XML Prefiltering as a String Matching Problem

Christoph Koch, Stefanie Scherzinger, Michael Schmidt

Saarland University Database Group
Saarbrücken, Germany
{koch,scherzinger}@infosys.uni-sb.de, mschmidt@informatik.uni-freiburg.de

***Abstract*— We propose a new technique for the efficient search and navigation in XML documents and streams. This technique takes string matching algorithms designed for efficient keyword search in flat strings into the second dimension, to navigate in tree structured data. We consider the important XML data management task of prefiltering XML documents (also called XML projection) as an application for our approach. Different from existing prefiltering schemes, we usually process only fractions of the input and get by with very economical consumption of both main memory and processing time. Our experiments reveal that, already on low-complexity problems such as XPath filtering, in-memory query engines can experience speed-ups by two orders of magnitude.**

I. Introduction

In XML stream processing, or XML query evaluation using main memory-based query processors, we often process XML data ad-hoc, without loading it into a database or building an in-memory tree representation. Doing this efficiently has been recognized as an important data management problem [1]–[7]. In XML data management, we face similar problems as in string matching, as we often need to detect patterns (such as a specific tagname) within XML input streams. However, the state of the art in string matching to date has found little application in the acceleration of XML processing.

String matching algorithms have been subject to extensive study for more than thirty years [8]–[13]. In today's algorithms, the input is not processed one character at-a-time. Rather, string matching algorithms like Boyer-Moore [11] and Commentz-Walter [13] rely on the insight that matching keywords from right to left lets us *skip* parts of the input. For instance, the keyword "ICDE" has four characters. Suppose the fourth character in the input string is the letter "A". Then the keyword cannot be matched by the first four characters, and we can safely skip to the 8th character. If this character is the letter "C", then the pattern could be matched. Hence, we shift to the right for the substring "DE", and try to match the keyword from right to left.

This paper makes a case for leveraging ideas from string matching for XML stream processing. We take the leap from processing flat strings to structured documents, and present a new technique for the efficient search and navigation in XML documents and streams. What makes our approach very attractive is that it shares the advantages of established string matching algorithms: Using statically precompiled lookup tables of fixed size, the runtime algorithm comes at little expense for CPU and main memory resources. Moreover, it can be implemented as a streaming algorithm, where we scan the input with a fixed-size window in a single pass. Within the window held in main memory, we can locally jump back and forth, all the while trying to quickly process the input by skipping characters. As we confirm in our experiments, this results in significant speedups for searching flat strings and XML data alike. In particular, both the runtime costs and the number of character comparisons of our technique are comparable with Boyer-Moore style string matching algorithms.

While the runtime algorithm is simple, lean, and efficient, the static analysis computing the runtime data structures is not trivial. In moving from flat strings to structured data, new challenges arise. When the complete XML input document has been tokenized into opening- and closing tags, e.g. using a SAX parser, it is straightforward to track ancestor-descendant relationships between nodes in the document tree. However, when we skip data, we disregard parts of the document structure. For instance, assume we search for an occurrence of the keyword "$\langle a \rangle$", and once we have found it, we locate keyword "$\langle b \rangle$" in the input. Ad hoc, the relationship between these nodes is unclear.

To deal with this problem, we make use of schema information. DTDs provide us with the set of possible tagnames, parent-child and ancestor-descendant information, as well as order and cardinality constraints. We also take required attributes into account to compute XML-specific offsets, which let us skip parts of the input in addition to the skips performed by string matching algorithms. Based on a holistic static analysis, we decompose the task of navigating inside XML documents into multiple string matching problems, which are solved individually using established algorithms. This decomposition is robust, and computes very basic lookup-tables, to be used at runtime.

By only inspecting a fraction of the characters in the input, we are able to build highly scalable applications for XML stream processing. These applications exhibit high throughput and an economical use of resources. As a proof-of-concept, we apply our technique to *XML prefiltering*, an established XML data management technique, which has sprung from the following motivation.

Over the past years, a variety of applications for XML processing have been developed, such as XQuery, XSLT and XPath processors, or less expressive filters for publish-subscribe scenarios [1], [2]. Systems that are designed as main memory engines often fail to handle large amounts of XML data [5], [14]–[16]. While some systems push the limits of the available memory and computational resources to cope

```
<!DOCTYPE site [
  <!ELEMENT site (regions)>
  <!ELEMENT regions   (africa, asia, australia)>
  <!ELEMENT africa    (item*)>
  <!ELEMENT asia      (item*)>
  <!ELEMENT australia (item*)>
  <!ELEMENT item      (location,name,payment,
              description,shipping,incategory+)>
  <!ELEMENT incategory EMPTY>
  <!ATTLIST incategory category ID #REQUIRED>
  ... ]>
```

Fig. 1. Excerpt from the XMark DTD

with the size of the input [1]–[4], [6], [7], main memory nevertheless remains the limiting resource. XML prefiltering techniques, also called "XML projection" or "pruning" [5]–[7], tackle this problem. In prefiltering, only relevant data is passed on to the XML query engine, while irrelevant data is discarded. In many practical cases, this considerably reduces the amount of data stored in main memory [5], [6].

In the example below, we show how string matching algorithms can be leveraged to accelerate XML prefiltering.

Example 1: We discuss XML prefiltering for XQuery

```
<q>{ //australia//description }</q>
```

against the document from Figure 2. We consider the simplified XMark DTD [17] from Figure 1, and assume that all unlisted tags have `#PCDATA` content.

In XML prefiltering for this query, we are only interested in tag node `australia` and all its `description` descendants. By default, we preserve the top-level node to guarantee well-formed XML output. Hence, in prefiltering the document from Figure 2, we obtain the document below, on which query evaluation yields the same result.

```
<site><australia><description>Palm Zire 71
</description></australia></site>
```

In localizing a tag in the input we must keep in mind that tags may contain whitespaces or attributes. However, all tags for element t share the prefix "$\langle t$" or "$\langle /t$". While "$\langle \quad t\rangle$" is not allowed by the XML standard, "$\langle t \quad \rangle$" is valid syntax. Thus, we search for the keyword "$\langle t$" using string matching algorithms, and then locally seek the trailing "$\rangle$" or "$/\rangle$". In Figure 2, we use $\uparrow$ to mark characters that are checked from left to right, and $\downarrow$ for characters that are checked locally from right to left. Symbol $\updownarrow$ represents both $\uparrow$ and $\downarrow$.

At position **1**, we start by scanning for keyword "$\langle$*site*" using the Boyer-Moore algorithm. The 5th character ("e") is investigated, and the match is verified from right to left. Character ">" on the right asserts that an opening tag has been detected. Next we are interested in tag $\langle$*australia*$\rangle$. According to the DTD, "$\langle$*regions*$\rangle\langle$*africa/*$\rangle\langle$*asia/*$\rangle$" with length 25 is the minimum string preceding this tag. We skip 25 characters, and search for the keyword "$\langle$*australia*" with length 10. Thus, search is resumed 25 + 10 characters to the right, at position **2**. Up to position **3**, we check every 10th character, and observe that it is not contained in the keyword. Character "*l*" at position **3** is contained, and we shift $|\langle australia| - |\langle austral| = 2$ characters to the right. Reading character "*t*", we rule out a match, and the search continues. At position **4**, we finally match and output $\langle$*australia*$\rangle$.

Next, we search for the keywords "$\langle$*description*" and "$\langle$*/australia*" using the Commentz-Walter algorithm. Scanning for "$\langle$*/australia*" is necessary because the DTD does not assert the existence of tag $\langle$*description*$\rangle$ as a descendant of `australia`. Position **5** shows a suspected match for keyword "$\langle$*description*", which is aborted. At position **6**, we match $\langle$*description*$\rangle$. We record its start position and search for "$\langle$*/description*". We jump the size of $|\langle /description|$ characters to the right, match character "$\langle$", and verify a match for $\langle$*/description*$\rangle$ at position **7**. The data is copied to the output, starting from the recorded start position of $\langle$*description*$\rangle$ up to (and including) the closing tag. We resume the search for "$\langle$*/australia*" and "$\langle$*description*", and perform an initial jump for the string "$\langle$*shipping/*$\rangle\langle$*incategory category*='' /$\rangle\langle$*/item*$\rangle$". As the minimum distance to the next occurrence of tag $\langle$*description*$\rangle$ is greater, this jump offset is safe. Finally, we match and output $\langle$*/australia*$\rangle$ and $\langle$*/site*$\rangle$ at positions **8** and **9**.

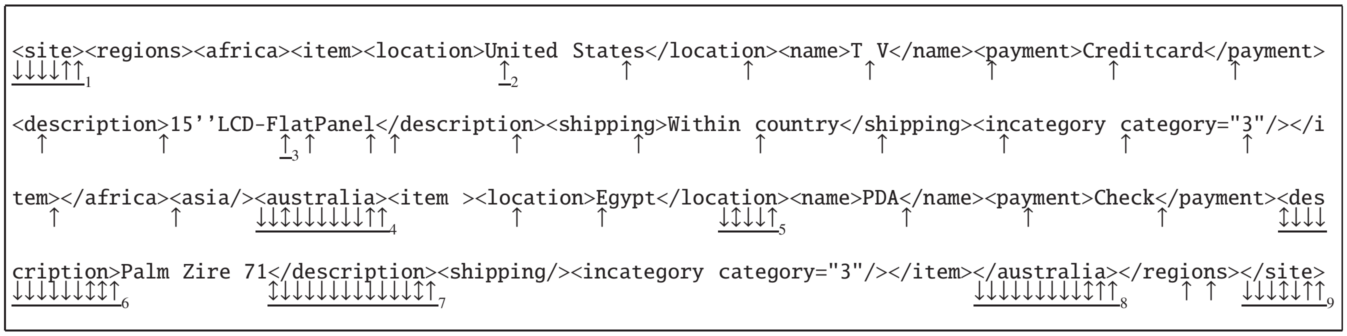
Even in this toy example, only about 22% of all characters need to be inspected. When processing documents in the Gigabyte range, we observe similar ratios. □

Related work. There are several efficient tools for searching XML and SGML documents [2], [18]–[20], evaluating XPath, regular path expressions, or nested text-region algebra [20]. To our knowledge, they all tokenize the complete input. GNU *grep* implements efficient string matching algorithms as done in this paper, yet its focus is search in flat text files. In [21], Aho-Corasick string matching is extended to files that mix single-byte and multi-byte characters. If XML schema is available, this approach can be applied to XML processing, by viewing opening- and closing tags as multi-byte characters. The authors thus build an XML parser, *on top* of which they implement path-matching in the style of [1], [22]. Our work differs in several regards. First of all, our approach is based on a different family of string matching algorithms, the Boyer-Moore and Commentz-Walter algorithm, which aim at skipping as many characters in the input as possible. Second, we present an integrated approach of parsing and path matching, which allows us to skip even larger offsets in the input. Finally, we exploit structure information and order constraints from the schema, as opposed to merely the vocabulary of possible tagnames.

Using the XML stream index (SIX) [2], which registers byte offsets within the input, subtrees can be skipped during XPath evaluation. In contrast to our approach, the SIX is precomputed in a prior pass, which is futile in most streaming scenarios.

Contributions. We make the following contributions.

- We show that established string matching techniques, originally designed for keyword search in flat text files, can be used for highly efficient search and navigation in unparsed, tree-structured XML data.



```
<site><regions><africa><item><location>United States</location><name>T V</name><payment>Creditcard</payment>
<description>15''LCD-FlatPanel</description><shipping>Within country</shipping><incategory category="3"/></i
tem></africa><asia/><australia><item ><location>Egypt</location><name>PDA</name><payment>Check</payment><des
cription>Palm Zire 71</description><shipping/><incategory category="3"/></item></australia></regions></site>
```

Fig. 2. XML prefiltering for the XQuery expression ⟨q⟩{ //australia//description }⟨/q⟩

- We propose a novel approach to XML prefiltering that exploits efficient string matching algorithms.
- We perform a holistic static analysis which reduces XML prefiltering to a set of string matching tasks. At runtime, an automaton switches between individual tasks based on the current state in XML prefiltering.
- The high throughput of our system relies on the idea of skipping parts of the input and a lean core algorithm that executes XML prefiltering with very little management overhead. This is made possible by precompiling the results of static analysis into fixed lookup tables.
- Our extensive experiments demonstrate the persistently high throughput and scalability of our prototype for a variety of datasets, document sizes, and queries. With our technique, in-memory XQuery engines in many practical cases overcome main memory limitations and scale up to documents in the Gigabyte range on currently common hardware. Even throughput rates of traditional SAX parsers are up to the order of one magnitude lower than the rates that we achieve for prefiltering. We conclude that prefiltering systems, that rely on a tokenization of their input, cannot compete with the throughput achieved by our technique.

Structure. In Section II we introduce the highly efficient runtime algorithm, which makes heavy use of precompiled lookup tables to realize efficient document prefiltering. We introduce a projection semantics in Section III, and discuss the static precompilation of lookup tables in Section IV. Our extensive experiments are presented in Section V, and we finish with a short conclusion in Section VI.

II. Runtime algorithm

We next introduce the runtime algorithm that schedules the execution of the single string matching problems to perform XML prefiltering. We assume that a nonrecursive schema is available, but emphasize that all techniques can be extended to handle recursiveness. The algorithm assumes that the input document is valid w.r.t. the DTD.

Example 2: We exemplarily discuss prefiltering for XPath expression `/a/b` against a document valid w.r.t. DTD

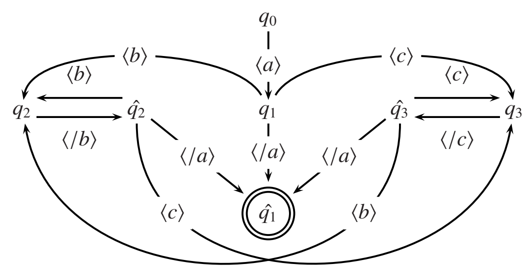
```
<!DOCTYPE a [ <!ELEMENT a (b|c)*>
  <!ELEMENT b #PCDATA>  <!ELEMENT c (b,b?)> ]>.
```

While we formally define a semantics for XML prefiltering in the next section, here it is evident that only the top-level nodes with label a, and their b-labeled children (with their subtrees) need to be preserved in prefiltering.

We decompose the prefiltering task into multiple string matching problems. The *frontier vocabulary* is the set of keywords defining the current string matching problem. We start XML prefiltering with tag $\langle a \rangle$ in the frontier vocabulary. We use the Boyer-Moore algorithm for single keyword search to match tag $\langle a \rangle$ in the input. Once we have located this keyword, we consider the frontier vocabulary with the tokens $\langle b \rangle$, $\langle c \rangle$, and $\langle /a \rangle$. As this defines a multi-keyword search, we use the Commentz-Walter algorithm to detect the closest match for any of these keywords. (1) If the closest match is token $\langle b \rangle$, then we have found a relevant node. (2) If we instead find token $\langle c \rangle$, we can ignore the subtree underneath. (3) If we match $\langle /a \rangle$, we have reached the end of the parent node. This way, we recognize just enough of the structure of the input yet without parsing the complete input into tokens.

Part (2) is crucial, because if we only scan for tags $\langle b \rangle$ and $\langle /a \rangle$, we cannot distinguish "$\langle a \rangle\langle b \rangle \ldots$" and "$\langle a \rangle\langle c \rangle\langle b \rangle \ldots$", and could mistake a b-labeled child of node c for a child of a. At the same time, if $\langle c \rangle$ has been matched, we can *immediately* search for $\langle /c \rangle$ without parsing the tokens in the subtree of the c-labeled node. □

Runtime data structures. The change between frontier vocabularies is captured by the runtime-automaton. Given the current automaton state and reading position in the input, string matching algorithms scan for the closest token for which a transition is defined. Four statically compiled tables (or associative arrays) provide all information required at runtime. Table A in Figure 3 (visualized as a graph) holds the transition function, mapping the current state and an input token from the frontier vocabulary to the next state. q_0 denotes the initial state. Table V provides fast access to the frontier vocabulary in each state. Tags can contain attributes or whitespace, so the string search does not consider the trailing bracket of the tag. Table J stores *initial jump offsets*, i.e. the number of positions that can initially be skipped when entering a new state. This is motivated in Example 3. The actions for each state are stored in table T. We can perform no operation ("nop") or



Runtime-automaton A.

q	$V[q]$	q	$J[q]$	q	$T[q]$
q_0	{ "$\langle a$" }	q_0	0	q_0	`nop`
q_1	{ "$\langle /a$", "$\langle b$", "$\langle c$" }	q_1	0	q_1	`copy tag`
$\hat{q}_1$	{ }	$\hat{q}_1$	0	$\hat{q}_1$	`copy tag`
q_2	{ "$\langle /b$" }	q_2	0	q_2	`copy on`
$\hat{q}_2$	{ "$\langle /a$", "$\langle b$", "$\langle c$" }	$\hat{q}_2$	0	$\hat{q}_2$	`copy off`
q_3	{ "$\langle /c$" }	q_3	4	q_3	`nop`
$\hat{q}_3$	{ "$\langle /a$", "$\langle b$", "$\langle c$" }	$\hat{q}_3$	0	$\hat{q}_3$	`nop`

Fig. 3. Lookup tables V, J, and T

```
q := q0;  // current state
c := 0;   // cursor position
while j ≤ end-of-file and q is not final do
begin
  c := c + J[q];   // initial jump offset
  if |V[q]| = 1
  then perform single-keyword search for token in V[q];   // (BM)
  else  perform multi-keyword search for tokens in V[q];   // (CW)
  t := the matched token from V[q];
  a := tagname in token t;
  shift cursor c to the right until reading "⟩" or "/⟩"   // (★)
    to determine matched tag;
  if tag is an opening tag then
      assign q := A[q, ⟨a⟩] and perform action T[q];
  else if tag is a closing tag then
      assign q := A[q, ⟨/a⟩] and perform action T[q];
  else if tag is a bachelor tag then begin
      assign q := A[q, ⟨a⟩] and perform action T[q];
      assign q := A[q, ⟨/a⟩] and perform action T[q];
  end
end
```

Fig. 4. The runtime algorithm

copy the current tag without or with its attributes ("`copy tag [+ atts]`"). To output a node with its subtree, we copy the input from the start position of the opening tag up to the last position of its closing tag ("`copy on/off`").

Example 3: Consider Figure 3. Assume we are in state q_3 and at some reading position in the input document. Then the frontier vocabulary is $V[q_3]$ = {"$\langle /c$"}, so we search for "$\langle /c$". We know from the DTD that node c has at least one child. The shortest string encoding of this child is as "$\langle b/\rangle$", using four characters. Thus, when starting to search for "$\langle c/$", we skip $J[q_3] = 4$ characters. □

Runtime algorithm. Figure 4 shows the runtime algorithm, which switches between string matching problems. The current state is denoted by q, and the current reading position in the input, the "cursor", by c. We iterate the following steps. In an initial jump, we shift the cursor $J[q]$ positions to the right. Then we search for the closest token from the frontier vocabulary $V[q]$. For unary frontier vocabularies, we utilize the Boyer-Moore algorithm (*BM*), otherwise the Commentz-Walter algorithm (*CW*). We then scan to the right for the end of the tag. If we have found an opening or a closing tag, we enter the next state and perform the associated action. In case we have found a bachelor tag $\langle a/\rangle$, we evaluate the steps for the opening tag $\langle a\rangle$ and the closing tag $\langle /a\rangle$ one after the other. The cursor now points to the last position with character "$\rangle$" of the matched token, and the iteration proceeds.

There is a special case that we omitted for simplicity. DTDs may specify tagnames that are prefixes of each other, such as `Abstract` and `AbstractText` in the Medline DTD [23]. Thus, if we scan for tag $\langle Abstract\rangle$ and locate "$\langle Abstract$", we must assert that we have not matched $\langle AbstractText\rangle$ instead. This can be done during the scan for the end of the tag (marked by (★) in the algorithm).

III. Projection semantics

We capture the relevant data in XML prefiltering by projection paths, as defined in [5]. A *simple path* is a sequence of XPath downward navigation steps without predicates, composed by "/". A *projection path* is an expression /*simplePath* or /*simplePath*#. The flag "#" indicates that the descendants of selected nodes are also required for query evaluation. We use the path extraction algorithm from [5], which covers full XQuery with downward XPath axes. Additionally, we extract the path `/*` by default. This path matches the top-level node and ensure well-formed output in prefiltering.

Example 4: The static analysis for the XQuery

```
<q>{//australia/description}</q>
```

extracts the paths `//australia//description#` and `/*`. For query Q13 from the XMark benchmark [17], i.e.

```
for $i in /site/regions/australia/item
return <item name="{$i/name/text()}">
                  {$i/description} </item>,
```

we extract `/site/regions/australia/item/name#`, `/site/regions/australia/item/description#`, and `/*`. □

Projection safety. For XML prefiltering to be correct, it needs to preserve all data relevant for the query. Only then will query evaluation on the projected document return the same result as on the original input. We formulate this as *projection-safety*. The definition is based on projection paths, so as not to restrict prefiltering to a specific query language. It assumes the standard XPath semantics in evaluating path expressions against documents, and interprets the #-flag as step expression `descendant-or-self::node()`. In evaluating an XPath expression on an XML document, we compute a list of well-formed XML documents and strings. To compare the evaluation of XPath expressions over XML documents and their projections, we define an equality relation over such lists. This relation captures the idea that from the viewpoint of XPath evaluation, the original and the projected document cannot be distinguished.

Definition 1: Let L_1 and L_2 be two lists of XML documents and strings. We say L_1 and L_2 are top-level equal iff they have the same length, and the ith elements of L_1 and L_2 are either two equal strings or two XML documents trees where the root nodes have the same label.

Example 5: Let s be a fixed string. Then the lists $[\langle a\rangle b\langle/a\rangle, s]$, $[\langle a\rangle c\langle/a\rangle, s]$, and $[\langle a\rangle\langle/a\rangle, s]$ are pairwise top-level equal. □

Definition 2: Let f be a function mapping XML documents to XML documents. Let $\mathcal{P}$ be a set of projection paths. Then function f is projection-safe w.r.t. $\mathcal{P}$ if for all projection paths p in $\mathcal{P}$ and all XML documents X, the results of p evaluated on X and $f(X)$ are top-level equal.

A safe projection semantics. Our projection semantics has been successfully implemented in [7]. We consider XML documents $D = t_1 \dots t_n$, where each token t_i is either an opening, closing, or bachelor tag, or character data. We define function *branch*(t_i), which returns the document branch of ancestor nodes from the root up to t_i. For instance, in Figure 2, we have $branch(\langle name\rangle) = branch(\langle/name\rangle) = \langle site\rangle\langle regions\rangle\langle item\rangle\langle name/\rangle\langle/item\rangle\langle/regions\rangle\langle/site\rangle$ for the *name*-tags in line 1. Finally, we introduce set $\mathcal{P}^+$, which extends the projection paths $\mathcal{P}$ by all prefix paths in $\mathcal{P}$, e.g. for path `/a/b` in $\mathcal{P}$ we add paths `/` and `/a`.

Definition 3: Let $\mathcal{P}$ be a set of projection paths, $\mathcal{P}^+$ its extension by all prefix paths in $\mathcal{P}$, and let t_{i_n} be a token in document D. If t_{i_n} is a tag node, let branch(t_{i_n}) be $\langle t_{i_1}\rangle\dots\langle t_{i_{n-1}}\rangle\langle t_{i_n}/\rangle\langle/t_{i_{n-1}}\rangle\dots\langle/t_{i_1}\rangle$, and otherwise $\langle t_{i_1}\rangle\dots\langle t_{i_{n-1}}\rangle$"$t_{i_n}$"$\langle/t_{i_{n-1}}\rangle\dots\langle/t_{i_1}\rangle$. Token t_{i_n} is relevant according to $\mathcal{P}$ if one of the following conditions holds.

- C_1: *the leaf node in branch(t_{i_n}) is matched by a path in $\mathcal{P}^+$,*
- C_2: *any node in branch(t_{i_n}) is matched by a path in $\mathcal{P}^+$ marked with #,*
- C_3: *there is a tag `t` s.t. $\mathcal{P}^+$ contains paths of the form $/p_1/\dots/p_i/$`t` and $/p'_1/\dots/p'_j//$`t`, which both match the leaf node in $\langle t_{i_1}\rangle\dots\langle t_{i_{n-1}}\rangle\langle t/\rangle\langle/t_{i_{n-1}}\rangle\dots\langle/t_{i_1}\rangle$.*

Condition C_1 covers all document nodes directly matched by a projection path or its prefix, while C_2 captures all descendants of nodes matched by #-marked paths. Finally, C_3 maintains vital ancestor-descendant relationships, as motivated by the following example.

Example 6: For the query `<x>{/a/b,//b}</x>` we compute $\mathcal{P}$={`/*`, `/a/b#`, `//b#`} and $\mathcal{P}^+$={`/`, `/a`, `/*`, `/a/b#`, `//b#`}. All tokens in the document $D = \langle a\rangle\langle c\rangle\langle b\rangle T\langle/b\rangle\langle/c\rangle\langle/a\rangle$, are relevant according to $\mathcal{P}$. Both the a and b-labeled tags are relevant according to condition C_1, because the leaf nodes of their document branches $branch(\langle a\rangle)$=$branch(\langle/a\rangle)$=$\langle a/\rangle$ and $branch(\langle b\rangle)$=$branch(\langle/b\rangle)$=$\langle a\rangle\langle c\rangle\langle b/\rangle\langle/c\rangle\langle/a\rangle$ are matched by projection paths `/a`, and `//b#`, respectively. The branch of text node "T" is the document D itself. Clearly, the b-labeled nodes in the document are matched by the #-flagged path `//b#`, hence the text node is relevant due to C_2. Finally, consider the c-tags with $branch(\langle c\rangle) = branch(\langle/c\rangle) = \langle a\rangle\langle c/\rangle\langle/a\rangle$. We choose $t = b$ in condition C_3, and $\mathcal{P}^+$ contains the paths $/a/b$ and $//b$#, which both match $\langle a\rangle\langle b/\rangle\langle/a\rangle$. Thus, the c-tags are also relevant. Note that the c-tags are indeed required for query evaluation, as the evaluation results on the original document and on document $\langle a\rangle\langle b\rangle T\langle/b\rangle\langle/a\rangle$ differ. □

Based on the definition of relevant nodes, we implement XML prefiltering by preserving exactly the relevant nodes. The lemma below states the correctness of this approach.

Lemma 1: Let $\mathcal{P}$ be a set of projection paths. A function over XML documents which preserves all nodes relevant according to $\mathcal{P}$ with their ancestor-descendant and following-relationships is projection-safe w.r.t. $\mathcal{P}$.

As shown in [7], this approach to XML prefiltering can be evaluated on-the-fly, in a single pass over the input.

IV. Static compilation of lookup tables

We statically compute the runtime lookup tables from projection paths and a nonrecursive DTD.

Computing the runtime-automaton. We compile the DTD into an automaton. *DTD-automata* are finite-state automata (FSAs) that recognize all XML documents valid w.r.t. a DTD. Ultimately, we want to associate actions with FSA states, as in Figure 3. To this end, we use Glushkov automata [24], a class of FSAs where all transitions into a state carry the same label. This property is called *homogeneity* [25]. A state in a homogeneous FSA is called *t-labeled* if its incoming transitions carry label t. State q_0 is the initial state, and dual states q and $\hat{q}$ distinguish reading opening- and closing tags.

Example 7: The DTD-automaton in Figure 5 has been constructed for the DTD from Example 2. □

We compute the runtime-automaton from a subgraph of the transitive closure of the graph defined by the DTD-automaton. Our goal is to select a *small* subgraph, as we do not want to recognize all tokens in the input, but rather skip parts of the input unparsed. Thereby, we need to ensure that we visit all tags that are part of relevant data, as this data needs to be preserved in XML prefiltering. As shown in Example 2, we may also have to stop over at additional nodes to maintain a minimum amount of orientation.

Definition 4: Given a homogeneous FSA D and the set of states S, the subgraph-automaton $D_{|S}$ is defined over the states $S \cup \{q_0\}$ as follows.

- *The initial state of D is also the initial state of $D_{|S}$.*

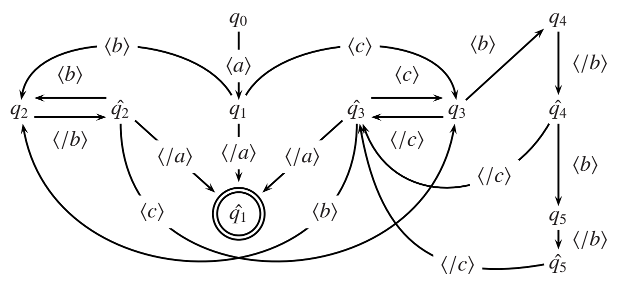


Fig. 5. DTD-automaton

- *For each sequence* $q \overset{t_{i_1}}{\to} q_{i_1} \overset{t_{i_2}}{\to} \cdots \overset{t_{i_n}}{\to} q_{i_n} \overset{t}{\to} p$ *of transitions of D, where* $n \geq 0$ *and only states q and p are in S,* $D_{|S}$ *has a transition* $q \overset{t}{\to} p$.
- *A state q of* $D_{|S}$ *is final if (1) q is a final state in D, or (2) there is a sequence* $q \to q_{i_1} \to \cdots \to q_{i_n} \to p$ *for* $n \geq 0$ *in the transitions of D where only state q is in S and p is a final state of D.*

By construction, $D_{|S}$ is also homogeneous.

The layout of the DTD-automaton reflects the parent-child relationships between nodes in the accepted XML documents. A state p is a *parent state* of a state q if there is a well-formed XML document where the nodes matched by p are parents of the nodes matched by q.

Example 8: In Figure 5, state q_0 has no parent states, but it is the parent state of q_1 and $\hat{q}_1$. In return, q_1 and $\hat{q}_1$ are the parent states of $q_2, \hat{q}_2$ and $q_3, \hat{q}_3$. □

For each state q_{i_n} in the DTD-automaton, there is a unique sequence (q_0, q_{i_1}), (q_{i_1}, q_{i_2}), …, $(q_{i_{n-1}}, q_{i_n})$ of parent-child states from initial state q_0 up to state q_{i_n}. Let t_{i_k} be the label of state q_{i_k}, for $1 \leq k \leq n$. We define the *document branch* of state q_{i_n} as $\langle t_{i_1}\rangle\langle t_{i_2}\rangle \ldots \langle t_{i_n}/\rangle \ldots \langle /t_{i_2}\rangle\langle /t_{i_1}\rangle$.

Example 9: In Figure 5, state q_0 has the empty document branch, states q_1 and $\hat{q}_1$ have document branch $\langle a/\rangle$, and states q_2 and $\hat{q}_2$ have document branch $\langle a\rangle\langle b/\rangle\langle /a\rangle$. □

We next define the semantics of projection paths evaluated on DTD automata instead of the XML documents.

Definition 5: A state q in the DTD-automaton is relevant according to $\mathcal{P}$ *if the leaf node of its document branch is relevant according to* $\mathcal{P}$ *by Definition 3.*

Example 10: Consider the DTD-automaton from Figure 5, and the projection paths $\mathcal{P}_1$ = {`/*`, `/a/b#`, `//b#`}. All states in the DTD-automaton are relevant according to $\mathcal{P}_1$. The (empty) document branch of q_0 is matched by the empty path step `/`. States q_2 and $\hat{q}_2$ with document branch $\langle a\rangle\langle b/\rangle\langle /a\rangle$ are also relevant, because the leaf node $\langle b/\rangle$ is matched by path `/a/b#`. For states q_3 and $\hat{q}_3$ with document branch $\langle a\rangle\langle c/\rangle\langle /a\rangle$, the token $\langle c/\rangle$ is relevant according to condition C_3 in Definition 3. In contrast, only $q_0, q_1, \hat{q}_1$, and $q_2, \hat{q}_2$ are relevant according to $\mathcal{P}_2$ = {`/*`, `/a/b#`}. □

```
Given DTD-automaton $D$ and a set of projection paths $\mathcal{P}$,
(1) choose a subset $S$ of states as follows:
    (a) for each state $q$ in the DTD-automaton do
            if $q$ is relevant according to $\mathcal{P}$, then add $q$ to $S$;
    (b) for each pair of dual states $q$ and $\hat{q}$ in S do
        begin
            let $R$ contain all states $p$ of $D$ s.t.
              there is a path from $q$ to $\hat{q}$ via $p$ in $D$;
            if $R \subseteq S$ then remove states in $R$ from $S$;
        end
    (c) while there are changes to $S$ do
            if $D$ has transitions $q \to q_1 \to \cdots \to q_n \to p$
              and $q \to p_1 \to \cdots \to p_m \to p'$ for $n, m \geq 0$,
              where only $q$ and $p$ are in $S$
              and $p$ and $p'$ have the same label
        then add the parent states of $p'$ to S;
(2) compute the subgraph automaton $D_{|S}$;
(3) determinize $D_{|S}$ to obtain the runtime-automaton;
```

Fig. 6. Compilation of the runtime-automaton

Figure 6 shows the static compilation of the runtime-automaton, which we illustrate in the following examples.

Example 11: We consider the DTD-automaton from Figure 5 and $\mathcal{P}$ = {`/*`, `/a/b#`}. In step (1), we select states for computing a subgraph-automaton. Step (a) initializes set S with $q_0, q_1, \hat{q}_1, q_2$, and $\hat{q}_2$. This ensures that the runtime algorithm visits all nodes that must be preserved for query evaluation. Step (b) does not apply here. In step (c), we again extend S. For instance, we observe the transitions $q_1 \overset{\langle b\rangle}{\to} q_2$ and $q_1 \overset{\langle c\rangle}{\to} q_3 \overset{\langle b\rangle}{\to} q_4$ where q_2 and q_4 are b-labeled, but only q_1 and q_2 are contained in S. Hence, S is extended by q_3 and $\hat{q}_3$. This ensures that the runtime-algorithm is not thrown off-track when it skips input passages. Step 2 computes the subgraph-automaton, that is shown in Figure 3 (since the FSA is already deterministic, it is identical to the runtime-automaton). □

Example 12: We next consider $\mathcal{P}$ = {`/*`, `//c#`} and the DTD-automaton from before. In step 1(a), we compute $S = \{q_0, q_1, \hat{q}_1, q_3, \hat{q}_3, q_4, \hat{q}_4, q_5, \hat{q}_5\}$. Yet at runtime, once we have located a token $\langle c\rangle$ in the input, the complete subtree of the matched node is relevant, and copied to the output anyway. Hence, we can scan for the closing tag $\langle /c\rangle$ without locating the tags for any of its descendants. We thus prune S to $S = \{q_0, q_1, \hat{q}_1, q_3, \hat{q}_3\}$ in step 1(b). □

Remaining lookup tables. The compilation of the remaining runtime-tables is quite straightforward. Table V is constructed from the transitions of the runtime automaton. The initial jump offsets are computed based on the runtime-automaton and the DTD-definition. When computing offsets, required attributes may be factored in.

In defining the action table, we unambiguously map an action to each state, exploiting the fact that the runtime-automaton is homogeneous (homogeneity is preserved by determinization via subset construction [25]). Table T is derived as follows. States that do not describe relevant nodes are assigned action "`nop`". For the remaining states, we consider the pairs of states q and $\hat{q}$ for reading the opening- and closing-tag of a node. If the leaf node in the document branch associated with q and $\hat{q}$ satisfies condition C_2 in Definition 3, then we are interested in the descendants of this node and assign $T[q]$ ="`copy on`" and $T[\hat{q}]$= "`copy off`". Otherwise, we assign action "`copy tag`" for both states, possibly also copying the attributes for the opening tag, depending on the matched projection paths.

Correctness. Let $R(D, \mathcal{P}, X)$ be the runtime algorithm from Figure 4 computed for DTD D and the set of projection paths $\mathcal{P}$ in document X. As the following theorem states, the runtime-algorithm preserves all relevant nodes.

Theorem 1: Let X be a document valid w.r.t. a DTD D, and $\mathcal{P}$ be a set of projection paths. Algorithm $R(D, \mathcal{P}, X)$ preserves all nodes relevant according to $\mathcal{P}$ with their ancestor-descendant and following-relationships as in X.

It follows from Lemma 1 that $R(D, \mathcal{P}, X)$ implements a projection-safe function, and thus can be used for XML prefiltering on documents conforming to the DTD.

V. Experiments

We have implemented a prototype in C++, called *SMP*. Our prototype takes the projection paths and a nonrecursive DTD as input and performs static analysis (c.f. Section IV). The data structures for string search are computed lazily, when an automaton-state is first entered. SMP uses a pre-allocated buffer to read the document in fixed-size chunks, which we set to eight times the system page size.

Experimental Setup. All tests were carried out on a Core2 Duo IBM ThinkPad Z61p with a T2500 2.00GHz CPU, 1GB RAM available, running Ubuntu Linux 6.06 LTS. We run Java query engines with J2RE 1.5.0_09. *Usr* is the total number of seconds the process used directly, and *Sys* the CPU seconds used by the system on behalf of the process. *CPU* workload is computed as *Usr+Sys* divided by total running time. To avoid warm caches, we alternately ran experiments and loaded large dummy files into main memory.

Outline. The experiments are organized as follows. We first examine the performance characteristics of SMP on different datasets and query workloads. Next, we study how SMP performs when evaluated in sequence or in pipelining with in-memory XPath and XQuery engines. Finally, we contrast the throughput of SMP with that of an industrial-strength SAX parser, to demonstrate the overhead caused by input tokenization alone. We conclude with a comparison of SMP with an existing prefiltering tool that also exploits schema information, but relies on input tokenization.

A. *Performance characteristics of SMP*

We study the behavior of SMP for different queries, documents, and documents sizes. We ran experiments with XMark [17], MEDLINE [23], and Protein Sequence [26] datasets and the corresponding DTDs. Due to space limitations, we refer to [27] for the Protein Sequence results.

XMark data. We tested SMP with data from the XMark benchmark [17]. Note that the XMark DTD allows recursive lists within item descriptions. We modified the DTD accordingly and restricted our experiments to queries XM1-14 and XM17-20, which do not address the recursive lists. Table I shows our results for a 5GB XMark document. To provide an idea of how SMP performs on smaller documents, we list the maximum deviation "±" (in positive or negative direction) on the 10MB, 100MB, and 1GB documents for selected values. We state the size of the projected document, and the maximum memory consumption (*Mem*). The total runtime (*Time*), the sum of *Usr* and *Sys* time, and the average *CPU* load are also listed. The static analysis, which comprises parsing of the DTD and of the files containing the projection paths, as well as the construction of the lookup tables A, V, J, and T, is included in the time measurements, and varied between 0.03s and 0.2s.

States is the number of states in the runtime-DFA. The value of *CW + BM* denotes the number of states for which Commentz-Walter (*CW*) or Boyer-Moore (*BM*) lookup tables are constructed. For instance, for query XM1, the runtime-DFA has 9 states, two of which require *CW* lookup tables, and six of which need *BM* lookup tables.

When we scan the input there are forward shifts performed in string pattern matching and initial jump offsets computed by static analysis. *∅Shift Size* denotes the average size of forward shifts, which depends on the lengths of keywords in the frontier vocabularies, but also on the structure of the input document. When we verify a potential match for a keyword, forward shifts are followed by a scan from right to left. Hence, *∅Shift Size* cannot be used to compute *Char Comp*, the percentage of character comparisons relative to the document size. *Initial Jumps* denotes the percentage of characters skipped by initial jump offsets alone. The small deviations (±) for different input sizes suggest that the XMark data generator creates documents that are very similar in their structure.

We observe that larger outputs go hand in hand with higher total processing times. For instance, prefiltering for query XM14 produces the largest output, and requires the longest running time. The *Usr+Sys* time is mainly driven by the number of character comparisons, while the *CPU* load depends on the output size and the number of characters comparisons, and ranges between 11% and 21%. Thus, the system spends most of the time holding out for new data from the disk. The average size of forward shifts depends on the input and the size of the tags used in the projection paths. In evaluating query XM5, we observe comparatively large average forward shifts. Consequently, the *Usr+Sys* time is low, and SMP inspects only about 10% of the input (*Char Comp*). Overall, SMP inspects at most 23% of the input. Comparatively little can be gained

TABLE I

SMP projection results on 5GB XMark data

	XM1	XM2	XM3	XM4	XM5	XM6	XM7	XM8	XM9
Proj. Size	67.64MB	123.26MB	123.26MB	151.14MB	22.10MB	12.03MB	105.74MB	93.78MB	121.01MB
Mem	1.64MB	1.72MB	1.72MB	1.75MB	1.68MB	1.64MB	1.77MB	1.72MB	1.78MB
Time	252.48s	283.33s	281.8s	290.42s	252.35s	241.70s	256.94s	252.95s	258.93s
Usr+Sys	31.00s	41.65s	41.59s	42.40s	19.91s	29.36s	50.47s	35.91s	30.41s
CPU [%]	12.52±2.51	14.99±0.75	15.04±2.53	14.90±4.90	8.05±1.52	12.39±4.89	20.02±7.52	14.48±5.73	11.98±4.68
States (CW+BM)	9 (2 + 6)	11 (4 + 6)	11 (4 + 6)	13 (5 + 7)	9 (2 + 6)	7 (2 + 4)	11 (4 + 6)	15 (4 + 10)	25 (6 + 18)
∅ Shift Size [char]	5.72±0.02	7.62±0.01	7.62±0.01	7.65±0.01	10.83±0.04	5.17±0.00	6.55±0.04	7.42±0.00	7.50±0.05
Initial Jumps [%]	0.32±0.00	1.42±0.01	1.42±0.01	1.37±0.00	0.43±0.00	1.98±0.01	2.61±0.01	0.75±0.01	1.18±0.01
Char Comp. [%]	18.86±0.05	15.8±0.04	15.8±0.04	16.37±0.12	9.87±0.02	19.91±0.03	18.40±0.16	15.10±0.04	15.29±0.15

	XM10	XM11	XM12	XM13	XM14	XM17	XM18	XM19	XM20
Proj. Size	307.63MB	95.37MB	65.73MB	137.63MB	1357.28MB	75.44MB	21.08MB	71.22MB	38.52MB
Mem	1.96MB	1.74MB	1.74MB	1.66MB	1.64MB	1.67MB	1.69MB	1.66MB	1.67MB
Time	295.92s	256.54s	256.85s	250.35s	321.03s	255.94s	249.29s	243.67s	249.88s
Usr+Sys	54.94s	34.85s	32.40s	26.39s	53.71s	34.95s	23.54s	32.16s	31.67s
CPU [%]	18.93±2.73	13.85±4.47	12.86±2.14	10.75±3.25	17.07±2.93	13.92±1.89	9.63±0.83	13.45±1.78	12.92±4.59
States (CW+BM)	33 (10 + 22)	17 (5 + 11)	15 (5 + 9)	13 (2 + 10)	9 (2 + 6)	11 (3 + 7)	9 (3 + 5)	11 (2 + 8)	9 (3 + 5)
∅ Shift Size [char]	5.68±0.01	6.58±0.01	6.60±0.02	6.06±0.00	5.16±0.01	5.72±0.01	8.29±0.04	5.17±0.00	5.75±0.00
Initial Jumps [%]	0.16±0.01	1.85±0.01	2.00±0.00	0.13±0.00	1.35±0.01	0.32±0.00	0.80±0.01	1.64±0.01	0.59±0.01
Char Comp. [%]	22.38±0.01	17.15±0.11	16.81±0.11	17.17±0.03	21.24±0.08	18.99±0.03	12.95±0.03	20.57±0.03	18.67±0.03

TABLE II

SMP projection results on 656MB MEDLINE data

M1	/MedlineCitationSet//CollectionTitle
M2	/MedlineCitationSet//DataBank[DataBankName/text()="PDB"]/AccessionNumberList
M3	/MedlineCitationSet//PersonalNameSubjectList/PersonalNameSubject[LastName/text()="Hippocrates" or DatesAssociatedWithName="Oct2006"] /TitleAssociatedWithName
M4	/MedlineCitationSet//CopyrightInformation[contains(text(),"NASA")]
M5	/MedlineCitationSet/MedlineCitation[contains(MedlineJournalInfo//text(),"Sterilization")]/DateCompleted

	M1	M2	M3	M4	M5
Proj. Size	0MB	0.42MB	0.34MB	0.19MB	47.4MB
Mem	1.94MB	2.01MB	2.11MB	1.99MB	2.00MB
Time	33.72s	33.62s	33.69s	33.47s	35.51s
Usr+Sys	2.96s	4.46s	3.02s	3.24s	4.35s
CPU [%]	9.02	13.76	9.26	9.99	12.43
States (CW+BM)	5 (1 + 1)	9 (3 + 5)	13 (4 + 4)	5 (2 + 2)	9 (3 + 5)
∅ Shift Size [char]	12.24	6.86	12.49	12.69	13.43
Initial Jumps [%]	0.00	0.00	0.00	0.01	7.61
Char Comp. [%]	8.37	14.63	8.4	8.52	9.81

by initial jump offsets in this set of experiments.

Note that queries XM2 and XM3 have identical projection paths, which is reflected in similar results.

MEDLINE. We consider the XPath expressions M1-M5 from Table II and a 656MB MEDLINE [23] document. In contrast to XMark, MEDLINE data is not synthetic. The queries only use paths satisfiable by the DTD.

Table II summarizes the results. Query M1 searches for nodes which are defined by the DTD, but do not occur in the input. Scanning with an average forward shift of about 12 characters, only 8.37% of the input is inspected.

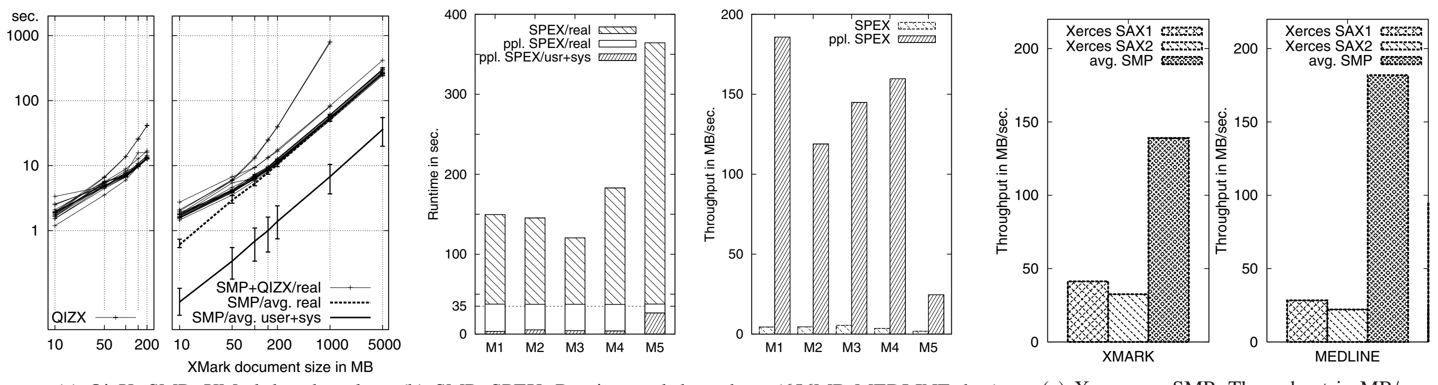
In comparison to the XMark results in Table I, XML prefiltering on the MEDLINE dataset features larger average forward shifts (*∅Shift Size*) due to longer tagnames in the queries. For queries M1-M4, no initial jumps were possible. Upon closer inspection, it turns out that the MEDLINE DTD specifies many nodes as optional, but only required nodes are beneficial for the computation of initial jump offsets. For query M5, we observe comparatively large initial jumps. In total, about 50MB of the 656MB input document are skipped by initial jumps alone.

B. Filtering for XQuery/XPath evaluation

We next examine how main-memory XML query engines perform when they are run in combination with SMP, and show that the low CPU workload of SMP facilitates efficient pipelining of prefiltering and query evaluation.

XQuery evaluation. We evaluated the XMark queries from before with the main memory-based XQuery processors QizX [14] and Saxon [15]. For the input tested, QizX was superior to Saxon regarding both execution time and main memory consumption, so we only report on the results for the QizX engine. Runtime and main memory were limited to 1h and 1GB, respectively. We study a sequential setup, where SMP writes the prefiltered input back to disk, and QizX reloads the (prefiltered) document from disk to evaluate the query. All time measurements include the additional write and read operations.

Figure 7(a) shows the results. The graphs use log scale for the x-axis (document size) and the y-axis (time). Without projection, QizX can successfully load the input and evaluate all queries within the memory and runtime limitations on documents up to 200MB, but fails for all queries on the 1GB and 5GB documents. We show the runtimes for stand-alone query evaluation on the left. To the right, we plot the runtimes for sequential evaluation of SMP and QizX. On documents up to 200MB, the runtimes differ only marginally, due to the overhead of one extra write and read. When coupled with prefiltering, QizX can evaluate all queries for the 1GB document, and still 15 queries on the 5GB document. We can



(a) QizX+SMP: XMark benchmarks (b) SMP+SPEX: Runtime and throughput (656MB MEDLINE data) (c) Xerces vs. SMP: Throughput in MB/sec

Fig. 7. SMP benchmark results

discern two outliers (XM11 and XM15) which close in on the timeout for 1GB, and fail for the 5GB document.

On the right, we further depict the average real time of SMP prefiltering, and the minimum and maximum values for all queries. The average user and system time is well below the real time, indicating that SMP predominantly waits for new input from the disk.

In summary, the results show that in-memory XQuery engines such as QizX can be made to scale to inputs in the Gigabyte range when coupled with SMP.

XPath in pipelining with SMP. The SPEX [3] XPath evaluator is a representative of a class of engines such as XFilter, YFilter [1], and the XPush Machine [2], which were developed for XML stream processing. While the latter systems evaluate high workloads of queries, SPEX and SMP focus on single queries. We expect similar results as for SPEX when combining XFilter, YFilter, and the XPush machine with our prefiltering tool.

In Figure 7(b) on the left, we show runtimes for the XPath queries from Table II on MEDLINE data. We compare two scenarios, where we ran SPEX on the unprojected document, and next projected the document using SMP and *piped* the output directly into SPEX (denoted “ppl. SPEX”). It is remarkable that, in the pipelined scenario, evaluation real time differs only marginally from the real time for prefiltering alone (see Table II). In the plot, this is indicated by the 35 seconds line. In particular, we observe that the computational effort needed for query evaluation mostly manifests in the SPEX *Usr+Sys* time (see “ppl. SPEX/usr+sys”) rather than the real time. For instance, the projected output for M5 is still of considerable size (47.4MB), which causes a comparatively high *Usr+Sys* time. For the other queries, SMP filters out large parts of the input, so SPEX *Usr+Sys* time is lower. We conclude that the low CPU load of SMP enables an effective interleaving of prefiltering and query evaluation.

To the right in Figure 7(b), we compare the throughput achieved by SPEX as a stand-alone tool with pipelining of SMP and SPEX. The difference is significant, and we reach up to 190 MB/sec for query M1. The throughput for M5 is smaller, because the projected document is still large, yet the pipelined system remains superior.

C. Parsing and projection

We next show that on the queries and datasets previously discussed, SMP achieves a significantly higher throughput than an industrial-strength SAX parser. SAX parsers are used by virtually all competing approaches to tokenize the input, which suggests that these systems are inevitably inferior in terms of scalability. To affirm this claim, we compare SMP against a prefiltering tool that also exploits schema knowledge, but which relies on a tokenization of the input.

Xerces. All existing approaches to XML prefiltering process XML documents that are tokenized, e.g. by a SAX parser. We next compare the throughput of SMP and the Xerces C++ [28] SAX parser. We have built a minimal application on top of the Xerces API that just parses the input into tokens. Note that the Xerces SAX parser checks well-formedness by default.

The results for XMARK (5GB) and MEDLINE datasets are visualized in Figure 7(c). The program throughput of tokenizing the input with Xerces, either using the SAX1 or the SAX2 XML reader, is well below the average program throughput that SMP achieves in prefiltering the same data for both the queries of Table I (XMark) and Table II (MEDLINE), respectively. Even pipelining SMP prefiltering and XPath evaluation with SPEX has a higher throughput on MEDLINE data than just tokenizing the input with Xerces (c.f. Figure 7(b)).

Overall, SMP is by a factor of 3–9 faster than Xerces while at the same time performing a more complex data management task. The results confirm that the throughput achieved by our approach substantially surpasses that of projection systems that rely on a tokenization of their input.

Type-based projection. Type-based projection (TBP) [6] is a natural choice for comparison with SMP, as it also exploits schema knowledge, but tokenizes its complete input. To the best of our knowledge, TBP is the only operational publicly available projection tool.[1]. TBP performs a powerful static

[1]We were not able to obtain a version of Galax with the projection feature mentioned in [5].

TABLE III
PROJECTION OF 1GB XMARK DATA

	Type-based Projection (OCaml)			SMP (C++)		
	Usr+Sys	Mem	Proj. Size	Usr+Sys	Mem	Proj. Size
XM3	756.77s	3.36MB	26.52MB	8.11s	1.72MB	24.62MB
XM6	812.56s	3.36MB	2.59MB	5.37s	1.64MB	2.40MB
XM7	1170.03s	3.36MB	34.50MB	9.83s	1.76MB	21.14MB
XM19	1027.13s	3.36MB	17.92MB	6.06s	1.65MB	14.23MB

analysis to prefilter for additional XPath axes and even for predicates. Yet for the query workload considered here, the sizes of the projected outputs are comparable with SMP. The differences are mainly due to whitespace formatting by TBP, and the fact that SMP is able to discard irrelevant ancestor nodes.

As TBP is written in OCaml, it is difficult to compare runtime results. We tested both the byte code and the native code compilation of TBP. We provide the results for the native code compilation, which turned out to be significantly faster in all cases. We consider the subset of XMark queries benchmarked both by SMP in this paper and by and TBP in [6], namely XM3, XM6, XM7 and XM19. Table III contains the results. Both XML filtering systems get by with an economical main memory consumption, yet the *Usr+Sys* times differ. To put our results into perspective, we point out that in computer language shootouts, OCaml programs rarely perform more than a factor of 20 worse than C++ programs compiled with *g++* (e.g. see [29]). Typically, we may expect a gap of factor 5 to 10. On the 1GB XMark document, SMP requires less than 10 seconds *Usr+Sys* time (and 2MB main memory) for all queries. This is at least a factor of 90 better in comparison to the *Usr+Sys* time consumed by TBP. The throughput of SMP is in the order of two magnitudes higher than that of TBP. This exceeds the difference one might expect by the choice of programming language. For the 10MB and 100MB documents we observe similar results, where our implementation is faster by at least a factor of 84. On the 5GB document, TBP exceeded the time limit of one hour.

In summary, SMP is able to project the 5GB document faster than type-based projection can process 1GB. Also, SMP requires fewer CPU seconds (*Usr+Sys*) on the 5GB document than TBP needs for projecting 1GB.

VI. CONCLUSION

We have shown that established string matching techniques, originally designed for searching flat text files, can be leveraged for the efficient search and navigation in XML documents and streams. We present a highly efficient approach to XML prefiltering based on these ideas. Our prototype implementation confirms that, just like in traditional keyword search, the search and navigation in tree-structured XML documents with advanced string matching techniques clearly outperforms systems that rely on a character-by-character processing of the input.

VII. ACKNOWLEDGMENTS

We thank Johannes Gehrke, Christian Lindig, Dan Olteanu, Maya Ramanath, and the anonymous reviewers for many helpful comments on earlier versions of this paper.

REFERENCES

[1] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, "Path Sharing and Predicate evaluation for High-Performance XML Filtering," *TODS*, vol. 28, no. 4, pp. 467–516, 2003.

[2] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML streams with deterministic automata and stream indexes," *TODS*, vol. 29, no. 4, pp. 752–788, 2004.

[3] D. Olteanu, "SPEX: Streamed and Progressive Evaluation of XPath," *TKDE*, vol. 19, no. 7, 2007.

[4] X. Li and G. Agrawal, "Efficient Evaluation of XQuery over Streaming Data," in *Proc. VLDB*, 2005.

[5] A. Marian and J. Siméon, "Projecting XML Documents," in *Proc. VLDB*, 2003.

[6] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyen, "Type-Based XML Projection," in *Proc. VLDB*, 2006.

[7] M. Schmidt, S. Scherzinger, and C. Koch, "Combined Static and Dynamic Analysis for Effective Buffer Minimization in Streaming XQuery Evaluation," in *Proc. ICDE*, 2007.

[8] A. V. Aho, "Algorithms for finding patterns in strings." in *Handbook of Theoretical Computer Science, Volume A*, 1990, pp. 255–300.

[9] B. W. Watson and G. Zwaan, "A taxonomy of sublinear multiple keyword pattern matching algorithms," *Sci. Comput. Program.*, vol. 27, no. 2, pp. 85–118, 1996.

[10] D. E. Knuth, J. H. Morris (Jr.), and V. R. Pratt, "Fast Pattern Matching in Strings," *SIAM J. Computing*, vol. 6, no. 2, 1977.

[11] R. S. Boyer and J. S. Moore, "A Fast String Searching Algorithm," *Commun. ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[12] A. V. Aho and M. J. Corasick, "Efficient string matching: An aid to bibliographic search," *CACM*, vol. 18, no. 6, pp. 333–340, 1975.

[13] B. Commentz-Walter, "A String Matching Algorithm Fast on the Average," in *Proc. ICALP*, 1979.

[14] "Qizx/open," http://www.axyana.com/qizxopen/.

[15] "Saxon," http://saxon.sourceforge.net/.

[16] Apache XML Project, "Xalan-Java Version 2.0," 2000, http://xml.apache.org/xalan-j.

[17] "XMark," http://monetdb.cwi.nl/xml/.

[18] A. Berlea and H. Seidl, "Binary Queries for Document Trees," *Nordic J. of Computing*, vol. 11, no. 1, pp. 41–71, 2004.

[19] "xgrep," http://www.wohlberg.net/public/software/xml/xgrep/.

[20] J. Jaakkola and P. Kilpeläinen, "Nested text-region algebra," *TR C-1999-2, Dept. of Comp. Sci. Univ. of Helsinki*, 1999.

[21] M. Takeda et al., "Processing Text Files as Is: Pattern Matching over Compressed Texts, Multi-byte Character Texts, and Semi-structured Texts," in *Proc. SPIRE*, 2002.

[22] M. Altinel and M. Franklin, "Efficient Filtering of XML Documents for Selective Dissemination of Information," in *Proc. ICDE*, 2000.

[23] United States National Library of Medicine, 2006, http://www.nlm.nih.gov/bsd/sample_records_avail.html.

[24] A. Brüggemann-Klein and D. Wood, "One-Unambiguous Regular Languages," *Information and Computation*, vol. **142**, no. 2, pp. 182–206, 1998.

[25] J.-M. Champarnaud, "Subset Construction Complexity for Homogeneous Automata, Position Automata and ZPC-Structures," *Theor. Comput. Sci.*, vol. 267, no. 1-2, pp. 17–34, 2001.

[26] "http://www.cs.washington.edu/research/xmldatasets/," 2002.

[27] "Additional SMP Benchmark Results," http://www.informatik.uni-freiburg.de/~mschmidt/smp/.

[28] The Apache XML Project, "Xerces C++ XML Parser," http://xml.apache.org/xerces-c/.

[29] http://shootout.alioth.debian.org/.