previous two queries. XPATH performance bad here not because the reverse axes, but because the intermediate result generated by the predicates (such as `//listitem/ancestor::item`) almost covers the whole dataset. Figure 7.13 shows the maximum memory allocated by each system when evaluating the three queries. The left most bar is the memory allocated when evaluating query `/site` when no intermediated result should be generated. We can see the more memory allocated by XPATH, the smaller the throughput of XPATH is. XSQ-R uses more memory in Q8 to buffer some of the data since the predicate `ancestor::regions/samerica` can be evaluated after around 50% of the dataset has been processed (samerica is the sixth continent listed in the data).

7.4.4 Number of Reverse Axes

In this set of queries, we illustrate that different systems handle reverse axes differently. Some systems (XSQ-R and XPATH) are more efficient when processing queries with reverse axes.

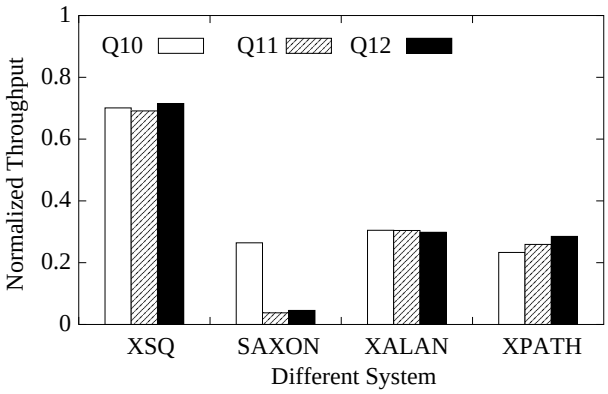


Q10: `/d::name[p::item/p::samerica/p::regions/p::site]`

Figure 7.15: Throughput of a query with reverse axes in the predicate

Figure 7.15 illustrates the normalized throughput of the systems when they evaluate a query with several `parent` axes in the predicate. We can see that all the systems perform well for this query. Although Saxon performs worse in the previous experiments when there are reverse axes in the predicate, it evaluates this query faster since the evaluation of a `parent` axis for every `name` element in the document tree requires exactly one visit to the parent node. As we can see in Figure 7.16, which shows the throughput of the systems when evaluating the queries over the 58MB XMark dataset, when the reverse axes in the predicate are closure axes, the performance of Saxon degrades a lot while the other systems are not affected as much.

However, the situation is different when there are reverse axes in the location steps of the query. Figure 7.17 shows the throughput of the systems when evaluating the queries over the 5.7MB XMark dataset (scale factor 0.5). We did not use larger dataset since Saxon went out of memory when evaluating Q14 and Q15 on this 5.7 MB dataset. The throughputs of both Saxon and Xalan also degrade drastically: Saxon uses more than 130 seconds to evaluate query Q16 and Q17 and Xalan uses

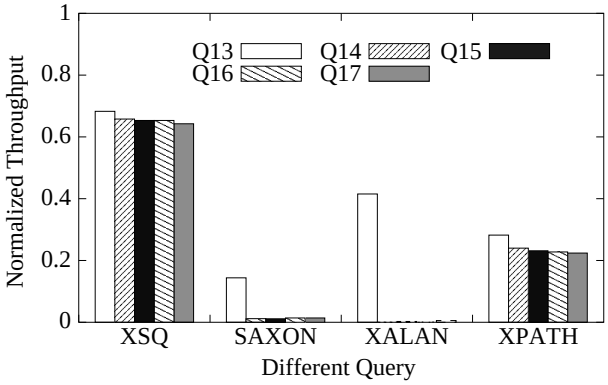


Q10:`/d::name[p::item/p::samerica/p::regions/p::site]`
Q11:`/d::name[a::item/a::regions]`
Q12:`/d::name[a::regions]`

Figure 7.16: Throughput of queries having different reverse axes in the predicate

more than one hour to evaluate query Q16 and Q17. However, Xalan does not use extra memory in these queries while Saxon seems to be generating large number of intermediate results in the process, as illustrated in Figure 7.18.

We can see from the experiments that XSLT processors such as Saxon and Xalan are not very efficient when processing the reverse axes. Although they are designed mainly for processing stylesheets that usually have more complex semantics and construction operations, it is obvious that our method could be used as the XPath processing component to improve the performance. For example, instead of traverse the whole document tree, the XSLT processor can receive the output of our XPath processor as the input.



Q13:`//text`
Q14:`//listitem/a::parlist/a::site//text`
Q15:`//listitem/a::parlist/a::description/a::site//text`
Q16:`//listitem/a::parlist/a::description/a::item/a::site//text`
Q17:`//listitem/a::parlist/a::description/a::item/a::regions/a::site//text`

Figure 7.17: Throughput for queries having different number of reverse axes

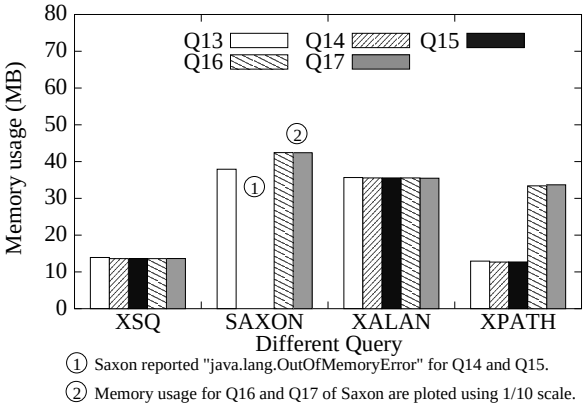


Figure 7.18: Memory usage of queries having different number of reverse axes

Chapter 8

An XPath Subscription Server

We describe in this chapter the XPaSS system, which simultaneously evaluates multiple XPath queries over possibly unbounded XML streams with arbitrary structure. Two distinguishing features of XPaSS are (1) the input is not assumed to be segmented into documents or other predefined units and (2) subscriptions are based on queries, not filters, permitting precise specification of desired results. XPaSS has applications in selective dissemination of data from syndicated Web sites, meteorological measurements, and a growing number of XML data sources. Our methods are based on grouping operations by XPath *segments*, permitting bitmapped implementations of key tasks that must be performed for each input item (SAX event). Our experimental study indicates that XPaSS scales easily to a few hundred of thousands queries with a good throughput and a small memory footprint.

8.1 Introduction

We may think of the XPaSS[1] system as one that generalizes the well-studied subscriptions servers to an environment without segmented input data and that permits queries, not just filters, as subscriptions. Alternately, we may think of XPaSS as an XPath query engine that is optimized for simultaneous execution of

[1]Multiple XPath Query-engine

a large number of XPath queries. In either case, the environment we study is the same: Data arrives in the from of an XML stream. We do not make any assumptions about the structure of the input (beyond well-formedness of XML). In particular, we do not assume that the input is segmented into documents, nor do we assume that there is a predefined set of element types that determine the type of subscription results. Subscriptions consist of XPath queries. Query results are to be continually evaluated on the input data and propagated to subscribers in a timely manner. The key considerations are the throughput and memory footprint of the subscription server.

As an example of the environment we study, consider the recent growth of Web sites with syndicated XML content. Authors (human or automated) publish streams of XML data. These streams are typically aggregated by content aggregators (e.g., `http://www.syndic8.com/`). It is natural to carry this development to the next logical step: delivery of customized content to subscribers. As another example, consider the dissemination of a large body of continually produced data (typically, *processed* instrument data). For instance, the National Weather Service continually produces data based on a large network of instruments. There is no particular segmentation of this data into documents that is universally useful. (Although the NWS structures this data into well-defined files, it is unlikely that a subscriber would want to receive such files as subscription results.) In both these examples, the number of subscribers is likely to be large enough to render naive implementations based on evaluating subscriptions individually impracticable.

Subscription servers have been studied extensively. However, the methods are

typically based on filtering, not querying, subscriptions, and their generalization to our environment is not obvious. Further, except for some recent work, attention has focused on a document-centric model with keyword-based subscriptions. There has also been much recent work on XPath (and XQuery) query engines. Although such work provides good methods for evaluating a single query (or a small number of simultaneous queries), it does not generalize easily to subscription-server scales (say, 10^5 subscriptions).

Filters vs. Queries As noted above, we distinguish between two kinds of subscriptions that an information dissemination system may support. **Filter subscriptions** are subscriptions that produce a boolean result on each segment of the incoming data stream. These segments (or documents) are predefined by the server and cannot be customized by subscribers. **Query subscriptions** are subscriptions that do not assume any particular segmentation of the incoming data stream. Instead, these subscriptions precisely specify the portions of the input that are of interest to the subscriber. Such flexibilities benefit both subscribers and publishers.

For subscribers, they are less likely to be overloaded with irrelevant and excessive data. For example, consider a server that processes subscriptions over a data stream consisting of business articles from a newswire agency. Different parts of such documents (e.g., headline, author, company name, stock symbol, recent stock price) are often tagged in a machine-readable manner. Such articles often also contain a significant amount of background information based on company filings. A subscriber may be interested in only the headlines and stock price data for com-

panies classified as computer software companies. Using a query subscription, it is possible to specify exactly such a profile. In contrast, filtering systems may narrow down the documents in the stream to a manageable number, but they would still return the remaining documents in their entirety, leaving the subscriber with the task of distilling the interesting information from them.

For publishers, they do not spend time to categorize the data into predefined categories to make suitable segmentations of the data. Categorizing is time-consuming and predefined categories may have to be evolved as new concepts emerge. Even these pre-processed data segments may not suit the users' needs, as it is very hard to predicate users' interests before hand. For example, the XML feeds from NWS[2] are currently served in many categories, such as "Forecasts", "Watch/warnings", and etc. In each category, single feeds are served for some geographical metrics, some by states and some by larger regions. Given a query subscription engine like XPaSS, all these pages can be combined into one single query interface in which users provide their queries and get the feeds they want.

Challenges Although an information dissemination environment that uses query subscriptions is attractive, it does complicate the data-processing task at the server. Consider a simple XPath query `//A[B]//C`. When it is used a filter subscription system, the engine tests for every document the *existence of one* `C` element inside, i.e., being a descendant of, an `A` element who has a `B` child. While in a query subscription system, the engine returns for this query *all* such `C` elements. In the

[2]http://www.weather.gov/xml/index.php

case some C descendants of an A element arrives earlier than the first B child of A, we need to *buffer* all those C descendants. Previous work has explored similar challenges in evaluating single XPath queries at a time.

The need to evaluate a large number of queries simultaneously makes for an even more challenging task. First, we need to identify the common features among all the queries that can be executed together and share those results for different queries. Previous work on filtering system addresses this problem by sharing common prefixed or suffixes among filters. However, when XPath expressions are used as queries, the same prefix of suffix may stand for different semantics. For example, in a filtering system, XPath expression `//A//B[//C]` and `//A[//B//C]` always return the same set of documents and share the prefix `//A//B//C`. However, in a querying system, the first query returns a set of B elements while the second returns a set of A elements. Therefore, extra mechanisms are required to handle these differences. Second, we need new mechanisms to address the buffer operations for multiple queries, which are not at all required in filtering systems but essentially for querying systems. It is obvious that creating a single buffer for every query will not scale.

Our solution to this problem is based on the idea of decomposing each XPath query into *segments*, which may be intuitively thought of as triples comprised of a pair of successive node labels in the query along with the connecting axis (/ or //). (Details are in Section 5.1.) The number of possible distinct segments in a large collection of XPath queries is relatively small, implying a high degree of expected overlap. Our method uses this overlap, along with efficient bitmapped versions of

the operations that must be performed in response to incoming data (SAX events). We note here that we do not achieve any asymptotic algorithmic improvements in this manner. However, the differences in the constants are significant, as illustrated by our experimental study!

We may summarize the main **contributions** of this chapter as follows:

- We motivate and develop the problem of information dissemination using XPath queries on streaming data of arbitrary structure. We discuss the relation of this problem to prior work on information dissemination, streaming query evaluation, and other areas and highlight the new challenges posed by this problem.

- We present methods for implementing XPath query subscription servers based on the idea of XPath query segments. These segments form a practical unit for aggregating computation across queries. Although not experimentally studied in this chapter, an interesting feature of our methods is that they permit dynamic insertion and deletion of subscriptions at run time without the need for expensive reorganizations.

- We have implemented all the methods described in this chapter in the XPaSS system. We present an experimental study of XPaSS that indicates that the current implementation scales well to a few hundred of thousands of queries on modest hardware. We discuss the main factors affecting the performance of XPaSS and highlight some promising areas for further research.

Outline This chapter is organized as follows: We first introduce a segment-based

evaluation algorithm for single XPath queries in Section 8.2. The algorithm for multiple queries is then described in Section 8.3. The experimental results are presented in Section 8.4. We conclude in Section 8.5.

8.2 Segment-based Evaluation

In streaming XML processing, we store the currently open elements in a stack. An incoming element e and each ancestor a in the stack form a pair (a, e), which may match a segment $M//N$ if a's tag matches node test M and n's tag matches node test N. An XPath query can be evaluated by linking this segment matching information. (For details of evaluating XPath using segments, please see Chapter 5.)

As we illustrate in Example 6 in Chapter 5, we can share the evaluation of segments among queries. We use this sharing idea to improve the evaluation performance when we need to evaluate multiple XPath queries simultaneously over XML streams. Existing techniques for multiple-query evaluation, in the mean while, share the evaluation effort among queries that have either common prefixes [3, 23, 15] or common suffixes [35].

We describe the segment-sharing method in Section 8.3. In this section, we describe the segment-based streaming evaluation for single queries as the first step. Extending the segment-based semantics of XPath (defined in Section 5.3), we define four types of matchings an element may have with a query node. The data structures and streaming evaluation algorithm for single queries are presented in detail next. We also discuss the correctness and complexities of the method.

We discuss in this chapter queries without value comparisons. However, our methods also apply to queries with such comparisons. For example, predicate `[name='BN']` can be approximated as `[name/child::text()='BN']`, since `text()` is essentially a special node test that matches all document node of type `text` while other regular node test (called name test) only matches document node of type `element`. Therefore, we may create a special query-tree node for `text()='BN'`, who matches the text content of a `name` element with value 'BN'. We also only describe the cases for `//` axes. Unless otherwise stated, these descriptions can be applied to the cases for `/` axes if we replace the ancestors with parents.

8.2.1 Partial Information of Matching

During streaming evaluation, we organization of the partial matching information as follows. Intuitively, we distinguish the partial information by which portion of the data it can be verified on. For example, we use the data outside of element e to determine whether there exists a matching between e and a query-tree node n, while we only use data inside e to determine whether e satisfies the predicates of n. We define the following four **matching-sets** for every element e:

- **Out**(e) contains XQT node n iff there exists a matching between e and n. We also say e **outer-matches** n.
- **In**(e) contains XQT node n iff the tag of e matches the label of n and e satisfies n's predicates. We also say e **inner-matches** n.
- **Part**(e) contains XQT node n iff n is a trunk node, e has inner- and outer-

matched n, and e has not matched n. In other word, whether e matches n depends on whether every ancestor of e in the matching satisfies the corresponding predicates. We also say e **partial-matches** n.

- **Match**(e) contains XQT node n iff either (1) n is a trunk node and e has matched n or (2) n is branch node and e has inner- and outer-matched n. For convenience, we also say e matches the branch node n in this case.

The difference of treatment to the branch nodes and trunk nodes in the Match() set comes from the difference between top-level queries and subqueries. The subquery rooted at a branch node n is evaluated in the context of every element e' that outer-matches the parent node $p(n)$ of n, be it a trunk node or a branch node in a higher-level subquery. If a descendant e of e' both outer- and inner-matches $p(n)$, evaluating the subquery rooted at n in the context of e' returns a non-empty result set. In other words, e makes e' satisfy the predicate regardless of the predicate evaluation results for the ancestors of e. For example, in Figure 5.1, element e_5 (the `price` element on line 5) matches branch node n_5 in Figure 5.2(2), since e_5 can be used to satisfy the predicate `[//price=10]` of its parent e_3.

8.2.2 Data Structures

We now describe the data structures that store necessary data for the evaluation. We describe in Section 8.2.3 the algorithm that updates these data structures dynamically upon incoming streaming data.

Stack We maintain a stack of open elements, which are elements whose start,

but not end, tags have been encountered. The stack is maintained in the natural manner: An element is pushed onto the stack upon its begin SAX event and it is popped off the stack upon its end event. Note that we allow access to the whole stack instead of only the top element.

Pending-set Suppose we have determined that element e has outer-matched a query node n. To determine whether e inner-matches n, we associate with e a pending-set, denoted as $\mathbf{Pending}(\mathbf{e}, \mathbf{n})$, when e is put on the stack. Pending(e, n) is initially set as $C(n)$, the condition-set of n. A query node x is removed from Pending(e, n) when a descendant of e matches x. When Pending(e, n) becomes empty, e inner-matches n since all predicates of n have been satisfied by e.

Buffer If stack element e outer-matches a trunk node n, we associate with e a buffer, denoted as $\mathbf{buffer}(\mathbf{e}, \mathbf{n})$, to store the elements whose matchings with the output node contain pair (e, n). In other words, whether e satisfies the predicates of n determines whether the matching is full. Note that in the implementation, we use a *queue* to store the elements in document order and store only pointers to the queue items in the buffers. The details are described in Section 8.3.5.

8.2.3 Streaming Evaluation

Our streaming evaluation method maintains the matching-sets for every open element e by responding to every incoming SAX events. Upon the start of the stream, the element e_0, corresponding to the document root and labeled as `root`, is pushed onto the empty stack. Matching-sets Out(e_0), In(e_0), and Match(e_0) are all

singleton sets that contain n_0, the root of the XQT; Part(e_0) is empty.

When the begin event of element e arrives in stream, we initialize all its feature-sets and pending-sets to empty. After its begin event, Out(e) contains all XQT nodes that e outer-matches. For each node in Out(e), the pending-set is created, to record which predicates it is still pending on.

Before the end event of an element e arrives in stream, some descendants of e may satisfy some predicates for e. We remove those satisfied predicate from the pending-sets of e. Some descendants may be potential results and depend on the predicate evaluation result of e. These descendants are put in the buffer of e.

When the end event of element e arrives in stream, we first check the pending-sets of e. If pending(e, n) is not empty, we know that e has failed some predicate and therefore we clear buffer(e, n). Otherwise, e has inner-matched n. If e is a trunk node, we should append buffer(e, n) to the buffer of an ancestor a that outer-matches $p(n)$, since whether e matches n depends on whether a matches $p(n)$. If n is a branch node, we know that e satisfies the subquery rooted at n for some ancestors, we should remover n from the pending-sets of those ancestors.

Basically, we compute the Out() sets at the begin events, In(), Part(), and Match() at the end events, when all the data needed to compute these sets are guaranteed to be available. Note that the Match() sets for trunk nodes are computed lazily at the end event of the element that outer-matches the child node of the XQT root. Only at that moment can we determine if all the elements in a matching have satisfied their predicates. The evaluation algorithm is presented in detail in Listing 12 and Listing 13 as event handlers. The stack operations are omitted from

Algorithm 12: Begin(Element e)

```
1  S ← {n|n.label matches e.tag};
2  foreach n ∈ S do
       /* p(n) is the parent node of n */;
3      if ∃ ancestor a s.t. p(n) ∈ Out(a) then
4          Out(e) ← Out(e) ∪ {n};
5          Pending(e, n) ← C(n);
```

the pseudocodes.

Begin Event Handler Upon the begin event of element e, for every XQT node n whose label matches the tag of e, we scan the stack looking for an ancestor a that outer-matches $p(n)$, the parent of n. If a is found, e outer-matches n since appending (e, n) to the matching between a and $p(n)$ forms a matching between e and n. To denote that e has to satisfy n's predicates, we then set the Pending(e, n) set as all branch children of n, i.e., the condition set $C(n)$. Recall that we only describe the process for // axes here. If the axis between n and $p(n)$ is /, we only check whether the top stack element outer-matches $p(n)$.

End Event Handler On encountering the end event of e, we first split the set Out(e) into two subsets (line 1 and 2): **NewMatch**(**e**) retains node n from Out(n) if Pending(e, n) is now empty, i.e., all predicates of n have been satisfied. Therefore, e has inner- and outer-matched every node in **NewMatch**(**e**). **Fail**(**e**) contains node n if Pending(e, n) is not empty. Since e has ended, no pending predicate of n could be satisfied by any descendant of e. Therefore, by checking the pending-sets we evaluate the predicates of every element upon its end event. Since no ancestor of e has encountered its end event and therefore satisfy its predicate, according to the

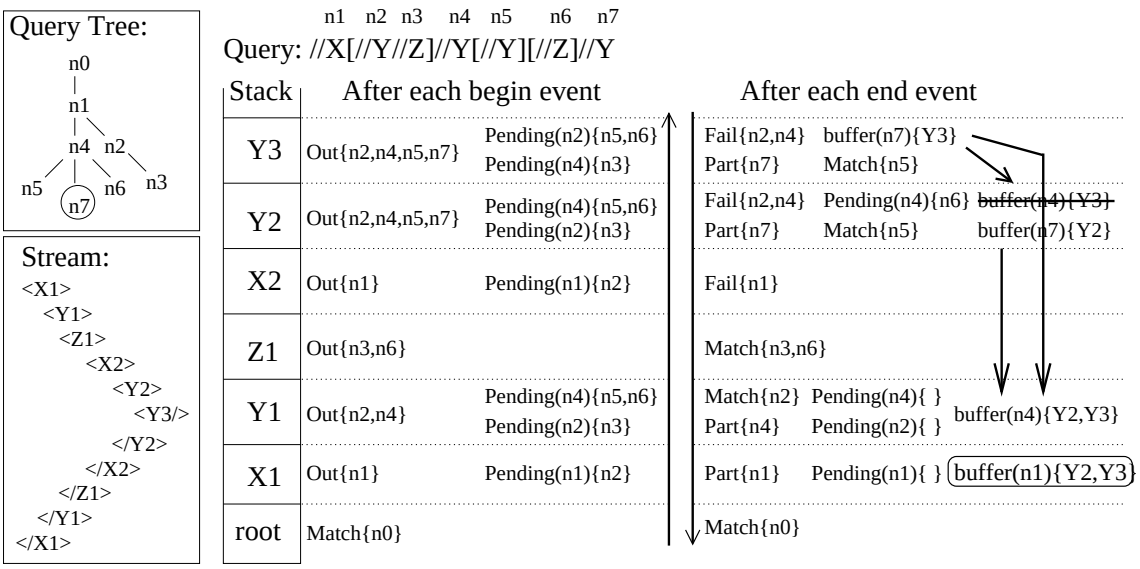


Figure 8.1: Evaluation of a single query

definition, Part(e) contains the nodes from **NewMatch**(**e**) that are trunk nodes, while Match(e) contains the branch nodes.

Note that the pending-sets of e are updated when we process its descendants. When we process e, we also update the pending-sets of its ancestors. For every ancestor a in the stack, if a is pending on some node n that e matches, we remove n from the corresponding pending-set, indicating that a has satisfied this predicate (see line 8 and 9).

The buffer operations maintain the following invariant: *Every buffer item in* **buffer**(**e**, **n**) *has a matching with the output node that contains pair* (e, n) *and, for any pair* (e', n') *after* (e, n), e' *has satisfied the predicates of* n'. This invariant holds when an element is buffered (line 5 and 6) since e both inner- and outer-matches O. Every time this buffer item is appended from a buffer **buffer**($\mathbf{e}', \mathbf{n}'$) to another buffer **buffer**(**e**, **n**) iff n is the parent of n', e is an ancestor of e', and e partial-matches n

(line 11 and 12). Thus the invariant holds.

Recall that the only non-empty Match() set of the stack elements is that of the document root e_0, which initially set to $\{n_0\}$. Therefore, if $p(n) \in \text{Match}(a)$, n must be the child node of n_0 and e must match n following the above invariant. In this case, we emit the buffer items in $\text{buffer}(e, n)$ as result (line 13).

Example 18 ***Figure 8.1*** *illustrates the evaluation process using a highly recursive data snippet and a query with conjunctive subqueries and repeated labels, as described in the left boxes of the figure. We use X_i to distinguish elements with the same tag X. To the right of each stack item, we list all non-empty matching-sets and buffers after processing the element's begin event and end event. The element names are omitted from the matching-sets and buffers. The crossed-out buffer items are cleared at the event. The buffer items in the circle are sent to output at the event. We describe the process of the begin and end event of the element Y_3 as a highlight.*

At the begin event of Y_3, its tag matches XQT node n_2, n_4, n_5, and n_7. Scanning the stack we find the parent node of each of them (n_1, n_1, n_4, and n_4, respectively) is outer-matched by an ancestor of Y_3. Therefore, Y_3 outer-matches all four nodes. We then create the pending-set for the two nodes that have branch children, n_2 and n_4.

At the end event of Y_3, since the pending-sets for n_2 and n_4 are non-empty, they are put into the Fail set. Then, $\text{Part}(Y_3)$ *now contains the trunk nodes left, i.e., n_7, and* $\text{Match}(Y_3)$ *the branch nodes, i.e., n_5. Since the output node n_7 is in* $\text{Part}(Y_3)$*, we put this element into* $\text{buffer}(Y_3, n_7)$.

Algorithm 13: End(Element e)

```
   /* update the sets */;
1  Fail(e) ← Out(e) ∩ {n|Pending(e, n) ≠ φ};
2  NewMatch(e) ← Out(e) − Fail(e);
3  Part(e) ← NewMatch(e) ∩ Trunk;
4  Match(e) ← NewMatch(e) ∩ Branch;
   /* O is the output node of the XQT */;
5  if O ∈ Part(e) then add(buffer(e, O), e);
6  foreach n ∈ Fail(e) do clear(buffer(e, n));
   /* update the ancestors in the stack */;
7  foreach ancestor a do
8      foreach pending-set Pending(a, x) do
9          Pending(a, x) ← Pending(a, x) − Match(e);
10     foreach n ∈ Part(e) do
11         if p(n) ∈ Out(a) then
12             append(buffer(a, p(n)), buffer(e, n));
13         else if p(n) ∈ Match(a) then emit(buffer(e, n));
```

We then update the ancestors. First, we remove $\text{Match}(Y_3)$*, i.e.,* n_5*, from the pending-set of every ancestor,* Y_2 *and* Y_1 *in this example. Next, for* $n_7 \in \text{Match}(Y_3)$*, we append* $\text{buffer}(Y_3, n_7)$ *to the buffer of every ancestor that outer-matches the parent node* n_4*. In this example, it is appended to* $\text{buffer}(Y_2, n_4)$ *and* $\text{buffer}(Y_1, n_4)$*.*

Correctness Given the invariant held by the buffer operations in the end event handler, it is easy to see that only correct result items are emitted by the method.

We then show that every result item is sent to output. If there exists a total-matching $M = ((e_0, n_0), \ldots, (e_k, n_k))$, $e_k = e$ and $n_k = O$, between e and O, by induction we know that after the begin event of e_i, $n_i \in \text{Out}(e_i)$. At the end event of e (i.e. e_k), since O (i.e. n_k) is in $\text{Part}(e)$, e is added to buffer $\text{buffer}(e, O)$. Then, when we update the ancestor e_{k-1}, since $n_{k-1} \in \text{Out}(e_{k-1})$, e is appended to

buffer(e_{k-1}, n_{k-1}) (see line 11 in Listing 13). Using induction we can show that e will be finally appended to the buffer buffer(e_1, n_1) and is sent to output.

Complexities We assume that looking up an item in a set takes unit time given a good hash function. Therefore, removing an item from a set also takes unit time. Computing the union or difference of two sets takes linear time in the size of the sets.

Suppose the tag of an element may match the label of T different query nodes, the stack's depth is H, and there are R elements that outer-matches the output node O. In the worst case, e.g., all node tests are wildcards, T could be the size of the query and R could be the size of the data. However, in most real-life scenarios, a tag may match the label of a small number of (usually one or two) query nodes. Moreover, R is also usually much smaller than the size of the dataset especially for queries of high selectivities.

The begin event handler uses $O(TH)$ time: For every query node n whose label matches the tag of e, we scan the stack for an ancestor that outer-matches $p(n)$. The end event handler uses $O(TR + HT^2)$ time: $O(TR)$ for the clear (line 6) and emit (line 13) buffer operations; $O(HT^2)$ for the loop that updates the ancestors in the stack (lines 7, 8, and 9). The buffers of each stack element uses at most $O(TR)$ space, and therefore the total buffer space is at most $O(THR)$. If we take into account that T can be almost deemed as a constant and H is usually very small for most streams (especially for document-centric datasets), the method is very efficient in itself. We show in next section that the segment-based scheme can be applied to

support multiple queries.

8.3 Segment-based Grouping

We present in this section our query evaluation algorithm for multiple queries that uses segment-based grouping. Our approach creates an index for the segments in all the queries. The index, stored in a *segment table*, records for each segment the queries in which it appears in and how the segments are connected in the queries. In the single query evaluation method, a segment is detected in the stream by connecting the incoming element and an ancestor in the stack. The runtime information, i.e., the matching-sets and the buffer, associated with that ancestor are then updated according to the segment. For multiple query evaluation, we follow the same segment-based evaluation scheme, i.e., detecting segments and updating ancestors. However, the matching-sets and buffers are now grouped by common segments to efficiently support multiple queries.

8.3.1 Compile Time

We assign each query a unique integer query-id and use Q_i to denote the query with query-id i. We compile each query into a XQT and compute the following **feature-sets** that consist of query-ids: The set $\mathbf{N}(\mathbf{X})$ contains query-id i iff label X appears in Q_i; $\mathbf{Trunk}(X)$ contains i iff X is a trunk node in Q_i; $\mathbf{Branch}(X)$ contains i iff X is a branch node in Q_i; and $\mathbf{Output}(X)$ contains i iff X is a output node in Q_i.

We assume each label only appear in a query at most once. If a label appear in a query more than once, we apply a global renaming scheme such that the nth appearances of the label in all queries have the same new label. The renaming scheme is implemented in the XPASS system and discussed in Section 8.3.5.

A **mapping M(XY)** contains query-id i iff Q_i has a node with label X that has a child with label Y, i.e. Q_i contains segment $X//Y$.

We organize these information in a **segment table**, where the key is the label X and the value is a data structure that contains all the feature-sets of X and a subtable that contains the mappings. The key of the subtable is another label Y and the value is the mapping $M(XY)$.

Example 19 *Figure 8.2 illustrates the segment table for four example queries. From the table, we know that* `store` *is a trunk node in* Q_1, Q_3, *and* Q_4. *The segment* `store//name` *is used in* Q_3 *and* Q_4. *Since* `name` *is a trunk node in* Q_3 *and a branch node in* Q_4, *we know the segment is is used as* `store//name` *in* Q_3 *and* `store[//name]` *in* Q_4. *The* `root` *label is a special label for the XQT root,* n_0.

8.3.2 Runtime

The algorithm to evaluate multiple queries follows the same scheme as the one to evaluate single queries. When processing the begin event of an element e, we compute its Out() set and the PENDING sets. Before the end event of e, the PENDING sets and buffers are updated when we process descendants of e. Upon the end event of e, we check the PENDING set of e, compute the remaining feature-sets,

Label	Feature-sets	Mappings	
X		Y	$M(XY)$
root	Trunk: {1,2,3,4} Branch: {1,2,3,4} N: {1,2,3,4}	store	{1,3,4}
		book	{2}
store	Trunk: {1,3,4} N:{1,3,4}	location	{1,4}
		*	{1,4}
		book	{3}
		name	{3,4}
location	Branch: {1,4} N:{1,4}		
*	Trunk: {1,4}	title	{1,4}
title	Trunk: {1,2,4} Branch: {3} N: {1,2,3,4} Output: {1,2,4}		
book	Trunk: {2} N:{2}	price	{2,3}
		title	{2,3}
price	Branch: {1,2,3} N:{1,2,3}		
name	Trunk: {3} Branch: {4} N: {3,4} Output: {3}		

Q_1:`//store[//location]//*[price]//title`

Q_2:`//book[//price]//title`

Q_3:`//store[//book[//price][//title]]//name`

Q_4:`//store[//name][//location]//*//title`

Figure 8.2: Segment Table

Algorithm 14: Begin(Element e)

```
1 S1 ← {X|e.tag matches label X};
2 foreach X ∈ S1 and label Z do
3   PENDING(e, XZ) ← PENDING(e, XZ) ∪ (Map(XZ) − Trunk(Z));
4 foreach ancestor a in the stack do
5   S2 ← {Y|a.tag matches label Y};
6   foreach (Y, X) ∈ S2 × S1 do
7     Out(e, X) ← Out(e, X) ∪ (Map(YX) ∩ Out(a, Y));
```

and operate the buffer accordingly. The revised algorithm is described in Listing 14 and 15. The revisions are summarized below.

Grouping Query Nodes An important revision is to operate on groups of query nodes sharing the same label instead of on single query nodes. Recall that in Section 8.2 we use $\text{Out}(e)$ to denote the set of query nodes that element e outer-matches. We now partition this set on the label and use $\text{Out}(e, X)$ to denote the partition with label X, which contains the query nodes *from all queries* that have label X are outer-matches of e. Other matching-sets are partitioned in the same manner. Moreover, matching-sets now consist of query-ids instead of query nodes. Since we assume each label appears in any query at most once, query-id i in $\text{Out}(e, X)$ unambiguously specifies a query node with label X in Q_i that e outer-matches. Therefore, all runtime actions are performed using set operations and can be implemented with efficient bit-wise operations. For example, line 7 in Listing 14 can be interpreted as: If $Y//X$ is a segment in Q_i (i.e., $i \in \text{Map}(YX)$) and ancestor a outer-matches query node $Q_i.Y$ (i.e., $i \in \text{Out}(a, Y)$), then e outer-matches $Q_i.X$ (i.e., $i \in \text{Out}(e, X)$).

PENDING-Set In the single-query method, a pending-set $\text{Pending}(e, n)$ is cre-

Algorithm 15: End(Element e)

```
1  S1 ← {X|e.tag matches label X};
2  foreach X ∈ S1 do
       /* Update the sets */;
3      Fail(e, X) ← ⋃_Z(PENDING(e, XZ));
4      In(e, X) ← N(X) − Fail(e, X);
5      NewMatch(e, X) ← Out(e, X) ∩ In(e, X);
6      Part(e, X) ← NewMatch(e, X) ∩ Trunk(X);
7      Match(e, X) ← NewMatch(e, X) ∩ Branch(X);
       /* update the buffer*/ ;
8      add(BUFFER(e, X), e, Part(e, X) ∩ Output(X));
   /* Update ancestors */
9  foreach ancestor a do
10     S2 ← {Y|e.tag matches label Y};
11     foreach (Y, X) ∈ S2 × S1 do
12         PENDING(a, YX) ← PENDING(a, YX) − Match(e, X);
13         UpdateBuffers( );
14 Procedure UpdateBuffers()
15 U ← Out(a, Y) ∩ M(YX) ∩ Part(e, X);
16 if U ≠ φ then
17     foreach b ∈ BUFFER(e, X) do
18         add(BUFFER(a, Y), b, U ∩ b.candidate);
19 R ← Match(a, Y) ∩ M(YX) ∩ Part(e, X);
20 if R ≠ φ then
21     foreach b ∈ BUFFER(e, X) do
22         emit(b, R ∩ b.candidate);
```

ated for every query node n that e outer-matches. It records all predicates that e is still pending on. When a descendant of e matches a node in Pending(e, n), we remove that node from it. In the multiple-query version, if the tag of e matches label X, we create a set **PENDING**(e, XZ) that contains query id i iff Q_i has a node X with a pending predicate rooted at a node Z. Initially $i \in$ PENDING(e, XZ) iff Z is a branch child of X in Q_i, i.e., $i \in M(XZ) -$ Trunk(Z) (line 3 in Listing 14).

Buffer Operations For a stack element e, we now create a buffer **BUFFER**(e, X) for each label X that e outer-matches (i.e., $\text{Out}(e, X) \neq \phi$) instead of for every query node. Each buffer item b is associated with a **candidate-set** that contains all query ids for which b is a potential result. The new **add(**BUFFER$, e, s$**)** operation (line 8 and 18 in Listing 15) adds element e to the buffer with candidate-set s. If e is already in the buffer, s is added to the existing candidate set. The **emit(**b, s**)** operation (line 22 in Listing 15) emits the buffer item b as the result of all queries in s.

The buffer operations, in lines 8 and 13, ensure that the following invariant hold for every buffer item b in $\text{Buffer}(a, Y)$: $\forall i \in b.\text{candidate}$*, there exists a matching* M *between* b *and* $Q_i.O$*, which contains pair* $(a, Q_i.Y)$ *and, for every pair* $(e, Q_i.X)$ *that appears between* $(a, Q_i.Y)$ *and* $(b, Q_i.O)$ *in* M*,* e *partial-matches* $Q_i.X$. This invariant is essentially the same as its counterpart in the case of single queries. It ensures that no false result is introduced into the buffer. The invariant holds trivially when a buffer item b is first created (line 8 in Listing 15). Every time b is added from buffer $\text{Buffer}(e, X)$ to $\text{Buffer}(a, Y)$, we know that, for every i in the new candidate set (computed in line 15 and 18): (1) e must partial-matches $Q_i.X$ ($i \in \text{Part}(e, X)$); (2) a outer-matches $Q_i.Y$ and $Q_i.Y$ is the parent node of $Q_i.X$ ($i \in M(YX)$); and (3) a outer-matches $Q_i.Y$ ($i \in \text{Out}(a, Y)$). Therefore, the invariant holds after every buffer operation.

Event	Match	Out	In	Part	Fail	PENDING	BUFFER
<root>	r{1,2,3,4}	r{1,2,3,4}	r{1,2,3,4}				
<store>		s{1,3,4}				sl{1,4} sb{3} sn{4}	
<location>		l{1,4} *{1,4}				 *p{1}	
< /location>	l{1,4}	l{1,4} *{1,4}	 *{4}	 *{4}	 *{1}	 *p{1}	
(store)		s{1,3,4}				**sl{}** sb{3} sn{4}	
<book>		b{2,3} *{1,4}				bp{2,3} bt{3} *p{1}	
<title>		t{1,2,3,4} *{1,4}				 *p{1}	
< /title>	t{3}	t{1,2,3,4} *{1,4}	t{1,2,4} *{4}	t{1,3,4} *{4}	 *{1}	 *p{1}	t{XML^{124}}
(book)		b{2,3} *{1,4}				bp{2,3} **bt{}** *p{1}	b{XML^{2}} *{$XML^{1,4}$}
<price>		p{1,2,3} *{1,4}				 *p{1}	
< /price>	p{12,3}	p{1,2,3} *{1,4}	p{1,2,3} *{4}	 *{4}	 *{1}	 *p{1}	
(book)		b{2,3} *{1,4}				**bp{}** bt{} ***p{}**	**b{XML^{2}}** ***{$XML^{1,4}$}**
<author>		*{1,4}			*p{1}		
< /author>		*{1,4}	*{4}	*{4}	*{1}	*p{1}	
< /book>	b{3}	b{2,3} *{1,4}	b{2,3} *{1,4}	b{2} *{1,4}		bp{}bt{} *p{}	b{$\mathbf{XML}_2$} *{$XML^{1,4}$}
(store)		s{1,3,4}				sl{}**sb{}**sn{4}	**s{$XML^{1,4}$}**
<name>		n{3,4} *{1,4}				 *p{1}	
< /name>	n{4}	n{34} *{1,4}	n{34} *{4}	n{3} *{4}	 *{1}	 *p{1}	n{BN^{3}}
(store)		s{1,3,4}				sl{}sb{}**sn{}**	s{$XML^{1,4}$, BN^{3}}
< /store>		s{1,3,4}	s{1,3,4}	s{1,3,4}		sl{}sb{}sn{}	s{$\mathbf{XML}_{1,4}$, $\mathbf{BN}_{3}$}

Figure 8.3: Evaluation

8.3.3 Complexities

We use T to denote the number of labels that a tag could match, H the depth of the stack, and R the number of elements that may appear in a buffer. These variables are similar to those used for the single query algorithm. We use Q to denote the number of queries and C the number of labels.

We consider each set operation of complexity $O(Q)$ as a unit operation. The begin event uses $O(TC + HT^2)$ time: $O(TC)$ is used to compute the PENDING-sets and $O(HT^2)$ is used to scan the stack and update the Out(e) sets for every segment. The end event uses $O(TC + TR + HT^2R)$: $Q(TC)$ is used to compute the Fail set; $O(TR)$ is used to update the current buffer; and $O(HT^2R)$ is used to update the ancestors. If we consider each set as a storage unit, the algorithm uses $O(C^2)$ memory to store the segment table and $O(THR)$ memory for buffer items. It is easy to see that the dominate factor in both time and space complexities is the number of queries. All the other factors usually do not increase when the number of queries increases.

The revised algorithm does not improve on asymptotic complexities from a naive method that evaluates every single query separately, as no method can: consider a group of queries with no two queries have a common node test. Our main goal is to implement the dominate factor efficiently: to group the query nodes such that the sets can be represented by bitmaps and the set operations can be implemented using bit-wise operations. Our experimental results illustrate that such strategy is useful in practice.

8.3.4 A Running Example

We use the method described above to evaluate the four queries in Figure 8.2 over the XML stream in Figure 5.1. The process is illustrated in Figure 8.3. We omit the process for the second `book` element, which is similar to that of the first `book` element. For each event, we show the sets associated with the corresponding stack element. If the sets of some ancestor are updated during this event, we also show the modified sets for the ancestor, which is shown in italic and parenthesized in the event column.

Notations For each set $S(e, X)$, we show S as the column title, show the first letter of X to the left of the set, and show e in the left most column. For example, Out(location, $* = \{1, 4\}$) is denoted as $*\{1, 4\}$ in the corresponding cell (row 4 column 3).

For every buffer item we show its text content. The candidate set is shown as the superscript. The subscript of a buffer item is the set of query ids that it is a result of. We only show the sets when they are created or updated. We use bold fonts to denote the sets that have been updated during the processing. The buffer items enclosed by a box are results for the queries.

Highlights We describe the process of two representative events here. First, upon the end event of the `title` element, following line 3 to 7 in Listing 15, we know that it matches the `title` query node in Q_3 and it partial-matches the output node in Q_1, Q_2 and Q_4. The element is added to the buffer with candidate set {1,2,4}. We then update the ancestors. Following line 12, we remove Q_3 from

the PEDNING-set of the `book` ancestor that is pending on the `title` predicate. Following lines 15 to 18, we add the buffered `title` element to BUFFER(book, $*$) with candidate set {1,4}, since the `book` ancestor outer-matches the `*` node in Q_1, Q_4 and segment `*//title` is used in Q_1 and Q_4. Similarly, we also add the buffer item to BUFFER(book, book) since the append operation is performed for every label that the ancestor outer-matches.

At the end event of the `book` element, the buffer items in BUFFER(book, book) are sent to output with result set {2} when we check the root element in the stack. Since the result of the computation from line 19 in Listing 15 is a non-empty set {2}, the emit operation on line 22 is performed. In the mean while, the buffer items in BUFFER(book, $*$) are added to the buffer of `store` ancestor by the operation on line 18 when we update the `store` ancestor. The difference here is determined by the segment table: The mapping M(root//$*$) is empty while M(root//book) is not.

8.3.5 Implementation

Queue An element is put in a queue if it outer-matches some output nodes upon its arrival. Instead of making multiple copies of the element, the buffers store pointers to the queue item. We keep a reference count for each queue item. By "adding an element to a buffer", we essentially put a pointer to the queue item into the buffer and increase the reference count. When we emit a buffer item as result of query Q_i, we indeed mark the queue item as result of Q_i. When we remove a buffer item from the buffer, we simply decrease the reference count of the queue item. The

queue item is disposed iff its reference count becomes zero.

Global Renaming During the compilation, we use a *global renaming scheme* to specify a **label id** to every XQT node such that: (1) No two query nodes in the same XQT have the same id; (2)The ith appearances, in arbitrary order, of label X in every XQT is assigned the same id. For example, consider two queries Q_1: `//A[A]//B[A]` and Q_2:`//A[B]//A`. The nodes with label `A` in Q_1 may be ordered from left to right and assigned with identifiers 1, 2, and 4. Then the two nodes with label `A` in Q_2 will have identifiers 1 and 2. Put it another way, we rename the labels such that the two queries become `//A[A']//B[A"]` and `//A[B]//A'`, but all labels `A`, `A'`, and `A"` match the tag `A`.

With the renaming scheme, our method supports queries with multiple occurrences of the same label. We use the pair (query id, label id) to specify a unique node in the whole query set. Recall that our method described earlier groups nodes sharing same label under the assumption that no label appears in a query twice, i.e., the pair (query id, label) uniquely specify a node. Therefore, the same method are implemented in XPASS without that assumption and groups nodes with the same node id.

Compressed Bitmap We implement the methods described in this chapter in the XPASS system using SUN Java SDK 1.5. A bitmap is used to represents a set of query ids. The ith bit is 1 iff query-id i is in the set. Two bitmap implementations are deployed in the system. The first one is the standard BitSet object delivered with the Java SDK. We also implement a compressed Bitmap scheme following the

basic idea of the *word-aligned hybrid code* [68]. The code separates the bitmap into 31-bit chunks. In the compressed bitmap, if the most significant bit of a 32-bit word is 1, the value of the remaining 31-bit is the number of continuous empty chunks this word represents, otherwise the remaining 31-bit is the actual bits from the original bitmap. We present evaluation results using both implementations in Section 8.4.

Memoization We also deploy a technique similar to *memoization*, which has been used in query evaluation in relation databases to store results for expensive subqueries. Consider the running example illustrated in Figure 8.3. If we process the second `book` element in the stream, the value the Out() set and PENDING sets are the same as the first `book` element. Moreover, if the second `book` element has the same types of descendants as the first one, the In(), Part(), and Fail sets will also be the same upon its end event. Therefore, we create a table to record the previous results such that they can be used later.

We define the **prefix** of element e as the tag list in the path from the document root to e, including both ends. If two elements e_1 and e_2 have the same prefix, Out(e_1) and Out(e_2) are always the same. Therefore, when we encounter another element with the same prefix, we can reuse the previous result.

We define a **subtree types** as an integer value: Two elements e_1 and e_2 are assigned the same subtree type iff the subtrees rooted at e_1 and e_2 are *bisimulation* of each other. In other words, if two elements have the same subtree type, their descendants satisfy the same set of predicates. Therefore, upon the end events of the two elements, In(e_1) equals In(e_2). Recall that Part(e) and Match(e) are computed

using Out(e) and In(e) together with the feature-sets. We can infer that, if two elements have the same prefixes and subtree types, their Part() and Match() sets will also be the same upon their end events.

Three hash tables hold the previous results: In the first table, the key is the prefix and the value is the Out() and PENDING sets. In the second table, the key is the subtree type and the value is the In() set. In the third table, the key is the pair (prefix,subtree type) and the value is the Part(e) and Match(e) sets. In the begin and end event handlers, the tables are queried first. The computation is continued only if the key is not found in the table. The result of the computation is then memorized in the table. Since the keys are easy to compute, the overhead of the method is mainly the memory usage. As we show in Section 8.4, the memory overhead is also moderate.

8.4 Performance Evaluation

8.4.1 Setup

All experiments are conducted on a PC-class machine with a 900 MHz Intel Pentium III processor and 1 GB of main memory running the Redhat 7.2 distribution of GNU/Linux (kernel 2.4.9). The memory limit for the Java Virtual Machine (JVM) is set to 512 MB. The XPASS system deploys two bitmap implementations: One is the standard Java BitSet class[3], and the other is our implementation of the compressed bitmap scheme described in Section 8.3.5. We use XPASS-c to denote

[3] `http://java.sun.com/j2se/1.5.0/docs/api/java/util/BitSet.html`

the version that uses the compressed bitmap scheme.

We compare XPaSS with YFilter in our experiments. (XPush was not publicly available at the time we conducted these experiments.) For every query, YFilter can output either matching document ids or every matching elements. We use YFilter-F and YFilter-Q to refer to YFilter operating in these two modes, respectively. To avoid the implications from the warmup process of the JVM, the test queries are evaluated over a dummy XML file before every run for both XPaSS and YFilter.

We measure **processing time** as the total time spent on evaluating all queries on the test data. This time does not include the query compilation time or the time required to output results (which is independent of the query evaluation method). We also measure the **memory usage** as the maximum memory allocated by the systems during the evaluation, including memory used by the JVM. Every data point reported is the mean value over 30 runs. The 5% confidence intervals are usually less than 0.5% of the reported value and thus are omitted from the results.

The parsing time is included in the processing time, which is not the same as the MQPT (*Multiple-Query Processing Time*) reported for the results of YFilter [23]. To report the MQPT, YFilter preparses the documents, evaluates the queries over the events generated from the in-memory parse tree, and reports the processing time. Since YFilter can also evaluate over the events that are generated directly from the parser, we evaluate YFilter using this non-preparsing method, which is also used in XPaSS. YFilter uses MQPT as the main metric since parsing usually is the dominate factor in the filtering scenario. However, it is not the case in the querying setting. As we can see in the results, since we need to check all the queries

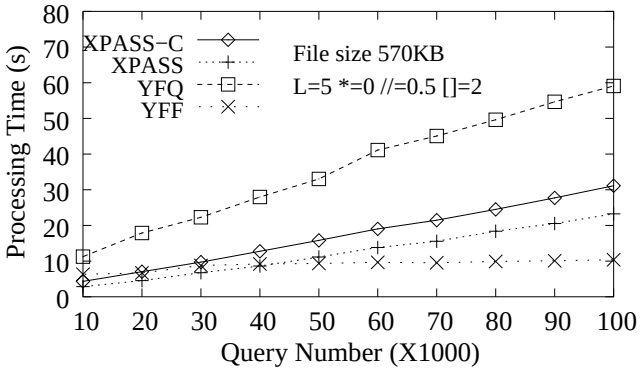


Figure 8.4: Processing Time for Different Number of Queries

that an element may match, the query processing time is usually much larger than the parsing time.

We use the query generator used by YFilter [25] to generate the test queries. In these experiments, we set the possibility of value-based predicates to zero. We vary the following parameters of the query generator and show them in the result figures using these corresponding symbols: **L: query length**, **[]: number of nested paths**, ***: probability of wildcards**, and **//: probability of // axes**. We use the XMark benchmark program and the IBM XML generator program to generate synthetic datasets using `dblp.dtd` and `auction.dtd` from the XMARK project [63]. We also use real data from the `DBLP` Bibliography[4]. The query sets used in the experiments have sizes ranging from 10,000 to 100,000. The XML files used in the experiments have sizes ranging from 100KB to 1MB.

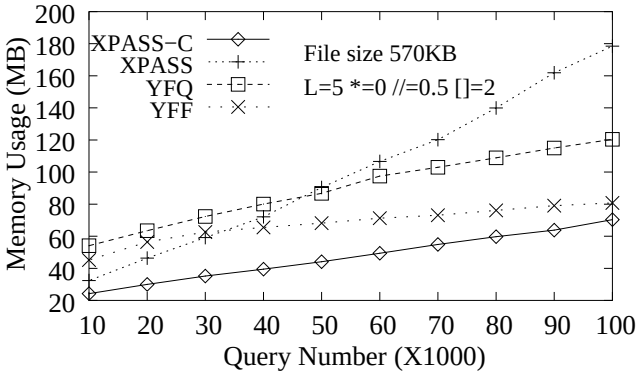


Figure 8.5: Memory Usage for Different Number of Queries

8.4.2 Scalability

Number of Queries In this set of experiments, we evaluate different number of queries, ranging from 10k to 100k and stepping by 10k, over a single document of size 570k generated using XMark benchmark. The processing time and memory usage are reported in **Figure 8.4** and **Figure 8.5** respectively.

In the querying scenario with a large number of subscriptions, book-keeping for every element is a heavy task in terms of both CPU and memory usage. This fact can be witnessed from the performance difference between YFilter-Q and YFilter-F, who share the same path-matching engine. Recall that YFilter-F reports for every document the matching query ids matches, while XPASS and YFilter-Q report for every element the matching query ids. Moreover, YFilter decomposes every nested subquery into a single path. All the path-matching results are processed in a post-processing phase when the whole document has been processed. Before that phase, YFilter-Q needs to maintain the matching paths for every element, while YFilter-F only needs to maintain the matching paths for every document.

[4] `http://www.informatik.uni-trier.de/~ley/db/`

As illustrated in Figure 8.4, although XPASS needs to maintain the matching information for every element, its processing time is very close to that of YFilter-Q. With a fixed document, we may consider its depth (H) as a constant. Moreover, the numbers of labels an element may match (T) and use in its predicates (C) are constant too. Therefore, XPASS performs constant number of set operations for every incoming element. Since the set operations are implemented using efficient bitwise operations, the processing time of XPASS, although proportional to the number of queries, scales better than YFilter-Q as the number of queries increases.

Since bitmap is the main data structure used by XPASS, the size of the bitmaps dominates the memory usage of XPASS. As we can see from Figure 8.5, XPASS that uses non-compressed bitmap doubles its memory consumption when the number of queries doubles. However, when we deploy a simple compressed bitmap scheme, the memory usage is much smaller and the growth rate is similar to that of YFilter-F. Since the bitwise operations for compressed bitmaps are slower than uncompressed bitmaps, we can determine which version to use according to actual application settings, e.g., hardware availability, number of subscriptions, requirements for response latencies, and etc.

Size of Dataset In this set of experiments, we use the XMark benchmark program to generate the datasets using scale factors from 0.001 to 0.01 stepping by 0.001. The result datasets are from 100KB to 1MB, respectively. We run a query set of size 50,000 over these datasets. The processing time and memory usage are reported in **Figure 8.6** and **Figure 8.7**.

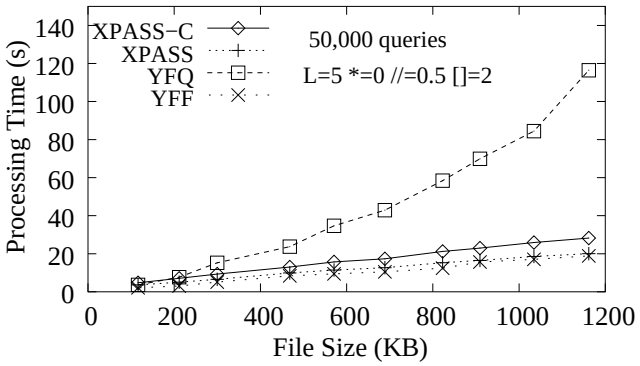


Figure 8.6: Processing Time for 50,000 Queries

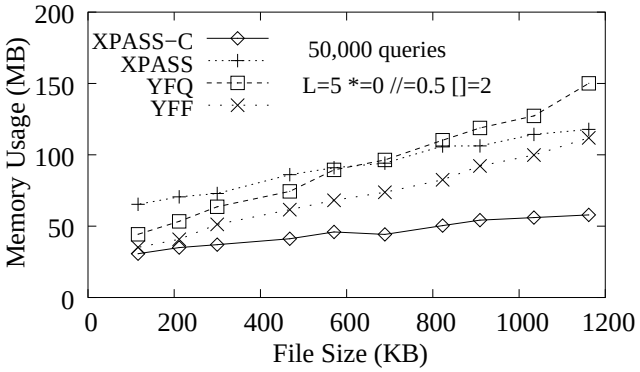


Figure 8.7: Memory Usage for 50,000 Queries

We can see from Figure 8.6 that XPASS scales well when the document size increases. Since the implementation of YFilter-Q is not documented in the report [23], it is not clear why YFilter-Q illustrates quadratic-like behavior. Figure 8.6 also illustrates that both XPASS systems and YFilter-F use linear amount of processing time w.r.t. the document size. For each SAX event, XPASS and YFilter-F perform a constant number of operations when other variables, such as queries and document depth, are fixed. Therefore, the processing time is proportional to the number of SAX events.

Figure 8.7 illustrates that all systems use linear amount of memory w.r.t. the document size. For XPASS and YFilter-Q, when the number of queries is fixed, the memory they use is proportional to the number of elements in the document.

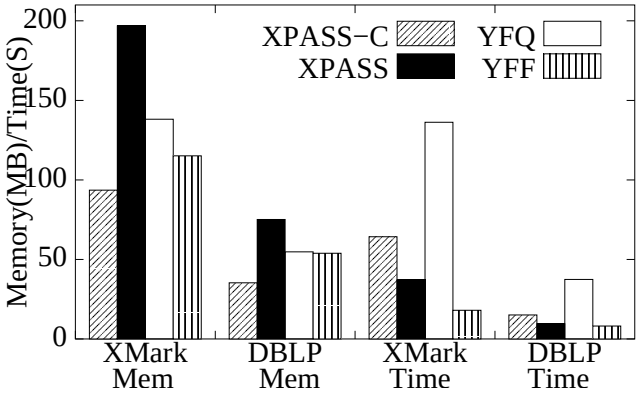


Figure 8.8: Processing Time and Memory Usage for Different Datasets

For YFilter-F, the memory used should be dominated by the size of the automaton. When the document becomes larger, the number of unique prefix may become larger and therefore increases the size of the automaton.

An interesting observation is that in this set of experiments, the memory usages for XPASS-C and XPASS does not depart from each other as they do in Figure 8.5. In Figure 8.5, when the number of queries increases, the bitmaps tend to be more sparse and leads to better compression rate. However, when the file size increases, the load of the bitmap does not change much. Therefore, the compression ratio of the bitmaps is almost the same for a small file and a large file.

8.4.3 Varying Dataset

In this set of experiments, we evaluate 50k queries generated from two different DTDs over a document of size 1MB generated using the respective DTD. The results are illustrated in **Figure 8.8**.

The performances of all systems are significantly different on the two datasets. There are several reasons that the XMark dataset is more expensive to evaluate.

First, the XMark dataset is more deeper than the DBLP dataset. The maximum depth of XMark is 12 and the average depth is 5.6, while the maximum depth of DBLP is 4 and the average is 2.9. Deeper document structure leads to more expensive operations at runtime, such as traversing the stack (for XPaSS) and comparing prefixes (for YFilter). Second, there are 77 element types (i.e., tags) in XMark while there are only 39 in DBLP. More element types lead to larger segment table for XPaSS and larger automaton for YFilter, which consume more memory. More precisely, for XPaSS it is the number of unique label ids that affects the performance directly. There are 318 unique label ids in XMark, while there are only 150 in DBLP.

We also note that, in the XMark dataset, 77 unique tags generate 318 unique label identifiers, since the same tag string may appear in the same query multiple times. For example, in the synthetic work load, the maximum repetition of a tag string in one query is 13, i.e., the same tag string appears in one query for 13 times. Such cases rarely happen in real-life queries. Since the performance of XPaSS is largely affected by the number of unique label ids, we expect that XPaSS should perform better on real-life queries.

8.4.4 Varying Query Features

In this set of experiments, we explore the relation between various query features and the performance of the systems. We evaluate queries with different features over a single document generated by XMark benchmark of size 571KB.

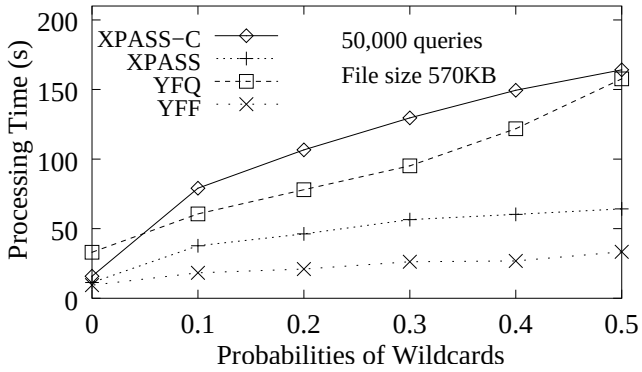


Figure 8.9: Processing Time for Different Number of Wildcards

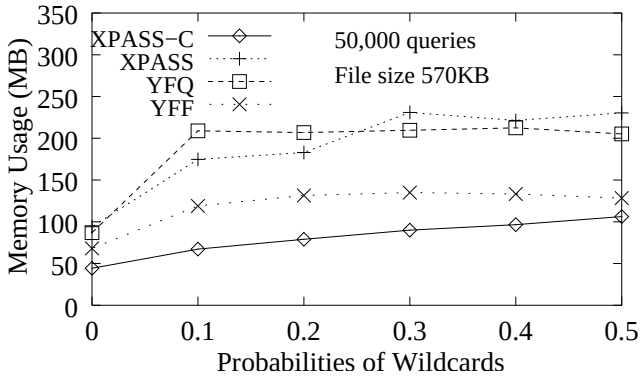


Figure 8.10: Memory Usage for Different Number of wildcards

Query with Wildcards The result depicted in **Figure 8.10** and **Figure 8.9** illustrates how the wildcards affect the performance of the systems. The highest tested probability of wildcards, which roughly equals to number of wildcards divided by the total number of node tests, is 0.5.

More wildcards lead to higher selectivities of the queries. However, although there is a leap from no-wildcard scenario to with-wildcard scenario, increasing number of wildcards seems has little influence on memory usage for XPASS and YFilter systems. Meanwhile, the memory usage of XPASS-c, although the smallest, is almost linear w.r.t the number of wildcards. Note that the memory usage of the systems are affected by the size of the internal data structures: segment-table for

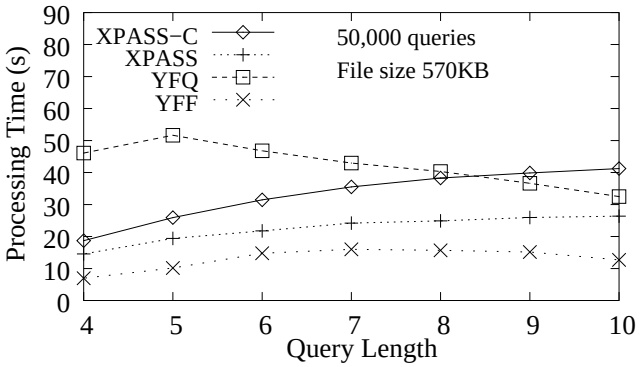


Figure 8.11: Processing Time for Different Length of Queries

XPASS and automaton for YFilter. With no wildcard allowed in the query, there are must less unique segments and prefixes and these systems use much less memory. However, for XPASS-c, higher selectivities of the queries lead to denser bitmaps and smaller compress ratio. Therefore, its memory usage is almost proportional to the number of wildcards. The performance of XPASS-c is also affected by the number of wildcards in a larger degree than XPASS is affected, since denser bitmaps are also more expensive to compute in compressed form.

Wildcards in XPath queries are generally expensive in streaming evaluation since they increase the selectivity of the query and introduce nondeterminism. A query starts with `//*` requires processing every begin event in the stream. However, we do not think this problem is important in real-life queries since they usually have few wildcards. Moreover, even if there exists many queries with many wildcards, we may develop some optimization techniques, e.g., using DTD information to instantiate the wildcards. Some results from the query containment research may also be used. For example, `//*//A` is equivalent to `//A`.

Query of Different Length This set of experiments illustrates how the query

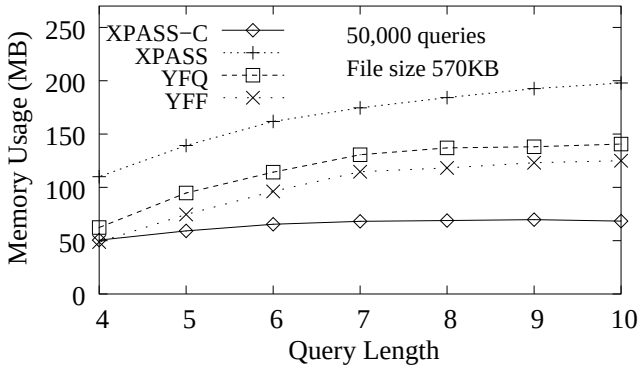


Figure 8.12: Memory Usage for Different Length of Queries

length affects the performance of the systems. The processing time and memory usage are illustrated in Figure 8.11 and 8.12. For XPASS, longer queries imply larger number of segments and denser segment table. For YFilter, longer queries imply larger number of states. Therefore, the memory usages of all systems are proportional to the query length. XPASS-c uses less memory because that, as the number of location step increases, many bitmaps become more sparse (considering that the average depth of the XMark dataset is only 5.6). This effect is also proved by the fact that the compress-ratio of the bitmaps increases when the length of the query increases, i.e. the two curves of XPASS and XPASS-C are departing from each other.

For the same reason, longer queries may actually be processed faster since the result set is smaller, as illustrated in Figure 8.11 for the YFilter systems. The processing time of XPASS systems continues to increase when the query becomes longer. This fact is caused by another overhead of the memoing method we deploy, since the subtree types are computed even if the element does not outer-match any query node. However, real-life queries usually are not very long. Moreover, the

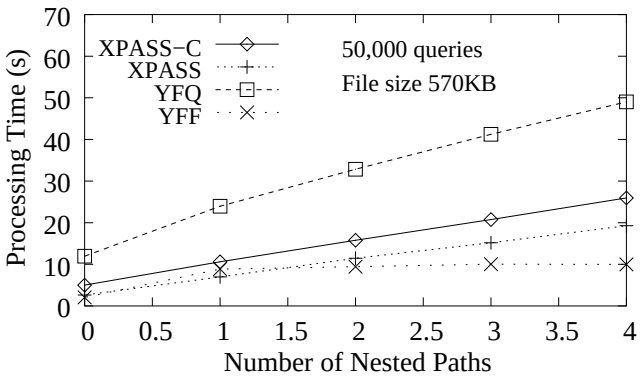


Figure 8.13: Processing Time for Different Number of Nested Paths

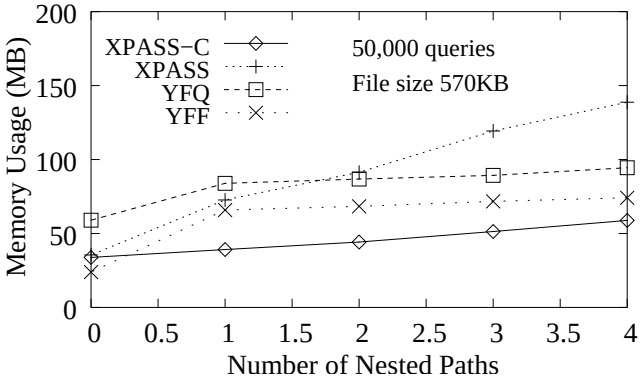


Figure 8.14: Memory Usage for Different Number of Nested Paths

overhead is still moderate, as XPASS is faster than YFilter in most cases.

Different Number of Nested Paths This set of experiments illustrates how the number of nested paths affects the performance of the systems. The processing time and memory usage are illustrated in Figure 8.13 and 8.14.

As reported in [23], the performance of YFilter-F degrades a fair amount when number of nested queries increases from 0 to 1. However, adding extra nested path both increases the size of the NFA and increases the selectivity of the queries. The overall result is that the performance almost stays the same when extra nested paths are added. YFilter-Q illustrates similar pattern in memory usage as YFilter-F. However, since there is no documentation about how the querying component is

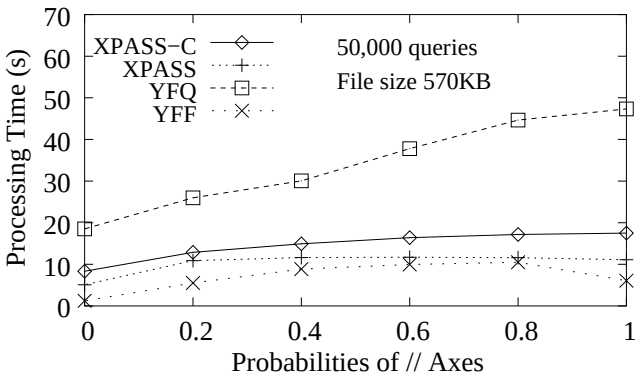


Figure 8.15: Processing Time for Different Number of // Axes

implemented, it is not clear why the processing time of YFilter-Q is proportional the number of nested paths.

For XPASS, higher selectivity implies more sparse bitmap. As we can see from Figure 8.13, the compress-ratio increases along with the number of nested path. However, the processing time still increases moderately because of the overhead that we need to compute the inner-matches regardless of the overall selectivity.

Different Number of // Axes This set of experiments illustrates how the number of // axes affects the performance of the systems. The processing time and memory usage are illustrated in Figure 8.15 and 8.16.

It is easy to see that the systems perform better when all the axes are child axes. For YFilter, when all the axes are the same(/ or //), there are smaller number of common prefixes. Therefore, the size of the NFA is smaller and thus leads to less memory usage. Again, it is not clear why the processing time of YFilter-Q is proportional to the number of // axes.

For XPASS, if there are only / axes in the queries, only the top element in the stack needs to be checked in the event handlers. Moreover, there are less items

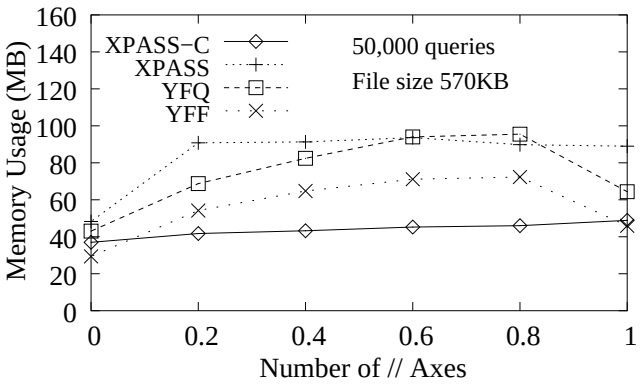


Figure 8.16: Memory Usage for Different Number of //Axes

in the segment table since each element can only form segments with its children. Therefore, XPASS performs better in this case. However, if the queries contain // axes, the number of // axes does not make much difference in the performance.

8.5 Conclusion

This chapter studies a publish-subscribe environment with two twists. First, the data consists of an XML stream that does not necessarily admit a natural segmentation into documents or other units. Second, subscriptions are expressed as queries on this stream instead of filters on documents. As a result, subscriptions may be tailored more accurately to the needs of a user or agent.

We highlighted the challenges for efficient subscription-server implementations in such environments, and described the methods used by our XPASS system. These methods are based on grouping operations by XPath query segments and efficient bitmap-based operations. Our experimental results indicate that XPASS can process at least 10^5 queries while maintaining a good throughput and, more important, a modest memory footprint. In continuing work, we are exploring extending XPASS

to a larger subset of XPath, conducting further experimental studies with a larger numbers of queries, and building a user interface that facilitates the creation of subscriptions.

Chapter 9

Future Work

In this chapter we propose two possible extensions of our current work. The first direction is to introduce a schema-aware runtime optimization technique, which can use optional schema information of the data to further optimize the query engine. Such optimization should be dynamic since there is no guarantee that the schema information is available, or there may be different schema for different data sources that may come in at the same time. We also investigate the possibility to extend the system to support more features of XPath and more complex query languages such as XQuery. Since XQuery is the de facto query language of XML and used in many applications, supporting XQuery will further improve the usability of the system and provide more insights in more general XML streaming processing problems. We describe these possible extensions in more detail below.

9.1 Schema-based Runtime Optimization

The current XSQ system does not take into account the schema information of the data. If the data source provides the schema of the data, either as DTD or as XML Schema, the system should be able to use those information to guide the evaluation to achieve better performance. The following example illustrates some

potential optimizations given the schema information.

[Example: Schema-based Optimization]

Suppose we want to evaluate the query: `//pub[year=2002]//book`. There are several possible optimizations that can be applied if we have schema information.

- If we know from the schema that the `pub` element does not have a `year` child or a `book` descendant, we can simply ignore the whole document.

- If we know that the `pub` element has only one year child, and we have encountered the first `year` child of the element whose value is not 2002, we can simply ignore the whole `pub` element since it is impossible for this element match the query. Without schema information, we need to buffer all the the `book` elements inside the `pub` element, and remove them at the end of the `pub` element.

- If we know that the `book` element can be only the *child* of the `pub` element, we can ignore any other child elements in the `pub` element. If we do not have the schema information, we have to process all descendants of the `pub` element since it is always possible that there is a `book` element inside them, which is the descendant of this `pub` element and therefore matches the query.

- If we know from the schema that the `pub` element is the child of the document root and the `book` element is the child of the `pub` element, there is no need to use the complex mechanisms, such as depth vectors and hierarchical index, which are used to support multiple matchings between the query and the

elements. Essentially we can use a DFA (deterministic finite state automaton) to evaluate such queries with small modifications.

We need to address several interesting issues when we apply the dynamic schema-based optimization. The optimization has to be dynamic and optional since we want the system is schema-aware while still be able to process streaming data from heterogeneous data sources that may not provide schema information. Another problem arises when we have several data sources of which only some provide schema information while others do not. If we want to evaluate a query on streams from these datasets simultaneously, we have to assign different operations for data items from different sources since the optimization should be different for different schema. Moreover, it is difficult to obtain the optimization rules if the schema and the query are complex. Even if we have all the rules from the schema and the query, it is still an interesting problem that how we check the rules efficiently at runtime.

9.2 Streaming XQuery Evaluation

We also investigate to extend our current work to streaming XQuery evaluation. Since XQuery uses XPath to address parts of the document, we can use the result from XSQ as the input of upper layer XQuery engine that can construct the results in a streaming manner. Since XQuery is a turning-complete language, it is an interesting research problem to support streaming XQuery evaluation. The following example shows an XQuery query that returns all the books that is published in year 2003 and is more expensive than the average price. The results are enclosed

by a user defined new tag "ExpensiveBooks".

[Example:XQuery Query]

```
let $y :=  2003
let $a :=  doc( "book.xml" )//pub
let $b :=  $a[year=$y]//book
let $c :=  price
return
      <ExpensiveBooks>
         <year>$y</year>
         $b[$c > average($a//$c$)]
      </ExpensiveBooks>
```

Evaluating XQuery queries over streaming data is a challenge task since it is a superset of XPath (whose features are not all supported in the current streaming systems) and provides a whole set of query mechanisms such as joins and aggregations. We list some of the interesting problems below.

- XQuery allows joins, which means we need to have several XPath engines working in parallel and we also need some new mechanisms to join their results dynamically. Since XSQ has the stream-in-stream-out feature, it is suitable to be used as a sub-operator for a stream-join operator. The current streaming systems only support one-side streaming for the join operation. It will be

an interesting problem to allow both data source of the join operator to be streaming.

- Blocking operations in XQuery, such as grouping and sorting, are difficult to evaluate in streaming data. Although it is impossible to get the accurate result of these blocking operations for streaming data, we can try to get some approximation of the result, or get the up-to-date result. It is also interesting to investigate the different result set under different streaming semantics such as window-based semantics and internal-based. (Our current system, which always return the exact result, uses the event-based semantics.)

- For XQuery queries that use variable bindings, i.e., assign the result of an XPath expression to a variable as shown in the above example, we may combine all the expressions used in an XQuery query and processed them in one single engine. Since these expression are likely to be different, we need to devise a new mechanism that transform multiple query into one automaton that has multiple output channels. If two expressions are actually connected by variables, we need to detect this fact to use it in the transformation. For example, if variable $\$c$ is bound to expression `price` and it is connected to both $\$b$ and $\$a$ as shown in the above example, we need to mechanism to share the common subqueries inside one query.

Chapter 10

Conclusion

As a W3C standard, XPath is arguably a cornerstone of the XML world since it is widely used in almost every XML application to retrieve the data matching with specified pattern. Evaluating XPath queries over streaming XML data is an important and challenge task. In a streaming environment, traditional approach is inapplicable since we cannot visit the data arbitrarily but in a single sequential pass. Since potential result items may come before the data that are required to determine their membership in the final result set, we have to buffer those items *together with the partial results for each of them*. Moreover, it is also challenging to match the complex pattern, with boolean operators connecting sub-patterns and with both existential and universal quantification semantics, in only once pass of the data, which is required for streaming evaluation.

We present in this thesis the XSQ system, which addresses the problem of streaming evaluation of XPath queries. XSQ is the first system that handles complex XPath features such as closure axes, multiple predicates, aggregations, subqueries, and reverse axes. The segment-based evaluation method has been proved to be very efficient and can be extended to support multiple queries. Besides being the first system that handles these features in XPath in streaming evaluation, XSQ is also released under GNU/GPL license as a tool for researchers and XPath users.

We also did a very thorough performance study to compare the performance of XSQ with other available XPath processors (such as XMLTK, Saxon, Xalan, XQEngine, Joost, Galax, and XPATH-TF), for different synthetic (XMark, XML-generator, and Toxgene) and real-life datasets (Shakespeare's plays; the NASA ADC XML dataset, bibliographic records from the DBLP site (DBLP), and the PIR-International Protein Sequence Database (PSD)), for different features of the datasets (shallow, recursive, document-oriented, data-oriented), and for different features of the queries (number of location steps, number of closure axes, number of reverse axes, different query structures).

The performance study shows that XSQ has very high throughput and use small amount of memory. The normalized throughput of XSQ, which is the raw throughput normalized by the throughput of a pure parser, is almost twice as much as the popular and fastest XPath processors such as Saxon and Xalan (and even more faster than other systems such as XQEngine, Galax, Joost, and XPATH-TH). XSQ achieves almost the same normalized throughput of XMLTK while handles much more complex queries, even if XMLTK supports only very simple XPath queries that do not need buffering in streaming evaluation.

We also present in this thesis the XPASS system, which is the first XPath-based publisher/subscriber system that supports querying instead of filtering. Instead of always return the whole document as the result, it returns the results of each query to the users. Extended from the segment-based evaluation method, the XPASS system is very efficient in both throughput and memory usage. We compare our XPASS system with YFilter, which is a leading pub/sub system and the only one

publicly available for testing. Our performance study shows that the throughput of XPaSS is comparable to that of the YFilter system, although YFilter always returns the whole document as the result. Meanwhile, using the compressed-bitmap optimization, XPaSS also use less memory than YFilter despite the fact that YFilter does not need to keep track of potential results for every single query.

BIBLIOGRAPHY

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

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

[3] 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.

[4] 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/`.

[5] R. Avnur and J. M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 261–272, Dallas, Texas, May 2000.

[6] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a Template-based Data Generator for XML. In *The 5th International Workshop on the Web and Databases*, pages 49–54, Madison, Wisconsin, June 2002.

[7] 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 (ICDE)*, pages 455–466, Bangalore, India, March 2003.

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

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

[10] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. Xquery 1.0: An xml query language 1.0. W3c working draft, W3C, `http://www.w3.org/TR/xquery/`, 2003.

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

[12] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D. Winer. Simple object access protocol (SOAP) 1.1. `http://www.w3.org/TR/SOAP/`, May 2000.

[13] P. Buneman, S. Davidson, G. Hillebrand, and D. Suciu. A query language and optimization techniques for unstructured data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 505–516, Montréal, Québec, June 1996.

[14] 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.

[15] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 235–244, San Jose, California, February 2002.

[16] S. Chaudhuri, G. Das, and V. Narasayya. A Robust, Optimization-based Approach for Approximate Answering of Aggregate Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 295–306, Santa Barbara, California, May 2001.

[17] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 379–390, Dallas, Texas, May 2000.

[18] Z. Chen, H. V. Jagadish, F. Korn, N. Koudas, S. Muthukrishnan, R. T. Ng, and D. Srivastava. Counting Twig Matches in a Tree. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 595–604, Heidelberg, Germany, April 2001.

[19] 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.

[20] B. Choi. What Are Real DTDs Like. In *The 5th International Workshop on the Web and Databases*, pages 43–48, Madison,Wisconsin, June 2002.

[21] J. Clark and S. DeRose. XML path language (XPath) version 1.0. W3C Recommendation `http://www.w3.org/`, Nov. 1999.

[22] A. Deutsch, M. F. Fernández, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. `http://www.w3.org/xml/`, Aug. 1998.

[23] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Computer Systems (TOCS)*, December 2003.

[24] Y. Diao, D. Florescu, D. Kossmann, M. J. Carey, and M. J. Franklin. Implementing memoization in a streaming XQuery processor. In *Proceedings of the 2nd International XML Database Symposium*, Toronto, Canada, August 2004.

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

[26] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 61–72, Madison, Wisconsin, June 2002.

[27] M. F. Fernández, D. Florescu, J. Kang, A. Y. Levy, and D. Suciu. STRUDEL: A Web-site management system. In J. Peckham, editor, *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 549–552, Tucson, Arizona, May 1997.

[28] M. F. Fernández and J. Siméon. Galax, 2002. `http://db.bell-labs.com/galax/`.

[29] 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.

[30] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates Over Continual Data Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 13–24, Santa Barbara, California, May 2001.

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

[32] 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.

[33] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with Deterministic Automata. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 173–189, Siena, Italy, January 2003.

[34] S. Guha and N. Koudas. Approximating a Data Stream for Querying and Estimation: Algorithms and Performance Evaluation. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 567–576, San Jose, California, February 2002.

[35] 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.

[36] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 287–298, Philadelphia, Pennsylvania, June 1999.

[37] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 171–182, Tucson, Arizona, May 1997.

[38] C. M. Hoffmann and M. J. O'Donnell. Pattern Matching in Trees. *Journal of the ACM (JACM)*, 29(1):68–95, 1982.

[39] A. L. Hors, P. L. Hgaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model Level 2 Core Spec-

ification. W3C Recommendation, W3C, http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113, November 2000.

[40] IPEDO. http://www.ipedo.com, 2002.

[41] J. X. Josephine M. Cheng. XML and DB2. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 569–573, San Diego, California, February 2000.

[42] V. Josifovski and M. Fontoura. Querying xml streams. *The VLDB Journal*, 2004.

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

[44] M. Kay. XSL Transformations (XSLT) Version 2.0. W3C Working Draft, W3C, http://www.w3.org/TR/xslt20/, November 2003.

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

[46] P. Kilpel. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, Dept. of Computer Science, University of Helsink, 1992.

[47] L. V. Lakshmanan and P. Sailaja. On Efficient Matching of Streaming XML Documents and Queries. In *The 8th International Conference on Extending Database Technology*, pages 142–160, Prague, Czech Republic, March 2002.

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

[49] L. Liu, C. Pu, and W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *Knowledge and Data Engineering*, 11(4):610–628, 1999.

[50] L. Liu, C. Pu, and W. Tang. WebCQ - Detecting and Delivering Information Changes on the Web. In *9th International Conference on Information and Knowledge Management*, pages 512–519, McLean, Virginia, November 2000.

[51] B. Ludascher, P. Mukhopadhayn, and Y. Papakonstantinou. A Transducer-Based XML Query Processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 227–238, Hong Kong, China, August 2002.

[52] 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 (ICDE)*, pages 555–566, San Jose, California, February 2002.

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

[54] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 65–76, Madison,Wisconsin, June 2002.

[55] D. Olteanu, T. Kiesling, and F. Bry. An Evaluation of Regular Path Expressions with Qualifiers against XML Streams. Technical Report PMS-FB-2002-12,

Institute for Computer Science, Ludwig-Maximilians University, Munich, May 2002.

[56] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking Forward. In *Workshop on XML-Based Data Management (XMLDM) at the 8th Conference on Extending Database Technology*, pages 109–127, Prague, Mar. 2002. Springer-Verlag.

[57] ORACLE. ORACLE XML DB, January 2003. `http://otn.oracle.com/tech/xml/xmldb/pdf/XMLDB_Technical_Whitepaper.pdf`.

[58] 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.

[59] F. Peng and S. S. Chawathe. The XSQ project. `http://www.cs.umd.edu/projects/xsq/`, 2003.

[60] B. Plale and K. Schwan. dQUOB: Managing Large Data Flows by Dynamic Embedded Queries. In *The 9th IEEE International Symposium on High Performance Distributed Computing*, pages 263–270, Pittsburgh, Pennsylvania, August 2000.

[61] B. Plale and K. Schwan. Dynamic Querying of Streaming Data with the dQUOB System. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):422–432, April 2003.

[62] Really simple syndication. `http://blogs.law.harvard.edu/tech/rss`, 2003. Version 2.0.

[63] 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.

[64] L. Segoufin and V. Vianu. Validating Streaming XML documents. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 53–64, Madison, Wisconsin, June 2002.

[65] P. A. Tucker, D. Maier, and T. Sheard. Applying Punctuation Schemes to Queries Over Continuous Data Streams. *Bulletin of the Technical Committe on Data Engineering, IEEE Computer Society*, 26(1):33–40, March 2003.

[66] J. Widom. The Starburst Active Database Rule System. *IEEE Transactions of Knowledge and Data Engineering*, 8(4):583–595, August 1996.

[67] 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.

[68] K. Wu, E. Otoo, and A. Shoshani. Compressing bitmap indexes for faster search operations. In *SSDBM*, Edinburgh, Scotland, July 2002.

[69] Y. Wu, J. M. Patel, and H. V. Jagadish. Estimating Answer Sizes for XML Queries. In *The 8th International Conference on Extending Database Technology*, pages 590–608, Prague, Czech Republic, March 2002.

[70] X-HIVE. `http://www.x-hive.com`, 2002.

[71] F. Yergeau, T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 (third edition). World Wide Web Consortium Recommendation. `http://www.w3.org/TR/REC-xml`, Feb. 2004.

[72] S. Zdonik, M. Stonebraker, M. Cherniack, U. Cetintemel, M. Balazinska, and H. Balakrishnan. The Aurora and Medusa Projects. *Bulletin of the Technical Committe on Data Engineering, IEEE Computer Society*, 26(1):3–10, March 2003.