Evaluating XPath Queries on XML Data Streams

Stefan Böttcher and Rita Steinmetz

University of Paderborn (Germany)
Computer Science
Fürstenallee 11 D-33102 Paderborn
stb@uni-paderborn.de, rst@uni-paderborn.de

Abstract. Whenever queries have to be evaluated on XML data streams - or when the memory that is available to evaluate the XML data is relatively small compared to the document - DOM based approaches that have to load and store large parts of the document in main memory will fail. In comparison, we present an approach to evaluate XPath queries on SAX streams that supports all axes of core XPath, including the sibling axes. Starting from the XPath query, our approach generates a stack of automata that uses the SAX stream as input and generates the result of the query as an output SAX stream. An evaluation of our implementation shows that in general our approach needs less main memory, but at the same time is faster than both, Saxon and YFilter.

1 Introduction

1.1 Motivation and Paper Organization

XML is becoming the de facto standard for information exchange and, as the amount of XML data is steadily growing, a key challenge is to process XML documents fast within the available main memory.

Our contribution focuses on scenarios, in which a system has to evaluate queries fast on documents that are multiple times larger than the main memory available to the system. One typical scenario is an XML news stream provided by a news agency using one of the typical XML formats NewsML [16] or NITF [17] to broadcast their news, and users who want to receive only parts of the news based on queries that represent their interests. Another typical scenario is that devices with a small amount of main memory (as e.g., mobile phones) shall work on large XML documents.

Whenever a scenario requires that the main memory available to evaluate queries on XML data is relatively small compared to the XML data size, approaches that are based on DOM will fail. These approaches have to load the complete XML document as a DOM tree into main memory, and as they need at least 4 pointers for each XML element (name, parent, first child, and next sibling) they yield a memory consumption that covers multiple times the size of the XML data.

Therefore, we propose a SAX based approach to the evaluation of XPath queries. Each input query is translated into an automaton that consists of only four different types of transitions, the treatment of which is described in Section 2. The small size of

R. Cooper and J. Kennedy (Eds.): BNCOD 2007, LNCS 4587, pp. 101–113, 2007.

the generated automata allows for a fast evaluation of the input XML data stream within a small amount of memory.

This paper is organized as follows: The remainder of the first section outlines the query language, summarizes the underlying assumptions, and outlines the problem definition. Section 2 summarizes the fundamental concepts used to describe our approach to evaluate XPath queries. The third section outlines some of the experiments that show the space efficiency and time efficiency of our prototype. Section 4 gives an overview on related work and is followed by the Summary and Conclusions.

1.2 Query Language

The subset of XPath expressions supported by our approach conforms to the set of *core XPath* as defined in [11]. This set is defined by the following EBNF grammar:

```
cxp         ::= `/' locationpath
locationpath      ::= locationstep ('/' locationstep)*
locationstep      ::= x `::' t | x `::' t `[' pred `]'
pred        ::= pred `and' pred | pred `or' pred
                | `not' `(' pred `)' | locationpath
                | locationpath ‘=’ const |`(' pred `)'
```

“cxp” is the start production, “x” represents an axis (attribute, self, child, parent, descendant-or-self, descendant, ancestor-or-self, ancestor, following, preceding, following-sibling, preceding-sibling), “const” represents a constant, and “t” represents a “node test" (either an XML node name test or “*”, meaning “any node name”).

Note that our system supports the sibling axes, whereas other approaches like XMLTK[1], χαοζ[4], AFilter[6], YFilter[9], XScan[14], SPEX[18], and XSQ[20] are limited to the parent-child and ancestor-descendant axes.

1.3 General Assumptions and Problem Definition

As our system is designed to efficiently evaluate XPath queries on a possibly infinite XML data stream, one requirement that our system has to meet is that each SAX event can be read only once, i.e., the stream has to be parsed in a single pass in document order. As we cannot jump backwards within the data stream, we have to rewrite user queries that use backward axes (i.e., ancestor-or-self, ancestor, preceding-sibling, and preceding) into equivalent queries containing only forward axes as described in [19]. The rewriting might lead to equivalent rewritten queries that are exponentially longer than the original queries, but as usually queries are rather short compared to the XML data, the growth of query length will usually not extend the runtime too badly.

Problem description: After rewriting queries, the remaining problem examined in this paper is the following. The input consists of a core XPath query containing only the forward axes and of an XML data stream in form of a SAX input event stream. The desired output is a SAX event stream of query results in document order. The main requirements of our system are to use as little main memory as possible in order to reach data throughput rates comparable to those of data streams.