Fast In-Memory XPath Search using Compressed Indexes

Diego Arroyuelo[1], Francisco Claude[2], Sebastian Maneth[3], Veli Mäkinen[4],

Gonzalo Navarro[5], Kim Nguyễn[6], Jouni Sirén[5] and Niko Välimäki[7]

[1]*Departamento de Informática, Universidad Técnica Federico Santa María, Chile*
[2]*Escuela de Informática y Telecomunicaciones, Universidad Diego Portales, Chile*
[3] *School of Informatics, University of Edinburgh, United Kingdom*
[4] *HIIT and Department of Computer Science, University of Helsinki, Finland*
[5]*Department of Computer Science, University of Chile, Chile*
[6]*LRI, Université Paris-Sud, France*
[7]*Department of Medical Genetics, Faculty of Medicine, University of Helsinki, Finland*

SUMMARY

XML documents consist of text data plus structured data (mark-up). XPath allows to query both, text and structure. Evaluating such hybrid queries is challenging. We present a system for in-memory evaluation of *XPath search queries*, that is, queries with text and structure predicates, yet without advanced features such as backward axes, arithmetics, and joins. We show that for this query fragment, which contains *Forward Core XPath*, our system, dubbed "SXSI", outperforms existing systems by 1–3 orders of magnitude. SXSI is based on state-of-the-art indexes for text and structure data. It combines two novelties. On one hand, it represents the XML data in a compact indexed form, which allows it to handle larger collections in main memory while supporting powerful search and navigation operations over the text and the structure. On the other, it features an execution engine that uses tree automata and cleverly chooses evaluation orders that leverage the speeds of the respective indexes. SXSI is modular and allows seamless replacement of its indexes. This is demonstrated through experiments with (1) a text index specialized for search of bio sequences, and (2) a word-based text index specialized for natural language search. Copyright © 0000 John Wiley & Sons, Ltd.

1. INTRODUCTION

As increasing amounts of data are stored, transmitted, queried, and manipulated in XML, the popularity of XPath and XQuery as query languages for semi-structured data grows. Evaluating such XML queries efficiently is challenging, and has triggered much research. Today there is a wealth of public and commercial XPath/XQuery engines, apart from several theoretical proposals. In this paper we focus on XPath, which is simpler and forms the basis of XQuery. XPath query engines can be roughly divided into two categories: *sequential* and *indexed*. In the former, which follows a *streaming* approach, no preprocessing of the XML data is performed. Each query sequentially reads the whole document, and the goal is to be as close as possible to making just one pass over the data, while using as little main memory as possible to hold intermediate results and data structures. Instead, the indexed approach preprocesses the XML document to build a data structure on it, so that queries can later be evaluated without traversing the whole document.

A serious shortcoming of the indexed approach is that the index can use much more space than the original data, and thus may have to be manipulated on disk even on moderate-sized collections

where the data itself would fit in main memory. Given the way disk performance favors sequential accesses, an index on disk may turn out to be slower than a streaming solution, even if the data is also stored on disk and if the index accesses only a minor part of the data. There are two approaches for dealing with this problem: (1) to load the index only partially (by using clever clustering techniques on disk), or (2) to use less powerful indexes that require less space. Examples of systems using these approaches are Qizx/DB [67], MonetDB/XQuery [11], and Tauro [64].

Most main memory XML query systems (such as Saxon [38], Galax [21], Qizx/Open [67], etc.) use machine pointers to represent XML data. We observe that on various well-established DOM implementations, this representation blows up memory consumption to about 5–10 times the size of the original XML data. As a result, they can only handle in main memory XML collections that are much smaller than what streaming approaches could acommodate (as these require no extra data apart from the plain XML).

In this work we aim at an index for XML that uses little extra space on top of that of the data (or actually less, as explained soon), yet without giving up on indexing power, but resorting instead to compact data structures. As a result, the index fits in main memory whenever the data does, thereby solving XPath queries without any need of resorting to disk. An in-memory index should outperform streaming approaches by far, even when the latter also operate in main memory. This is confirmed when comparing our indexed approach against two well-known streaming XPath engines (over data coming from a RAM-disk): GCX [40] and SPEX [59] are about 50 and 350 times, respectively, slower than our system.

An XML document can be regarded essentially as a *text collection* (that is, a set of strings) organized into a *tree structure*, so that the strings correspond to the text data and the tree structure corresponds to the nesting of tags. The problem of manipulating text collections and sequences within compressed space is now well understood [30, 25, 55], and also much work has been carried out on compact data structures for trees [36, 54, 28, 8, 2, 61]. In this paper we show how those types of compact data structures can be integrated into a compressed index representation for XML data, which is able to efficiently solve XPath queries.

A feature inherited from its components is that the compressed index *replaces* the XML collection, in the sense that the data (or any part of it) can be efficiently reproduced from the index (and thus the data itself can be discarded). The result is called a *self-index*, as the data is inextricably tied to its index. A self-index might thus require less space than the original data, while representing it and at the same time offering indexed access to it.

Ours is not the first self-index for XML data. The so-called XBW index [22, 23] is a self-index offering some XPath search support, yet this is reduced to a very limited class of queries that are handled particularly well: "simple paths", that is, queries of the form $//t_1/t_2/ \ldots /t_k$, where each t_i is a tag name. For such queries they can count the number of nodes satisfying the query in time $O(k)$, and can report them in time $O(\log^{1+\epsilon} n)$ per result, for any constant $\epsilon > 0$. For those specific queries, the XBW can be between one and two orders of magnitude faster than our system [50]. Likewise, there have been other attempts at using compact representations of the tree and the text. An earlier system in this line is BSBC [14]. They do not implement query evaluation on their compressed format, but their compression results are competitive. They show how some traversal operations used for XPath query evaluation can be sped up by using inverted indexes on the text content. Using inverted indexes, however, limits the applicability of the approach to compress natural language XML collections.

We aim at handling general sequences at the text nodes, and at a much more complete XPath coverage. Our system supports an extension of *Forward Core XPath* [31], that is, all forward navigational axes. The extension includes *text()* and the attribute axis, and the three text predicates = (equality), *contains*, and *starts-with*. We believe this to be a highly relevant and practical subset of XPath, and observe that a large fraction of real-world queries over datasets, such as *Medline* or *DBLP*, fall into this subset. Backward axes, arithmetics, and semi-joins, are not yet handled. Our system, dubbed *Succinct XML Self-Index* (SXSI), is the first practical and public tool for compressed indexing of XML data. It takes little space, solves a significant portion of

XPath, and largely outperforms the best public software supporting XPath we are aware of, namely MonetDB/XQuery [11] and Qizx/DB [67], in many cases by 1–3 orders of magnitude.

The main challenges in achieving our results have been to develop practical implementations of compact data structures (for texts, sequences, trees, and others) that are at a theoretical stage, to develop new compact schemes tailored to this particular problem, and to develop query processing strategies tuned for the specific cost model that emerges from the use of these compact data structures. The limitations of our scheme are that it is in-memory, that it is static (i.e., the index must be rebuilt when the XML data changes), and that it does not handle the more sophisticated parts of XPath nor XQuery. The first limitation is a design decision; the last two are subject of future work.

This paper introduces the three main ingredients of SXSI: (i) the text index, (ii) the tree index, and (iii) the query evaluator. While theoretical descriptions on the first two components can be found elsewhere in the literature, we mention here the main aspects of these components and focus on how they are integrated inside the practical SXSI system. For (iii) we have used tree automata, because they describe queries at a low level, thus allowing one to integrate the calls to our indexes. One important idea in integrating (ii) and (iii) is that of "jumping" to a descendant or following node (without traversing the intermediate nodes). This provides large speed-ups, and allows an "automata-optimal" evaluation [49]. Another important idea integrating (i) and (iii) is that of "true bottom-up runs": typically, a tree can only be accessed through its root node. In SXSI we are able to access leaves of the document tree. This allows us to start evaluation at text nodes, a technique that incurs large speed-ups for queries containing highly selective text predicates. Note that such queries are very common in practice.

The main part of the experimental section is about comparing SXSI against the state-of-the-art XPath engines MonetDB/XQuery and Qizx/DB. Certainly, this comparison must be considered with care: MonetDB and Qizx are full-blown XQuery engines with many features (such as multi-user and transaction support), while SXSI is a bare XPath engine. Hence, a comparison is hardly fair. However, our comparison shows the potential of succinct data structures and automata, as alternative core of XML databases. We use two batches of experiments: the "tree oriented" queries of the XPathMark benchmark [26] (over XMark data [62]) and our own "text oriented" queries (over Medline documents). Our results show that SXSI outperforms the other systems for virtually all tested queries, in many cases by 1–3 orders of magnitude, and moreover that the running times of SXSI are more predictable and "robust" than those of other systems. We also demonstrate SXSI's ability to seamlessly integrate other indexes. We replace the text index by (1) an index tailored towards bio sequence search and (2) a word-based index tailord towards natural language search.

We extend the work of [4] by (i) detailed explanations and examples, (ii) a description of important general optimizations used to efficiently run automata, (iii) a raw speed comparison of the indexes against naive solutions, (iv) an experimental analysis of the impact of different optimization techniques, and (v) an experimental comparison with alternative text indexes.

2. BASIC CONCEPTS AND MODEL

We regard an XML document as (i) an ordered set of strings and (ii) a labeled tree. The latter is the natural XML parse tree defined by the hierarchical tags, where the (normalized) tag name labels the corresponding node. We add an extra root node (labeled "&") on top of the document's root node; this node is needed for XPath semantics, but could also be used to hold additional information such as the document name. Each text node is represented as a leaf labeled #. Attributes are handled as follows in this model. Each node with attributes gets an additional single child labeled @ (at the first child position), and for each attribute @attr=value of the node, a child labeled attr is added to its @-node, and a leaf child labeled % to the attr-node. The text content value is then associated to that leaf. Thus, there is exactly one string content associated to each tree leaf labeled # or %. We refer to those strings as *texts*. We do not store empty texts; for instance, the XML document <a> is stored as a single a-labeled leaf node (which is the unique child of the &-labeled root node).

Table I. Notation.

Term	Meaning
T	Concatenation of all the texts in the collection
Σ	Alphabet of the distinct text symbols
$\$$	Character that terminates each text in the collection
n	Number of nodes in the XML tree
t	Number of different tag and attribute names in the document
d	Number of texts in the XML tree (in our model, tree leaves)
$H_k(S)$	k-th order empirical entropy of string S

Let us call T the concatenation of all the texts, each separated by a symbol "$\$$" smaller than any other. Let n the total number of tree nodes, Σ the alphabet of the strings, t the total number of different tag and attribute names, and d the number of texts (that is, # or %-labeled tree leaves). These receive *text identifiers* which are consecutive numbers assigned in a left-to-right parsing of the data. In our implementation Σ is simply the set of byte values 1 to 255, and 0 will act as a special terminator called $\$$. This symbol occurs exactly once at the end of each text in T. Note that our implementation can easily support also UTF-8 encoding and hence adheres to the XML standard. Table I summarizes the notation.

To connect tree nodes and texts, we define *global identifiers*, which give unique numbers to both internal and leaf nodes, in depth-first preorder. Figure 1 shows a toy document (top left) and our model of it (top right), as well as its representation using our data structures (bottom), which serves as a running example for the rest of the paper. In the model, the tree is formed by the solid edges, whereas dotted edges display the connection with the set of texts. The tree contains the extra root node (labeled &), as well as extra internal nodes (labeled #, @, and %). Note how the attributes are handled. There are six texts, which are associated to the tree leaves and receive consecutive text numbers (marked in italics at their right). Global identifiers are associated to each node and leaf (drawn at their left). The conversion between tag names and symbols, drawn within the bottom-left component, is used to translate queries and to recreate the XML data. Note that if the return and space (indentation) characters are present precisely as shown in the "XML data" box of the figure, then there are indeed several additional #-leaves in the tree: for instance, the whitespace (return and space characters) after the initial <parts> and before the final </parts> give rise to two extra texts (and therefore the parts-node in the tree has additional first and last children labeled #). In total there are seven such whitespace texts which have been omitted in our figure for reasons of readability.

Some notation and measures of compressibility follow, preceding a rough description of our space complexities. The *empirical k-th order entropy* [51] of a sequence S over alphabet Σ, $H_k(S) \le \log |\Sigma|$, is a lower bound to the output size per symbol of any k-th order compressor applied to S. The formula of the zero-order entropy is as follows:

$$H_0(S) = \sum_{c \in \Sigma} \frac{s_c}{s} \log \frac{s}{s_c},$$

where s_c is the number of occurrences of c in S and $s = |S|$. We assume $\log = \log_2$ and $0 \log 0 = 0$ henceforth. Let Σ^k denote the set of words over Σ of length k. Now let S_W be the set of characters preceding the occurrences of $W \in \Sigma^k$ in S, then for $k > 0$,

$$H_k(S) = \frac{1}{s} \sum_{W \in \Sigma^k} |S_W| H_0(S_W).$$

Note $0 \le H_k(S) \le H_{k-1}(S) \le \ldots \le H_0(S) \le \log |\Sigma|$.

We will build on self-indexes able of handling text collections T within $|T|H_k(T) + o(|T| \log |\Sigma|)$ bits [55, 25, 47]. On the other hand, representing an unlabeled tree of n nodes requires $2n - O(\log n)$ bits, and several representations using $2n + o(n)$ bits support many tree query and navigation operations in constant time (e.g., [61]). The labels require in principle other $n \log t$ bits.
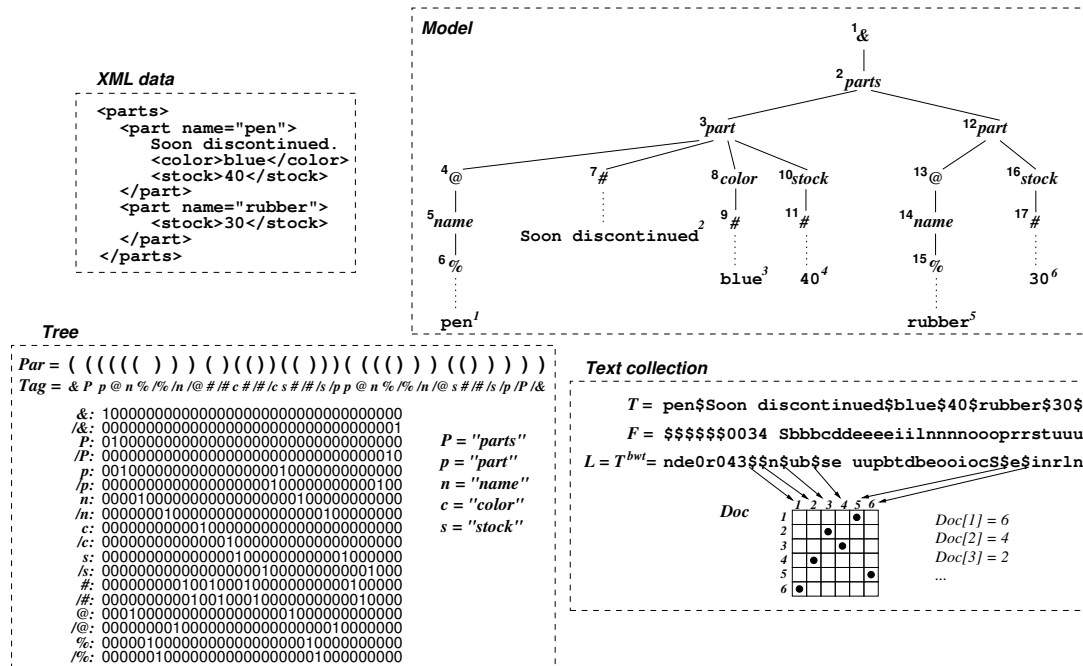
XML data
```
<parts>
  <part name="pen">
      Soon discontinued.
      <color>blue</color>
      <stock>40</stock>
  </part>
  <part name="rubber">
      <stock>30</stock>
  </part>
</parts>
```

Model

^{1}& — ^{2}parts — ^{3}part, ^{12}part

^{3}part: ^{4}@, ^{7}#, ^{8}color, ^{10}stock
^{4}@ — ^{5}name — ^{6}% — pen^1
^{7}# — Soon discontinued^2
^{8}color — ^{9}# — blue^3
^{10}stock — ^{11}# — 40^4

^{12}part: ^{13}@, ^{16}stock
^{13}@ — ^{14}name — ^{15}% — rubber^5
^{16}stock — ^{17}# — 30^6

Tree

Par = (((((( ))) ( )(())(( )))(((( ))) (( ))))
Tag = & P p @ n % /% /n /@ # /# c # /# /c s # /# /s /p p @ n % /% /n /@ s # /# /s /p /P /&

```
 &:  10000000000000000000000000000000000
/&:  00000000000000000000000000000000001
 P:  01000000000000000000000000000000000
/P:  00000000000000000000000000000000010
 p:  00100000000000000010000000000000000
/p:  00000000000000000100000000000000100
 n:  00001000000000000000100000000000000
/n:  00000010000000000000000100000000000
 c:  00000000001000000000000000000000000
/c:  00000000000010000000000000000000000
 s:  00000000000001000000000010000000000
/s:  00000000000000010000000001000000000
 #:  00000010001001000010000000010000100
/#:  00000000010010010000000000010000010
 @:  00010000000000000001000000000000000
/@:  00000001000000000000010000000000000
 %:  00000100000000000000000010000000000
/%:  00000010000000000000000001000000000
```

P = "parts"
p = "part"
n = "name"
c = "color"
s = "stock"

Text collection

T = pen$Soon discontinued$blue40rubber30
F = $$$$$$0034 Sbbbcddeeeeiilnnnnoooprrstuuu
$L = T^{bwt}$ = nde0r043$$nubse uupbtdbeooiocS$e$inrln

Doc

Doc[1] = 6
Doc[2] = 4
Doc[3] = 2
...

Figure 1. Our running example on representing an XML document.

Sequences S of length n over an alphabet of size t can be stored within $n \log t(1 + o(1))$ bits (and even $nH_0(S) + o(n \log t)$), so that any element $S[i]$ can be accessed, and they can also efficiently answer the following queries [33, 30, 25, 6]:

$rank_c(S, i)$ is the number of c's in $S[1, i]$;

$select_c(S, j)$ is the position of the j-th c in S.

These are essential building blocks for more complex functionalities, as seen later.

The final space requirement of our index will include:

1. $|T|H_k(T) + o(|T| \log |\Sigma|)$ bits for representing the text collection T in self-indexed form. This supports the string searches of XPath and can (slowly) reproduce any text.
2. $d \log d + o(d \log d)$ bits for the mapping between the self-index and the text identifiers, e.g., to determine to which text identifier a self-index position belongs, or restricting self-index searches to some texts.
3. $2n + o(n)$ bits for representing the tree structure. This supports many navigational operations in constant time.
4. $4n \log t + 2n + o(n)$ bits to represent the tags in a way that they support very fast XPath searches.
5. $2n + o(n)$ bits for mapping between tree nodes and text identifiers.
6. Optionally, $|T| \log |\Sigma|$ or $|T|H_k(T) + o(|T| \log |\Sigma|)$ bits, plus $O(d \log \frac{|T|}{d})$, to achieve faster text extraction than in 1).

As a practical yardstick, without the extra storage of texts (Item 6) the memory consumption of our system is about the size of the original XML file (and, being a self-index, includes it!), and with the extra text store the memory consumption is 1–2 times the size of the original XML file.

In Section 3 we describe our representation of the set of strings, including how to obtain text identifiers from text positions. This explains items 1, 2, and 6 above. Section 4 describes our representation for the tree and the labels, and the way the correspondence between tree nodes and

text identifiers works. This explains items 3, 4, and 5. Section 5 describes how we process XPath queries on top of these compact data structures. In Section 6 we give some implementation details and empirically compare our SXSI engine with the most relevant public engines we are aware of. We conclude in Section 7.

3. TEXT REPRESENTATION

Text data in SXSI is represented as a succinct full-text self-index [55] that is generally known as the *FM-index* [24]. The index supports efficient pattern matching operations that can be easily extended to support different XPath predicates.

3.1. FM-Index and Backward Searching

Given a string T of total length $|T|$, from an alphabet Σ, the *alphabet-friendly FM-index* [25] requires $|T|H_k(T) + o(|T|\log|\Sigma|)$ bits of space for any $k \leq \alpha\log_{|\Sigma|} n$ and any constant $0 < \alpha < 1$. The index supports counting the number of occurrences of a pattern P in $O(|P|\log|\Sigma|)$ time. Locating the occurrences takes extra $O(\log^{1+\epsilon}|T|)$ time per answer, for any constant $\epsilon > 0$.

The FM-index is based on the Burrows–Wheeler transform (BWT) of string T [15]. Assume T ends with the special end-marker \$. Let $\mathcal{M}$ be a matrix whose rows are all the cyclic rotations of T in lexicographic order. The first column of $\mathcal{M}$, denoted F, contains all symbols of T in lexicographic order. The last column L of $\mathcal{M}$ forms a permutation of T which is the BWT string T^{bwt}. The matrix is only conceptual; the FM-index uses only on the T^{bwt} string. Figure 2 illustrates the matrix $\mathcal{M}$ with its first and last rows (F and T^{bwt}) in bold. Figure 1 (bottom right) shows how this fits in our overall scheme.

The resulting permutation from T to T^{bwt} is reversible. There exists a simple last-to-first mapping from symbols in T^{bwt} to F [24]: Let $C[c]$ be the total number of symbols in T that are lexicographically less than c. Then the *LF-mapping* is defined as

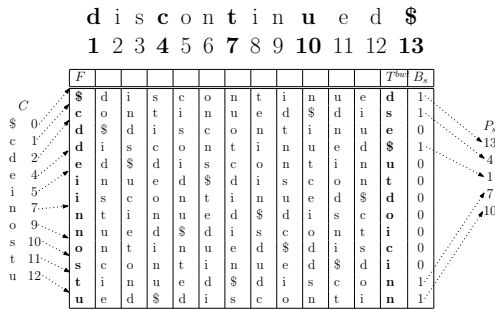$$LF(i) = C[T^{bwt}[i]] + rank_{T^{bwt}[i]}(T^{bwt}, i).$$

Note that $T^{bwt}[i]$ is the symbol preceding the i-th lexicographically smallest row of $\mathcal{M}$. Thus, if $T^{bwt}[i] = T[j]$, then $T^{bwt}[LF(i)] = T[j-1]$. The symbols of T can therefore be read in reverse order by starting from the location i such that $T^{bwt}[i] = \$$, and applying LF recursively:

$$T[|T|] = \$ = T^{bwt}[1] \qquad T[|T| - 1] = T^{bwt}[LF(1)] \qquad T[|T| - 2] = T^{bwt}[LF(LF(1))]$$

and so on until, after $|T|$ steps, we get the first symbol $T[1]$. The values $C[c]$ can be stored in a small array of $|\Sigma|\log|T|$ bits. Function $rank_c(T^{bwt}, i)$ can be computed in $O(\log|\Sigma|)$ time with a data structure called *wavelet tree* which, when built on T^{bwt}, uses only $|T|H_k(T) + o(|T|\log|\Sigma|)$ bits [33, 25, 45]. In practice we opt for a Huffman-shaped wavelet tree using uncompressed bitmaps inside [17]. Despite this achieves space $|T|(H_0(T) + 1)(1 + o(1))$, it is much faster than the other implementations. In particular, operations cost $O(H_0(T))$ time on average under some conditions, an improvement that applies to all the $O(\log|\Sigma|)$ worst-case complexities that follow.

Pattern matching is supported via *backward searching* on the BWT [24]. Given a pattern $P[1, m]$, the backward search starts with the range $[sp, ep] = [1, |T|]$ of rows in $\mathcal{M}$. At each step $i \in \{m, m - 1, \ldots, 1\}$ of the backward search, the range $[sp, ep]$ is updated to match all rows of $\mathcal{M}$ that have $P[i, m]$ as a prefix. The new range $[sp', ep']$ is given by $sp' = C[P[i]] + rank_{P[i]}(T^{bwt}, sp - 1) + 1$ and $ep' = C[P[i]] + rank_{P[i]}(T^{bwt}, ep)$. Each step takes $O(\log|\Sigma|)$ time using the wavelet tree, and finally $ep - sp + 1$ gives the number of times P occurs in T. Figure 2 gives the pseudocode.

To find out the location of each occurrence, the text is traversed backwards from each $sp \leq i \leq ep$ (virtually, using LF on T^{bwt}) until a *sampled* position is found. This is a sampling carried out at regular text positions, so that the corresponding positions in T^{bwt} are marked in a bitmap $B_s[1, |T|]$, and the text position corresponding to $T^{bwt}[i]$, if $B_s[i] = 1$, is stored in a samples array $P_s[rank_1(B_s, i)]$. If every l-th position of T is sampled, the extra space is $O((n/l)\log n) + o(n)$

d	i	s	c	o	n	t	i	n	u	e	d	$
1	2	3	**4**	5	6	**7**	8	9	**10**	11	12	**13**

C array:

$	0
c	1
d	2
e	4
i	5
n	7
o	9
s	10
t	11
u	12

Matrix (F ... T^{bwt} B_s):

F												T^{bwt}	B_s	P_s
$	d	i	s	c	o	n	t	i	n	u	e	d	1	13
c	o	n	t	i	n	u	e	d	$	d	i	s	1	4
d	$	d	i	s	c	o	n	t	i	n	u	e	0	
d	i	s	c	o	n	t	i	n	u	e	d	$	1	1
e	d	$	d	i	s	c	o	n	t	i	n	u	0	
i	n	u	e	d	$	d	i	s	c	o	n	t	0	
i	s	c	o	n	t	i	n	u	e	d	$	d	0	
n	t	i	n	u	e	d	$	d	i	s	c	o	0	
n	u	e	d	$	d	i	s	c	o	n	t	i	0	7
o	n	t	i	n	u	e	d	$	d	i	s	c	0	
s	c	o	n	t	i	n	u	e	d	$	d	i	0	10
t	i	n	u	e	d	$	d	i	s	c	o	n	1	
u	e	d	$	d	i	s	c	o	n	t	i	n	1	

FM-Count($p_1 p_2 \ldots p_m$)
1. $\quad i := m$
2. $\quad sp := 1$
3. $\quad ep := |T|$
4. $\quad$**While** $sp \leq ep$ and $i \geq 1$ **Do**
5. $\qquad c := p_i$
6. $\qquad sp := C[c] + rank_c(T^{bwt}, sp - 1) + 1$
7. $\qquad ep := C[c] + rank_c(T^{bwt}, ep)$
8. $\qquad i := i - 1$
9. $\quad$**If** $ep < sp$ **Then**
10. $\qquad$**Return** 0
11. $\quad$**Else**
12. $\qquad$**Return** $ep - sp + 1$

Figure 2. On the left, an example of the FM-index for text $T =$ "`discontinued`" sampled each $l = 3$ positions. On the right, counting algorithm on the FM-index.

(including the compressed B_s [60]) and the locating takes $O(l \log |\Sigma|)$ time per occurrence. Using $l = \Theta(\log^{1+\epsilon} |T| / \log |\Sigma|)$ for any $\epsilon > 0$ yields $o(|T| \log |\Sigma|)$ extra space and locating time $O(\log^{1+\epsilon} |T|)$.

Figure 2 illustrates a sampling of T each $l = 3$ symbols. Assume we look for $P =$ "`n`"; then backward search finds $[sp, ep] = [8, 9]$. Now to locate the occurrence at 8 we see that $B_s[8] = 0$, $B_s[LF(8)] = B_s[10] = 0$, and finally $B_s[LF(10)] = B_s[2] = 1$. This corresponds to position $P_s[rank_1(B_s, 2)] = P_s[2] = 4$. Since we applied LF twice, the answer is $4 + 2 = 6$. We have found the occurrence $T[6..] =$ "`n..`".

3.2. Text Collection and Queries

The textual content of the XML data is stored as $-terminated strings so that each text corresponds to one string. Let T be the concatenated sequence of the d texts. Array P_s is extended to record both the text identifier and the offset inside it. Since there are several $'s in T, we fix a special ordering such that the end-marker of the i-th text appears at $F[i]$ in $\mathcal{M}$ (see Figure 1, bottom right). This generates a valid T^{bwt} of all the texts and makes it easy to extract the i-th text starting from its $-terminator.

Now T^{bwt} contains all end-markers in some permuted order. This permutation is represented with a data structure Doc, that maps from positions of $s in T^{bwt} to text identifiers. Let $T^{bwt}[j]$ correspond to the first symbol of the text with identifier x, thus if $i = LF(j)$ it holds $T^{bwt}[i] = $. Then we store $Doc[rank_\$(T^{bwt}, i)] = x$. Furthermore, Doc can be stored in a format that allows for range searching (as illustrated in Figure 1 (right)): Given a range $[sp, ep]$ of T^{bwt} and a range of text identifiers $[x, y]$, Doc can be used to output identifiers of all $-terminators within the range $[sp, ep] \times [x, y]$, in $O(\log d)$ time per answer [46]. In practice, because we only use the simpler functionality in the current system, Doc is implemented as a plain array using $d \log d$ bits.

Note Doc allows us to never switch from one text to another while looking for the preceding sampled value: If we reach a $ before finding any $B_s[i] = 1$, array Doc can be used to determine that we are at the first position of some text with identifier x.

The basic pattern matching feature of the FM-index can be extended to support XPath functions such as *starts-with*, *ends-with*, *contains*, and operators $=, \leq, <, >, \geq$ for lexicographic ordering. Given a pattern and a range of text identifiers to be searched, these functions return all text identifiers that match the query within the range. In addition, existential (is there a match in the range?) and counting (how many matches in the range?) queries are supported. Time complexities are $O(|P| \log |\Sigma|)$ for the search phase, plus an extra for reporting. While we describe the operators in their general form, which needs the range reporting functionality from Doc, our current prototype implements only the simple case $[x, y] = [1, d]$, where Doc can be an array.

starts-with$(P, [x, y])$: The goal is to find texts in range $[x, y]$ prefixed by the given pattern P. After the normal backward search, the range $[sp, ep]$ in T^{bwt} contains the end-markers of all the texts prefixed by P. Now $[sp, ep] \times [x, y]$ can be mapped to *Doc*, and existential and counting queries can be answered in $O(\log |\Sigma| + \log d)$ time. Matching text identifiers can be reported in $O(\log d)$ time per identifier. If $[x, y] = [1, d]$ and *Doc* is an array, the counting time after the backward search is $O(\log |\Sigma|)$, and each text identifier can be reported in constant time.

ends-with$(P, [x, y])$: Backward searching is localized to texts in $[x, y]$ by choosing $[sp, ep] = [x, y]$ as the starting interval, since we have forced the ordering of $F[1, d]$ so that $F[z] = \$$ is the terminator of text with identifier z. After the backward search, the resulting range $[sp, ep]$ contains all possible matches, thus existential and counting queries are answered in constant time after the search. To find out text identifiers for each occurrence, the text must be traversed backwards to find a sampled position (or a \$). The cost is $O(l \log |\Sigma| + \log d)$ per answer, where l is the sampling step. If $[x, y] = [1, d]$ and *Doc* is an array, the cost is just $O(l \log |\Sigma|)$.

operator $= (P, [x, y])$: Whole texts which are equal to P, and with identifiers in the range $[x, y]$, can be found as follows. Start with a backward search as in *ends-with*, and then map to the \$-terminators as in *starts-with*. The time complexities are same as in *starts-with*.

contains$(P, [x, y])$: To find texts that contain P, we start with the normal backward search and finish like in *ends-with*. In this case there might be several occurrences inside one text, which have to be filtered. Thus, the time complexity is proportional to the total number of occurrences, $O(l \log |\Sigma|)$ for each. Existential and counting queries are as slow as reporting queries. The basic $O(|P| \log |\Sigma|)$-time counting of all the occurrences of P can still be useful for query optimization.

operators $\leq, <, >, \geq$: Operator $\leq$ matches texts that are lexicographically smaller than or equal to the given pattern. It can be solved like the *starts-with* query, but updating only the ep of each backward search step, while $sp = 1$ stays constant. While $[sp, ep]$ delimits the rows of $\mathcal{M}$ that start with $P[i, m]$, $[1, ep]$ delimits the rows that start with a prefix lexicographically smaller than or equal to $P[i, m]$. If at some point there are no occurrences of $P[i] = c$ within the prefix $T^{bwt}[1, ep]$, this means that $P[i, m]$ does not appear in T. To continue the search we replace $ep = C[c]$ and continue for $P[1, i - 1]$. Other operators can be supported analogously, and costs are as for *starts-with*.

The new XPath extension, *XPath Full Text 1.0* [68], suggests a wider functionality for text searching. Implementation of these extensions requires regular expression and approximate searching functionalities, which can be supported within our index using the general *backtracking framework* [41]: The idea is to alter the backward search to branch recursively to different ranges $[sp', ep']$ representing the suffixes of the text prefixes (i.e., substrings). This is done by computing $sp'_c = C[c] + rank_c(T^{bwt}, sp - 1) + 1$ and $ep'_c = C[c] + rank_c(T^{bwt}, ep)$ for all $c \in \Sigma$ at each step and recursing on each $[sp'_c, ep'_c]$. Then the pattern (or regular expression) can be compared with all substrings of the texts, allowing us to search for approximate occurrences [41]. The running time becomes exponential in the number of errors allowed, but different branch-and-bound techniques can be used to obtain practical running times [42, 43]. We omit further details, as these extensions are out of the scope of this paper.

3.3. Construction and Text Extraction

The FM-index can be built by adapting any BWT construction algorithm. Linear time algorithms exist for the task, but their practical bottleneck is the peak memory consumption. Although there exist general time- and space-efficient construction algorithms, it turned out that our special case of text collection admits a tailored incremental BWT construction algorithm [65] (see the references and experimental comparison therein for previous work on BWT construction): The text collection is split into several smaller collections, and a temporary index is built for each of them separately.

The temporary indexes are then merged, and finally converted into a static FM-index. The BWT allows extracting the i-th text by successively applying *LF* from $T^{bwt}[i]$, at $O(\log |\Sigma|)$ cost per extracted symbol.

3.4. Faster Text Extraction using More Space

To enable faster text extraction, we allow storing the texts in plain format in $n \log |\Sigma|$ bits, or in an enhanced LZ78-compressed format (derived from the LZ-index [5]) using $|T|H_k(T) + o(|T| \log |\Sigma|)$ bits. These secondary text representations are coupled with a delta-encoded bit vector storing starting positions of each text in T. This bitmap requires $O(d \log \frac{|T|}{d})$ more bits.

In fact, keeping next to the FM-index an additional copy of all texts in plain format has more advantages. As mentioned before, the time complexity of contains-queries is proportional to the total number of occurrences. This implies that for large occurrence numbers, it becomes faster to search over the plain texts than over the FM-index. The precise cut-off point depends on the sampling factor l, see Section 6.3 for more details. Since a global count over the FM-index is fast ($O(|P| \log |\Sigma|)$ time), we use it to decide whether to search over the plain text or over the FM-index.

In practice we opt for a plain text representation, which is much faster for extraction than an LZ-index at the price of not much more space.

4. TREE REPRESENTATION

4.1. Data Representation

The tree structure of an XML collection is represented by the following compact data structures, which provide navigation and indexed access to it. See also the bottom left of Figure 1.

4.1.1. Par This is the *balanced parentheses* representation of the tree structure (see, e.g., [53]). It is obtained by traversing the tree in *depth-first-search* (DFS) order (or pre-order), writing a "(" whenever we arrive at a node, and a ")" when we leave it (thus it follows the sequences of events generated by an XML SAX parser). In this way, every node is represented by a pair of matching opening and closing parentheses. A tree node is identified by the position of its opening parenthesis in *Par* (that is, a node is just an integer index within *Par*). In particular, we use the balanced parentheses implementation of [61], which supports a very complete set of operations, including finding the i-th child of a node, in constant time; for more information concerning implementation details and performance, see [3]. Overall *Par* uses $2n + o(n)$ bits. This includes the space needed for constant-time binary *rank* on *Par*, which is very fast in practice.

4.1.2. Tag This is the sequence of the tag identifiers of each tree node, including an opening and a closing version of each tag, to mark the beginning and ending point of each node. These tags are numbers in $[1, 2t]$ and are aligned with *Par* so that the tag of node i is simply $Tag[i]$.

We also need *rank* and *select* queries on *Tag*. They allow us to carry out special operations such as "TaggedDesc" which "jumps" to the first descendant of the given node having a given label (see Section 4.2.2). Several sequence representations supporting access and these operations are known [33, 30, 17]. Given that *Tag* is not too critical in the overall space, but it is in time, we opt for a practical representation that favors speed over space. First, we store the tags in an array using $\lceil \log 2t \rceil$ bits per field, which gives constant time access to $Tag[i]$. The rank and select queries over the sequence of tags are answered by a second structure. Consider the binary matrix $R[1..2t][1..2n]$ such that $R[i, j] = 1$ if $Tag[j] = i$. We represent each row of the matrix using Okanohara and Sadakane's structure `sarray` [58]. Its space requirement for each row i is $n_i \log \frac{2n}{n_i} + n_i(2 + o(1))$ bits, where n_i is the number of times symbol i appears in *Tag*. The total space of both structures adds up

to $2n \log(2t) + 2nH_0(\mathit{Tag}) + n(2 + o(1)) \le 4n \log t + 2n + o(n)$ bits. Thus we support access and *select* in $O(1)$ time, and *rank* in $O(\log n)$ time.*

4.2. Tree Navigation

We define the following operations over the tree structure, which are useful to support XPath queries over the tree. Most of these operations are supported in constant time, except when a *rank* over *Tag* is involved. In what follows, we assume that all the operations take an implicit argument *Tree* (we do not write it explicitly to improve the readability). Nodes of *Tree* (that is, positions in *Par*) are ranged over by x, y, and so on. We assume the existence of a *dummy* node `Nil` guaranteed to be distinct from any node of *Tree* (for instance -1).

4.2.1. Basic Tree Operations These are directly inherited from Sadakane's implementation [61]. We mention only the most important ones for this paper.

- Close(x): The closing parenthesis matching *Par*[x]. If x is a small subtree this takes a few local accesses to *Par*, otherwise a few non-local table accesses.
- Preorder(x) = $rank_{(}(\mathit{Par}, i)$: Preorder number of x.
- SubtreeSize(x) = (Close(x) $- x + 1)/2$: Number of nodes in the subtree rooted at x.
- IsAncestor(x, y) = $x \le y \le$ Close(x): Whether x is an ancestor of y.
- IsLeaf(x) = (*Par*[$x + 1$] = ")"): Whether node x is a leaf in the tree.
- FirstChild(x) = $x + 1$: First child of x, if any (*i.e.*, if *Par*[$x + 1$] = "("), `Nil` otherwise (*i.e.* if x denotes a leaf).
- NextSibling(x) = Close(x) $+ 1$: Next sibling of x, if any (i.e., if *Par*[Close(x) $+ 1$] = "("), `Nil` otherwise.
- Parent(x): Parent of x, found as the closest parentheses pair enclosing x. Somewhat costlier than Close(x) in practice, because the answer is less likely to be near x in *Par*. Return `Nil` for the root node.

4.2.2. Connecting to Tags The following operations are essential for our fast XPath evaluation. Let tag be a tag identifier.

- SubtreeTags(x, tag): Returns the number of occurrences of *tag* within the subtree rooted at node x. This is $rank_{tag}(\mathit{Tag}, \text{Close}(x)) - rank_{tag}(\mathit{Tag}, x - 1)$.
- Tag(x): Gives the tag identifier of node x. In our representation this is just *Tag*[x]. Returns `Nil` if there are no such nodes.
- TaggedDesc(x, tag): The first node (in pre-order) labeled *tag* strictly within the subtree rooted at x. It is obtained as $select_{tag}(\mathit{Tag}, rank_{tag}(\mathit{Tag}, x) + 1)$ if it is $\le$ Close(x), otherwise there is no such node and the function returns `Nil`.
- TaggedPrec(x, tag): The last node labeled *tag* with preorder smaller than that of node x, and not an ancestor of x. Let $r = rank_{tag}(\mathit{Tag}, x - 1)$. If $select_{tag}(\mathit{Tag}, r)$ is not an ancestor of node x, we return it. Otherwise, we set $r = r - 1$ and iterate. Returns `Nil` when $r = 0$.
- TaggedFoll(x, tag): The first node labeled *tag* with preorder larger than that of x, and not in the subtree of x. This is $select_{tag}(\mathit{Tag}, rank_{tag}(\mathit{Tag}, \text{Close}(x)) + 1)$. Return `Nil` if there is no such node.

4.2.3. Connecting the Text and the Tree Conversion between text numbers, tree nodes, and global identifiers, is easily carried out by using *Par* and a bitmap B of $2n$ bits that marks the opening parentheses of tree leaves containing text, plus $o(n)$ extra bits to support rank/select queries. The bitmap B uses an implementation of [60] which is described in [17], and it enables the computation of the following operations:

---

*They report higher complexities, but these are easily improved by using a representation for dense arrays that supports *select* in constant time.

- LeafNumber(x): Gives the number of leaves up to x in *Par*. This is $rank_1(B, x)$.
- TextIds(x): Gives the range of text identifiers that descend from node x. This is simply [LeafNumber($x-1$) $+ 1$, LeafNumber(Close(x))].
- XMLIdText(d): Gives the global tree preorder identifier for the text with identifier d. This is Preorder($select_1(B, d)$).
- XMLIdNode(x): Gives the global identifier for a tree node x. This is just Preorder(x).

4.3. Displaying Contents

Given a node x, we want to recreate its XML serialization, that is, return (a portion of) the original XML string. We traverse the structure starting from *Par*[x], retrieving the tag names and the text contents, from the text identifiers. The time is $O(\log \sigma)$ per text symbol (or $O(1)$ if we use the redundant text storage described in Section 3) and $O(1)$ per tag.

- GetText(d): Generates the text with identifier d.
- GetSubtree(x): Generates the subtree at node x.

5. XPATH QUERIES

In this section we define the XPath fragment "Core+", show its translation into automata, and discuss efficient execution of these automata. We do not formally define the semantics of XPath and assume the reader to be familiar with the basics of XPath, see, e.g., [31].

5.1. The XPath Fragment Core+

Our goal is to support a practical subset of XPath, while being able to guarantee efficient evaluation based on the data structures described in the previous sections. As a first shot we target the forward fragment of "Core XPath" [31]. Here is an EBNF for Core XPath.

Core	::=	LocationPath	/ LocationPath
LocationPath	::=	LocationStep (/ LocationStep)*
LocationStep	::=	Axis :: NodeTest	Axis :: NodeTest [Pred]
Pred	::=	Pred and Pred	Pred or Pred	not (Pred)	Core	(Pred)

We focus our presentation on the `descendant` and `child` axes, but `self`, `attribute` and `following-sibling` are also supported in our implementation. A node test is either the wild-card (`*`), a tag name, or a node type test, that is, one of "`text()`" or "`node()`".

Our fragment, called *Core+*, supports forward Core XPath and additionally all text predicates of XPath 1.0, that is, the `=` (equality), `contains`, `starts-with` and `ends-with` predicates. These predicates appear inside filters (square brackets) and are generated by the non-terminal "Pred". Equality tests if a (constant) string is equal to a string selected by a *Core+* expression; `contains` tests if the string is contained in the expression; and `starts-with` tests if the string is a prefix of the expression. Thus our Core+ fragment specializes the above EBNF with the following rules.

Axis	::=	descendant	child	self	attribute	following-sibling
NodeTest	::=	*	TagName	text()	node()
Pred	::=	Core+ = String	contains(Core+, String)	starts-with(Core+, String)
			ends-with(Core+, String)

An XPath query selects nodes of an XML document. The last axis in a query determines the selected nodes. For instance, the query `/descendant::listitem/child::keyword` selects all `keyword`-children of all `listitem`-nodes in the document. The query $T = $ `/descendant::text()` selects all text nodes of the document, and the query $A = $ `/descendant::*/attribute::*` selects

$$
\begin{array}{r|lcl}
1 & q_0, \{\&\} & \rightarrow & \downarrow_1 q_1 \\
2 & q_1, \{\text{listitem}\} & \rightarrow & \downarrow_1 q_1 \wedge \downarrow_1 q_2 \wedge \downarrow_2 q_1 \\
3 & q_1, \mathcal{L} & \rightarrow & \downarrow_1 q_1 \wedge \downarrow_2 q_1 \\
4 & q_2, \{\text{keyword}\} & \rightarrow & \text{mark} \wedge \downarrow_1 q_2 \wedge \downarrow_1 q_3 \wedge \downarrow_2 q_2 \\
5 & q_2, \mathcal{L} & \rightarrow & \downarrow_1 q_2 \wedge \downarrow_2 q_2 \\
6 & q_3, \{\text{emph}\} & \rightarrow & \top \\
7 & q_3, \mathcal{L} & \rightarrow & \downarrow_2 q_3
\end{array}
$$

Figure 3. The tree automaton for the query `/descendant::listitem/descendant::keyword[child::emph]`

all attribute nodes of the document. In terms of our example document in Figure 1, the query T selects the nodes 7, 9, 11, and 17 of our model. XPath processors return the XML content (subtrees) of the selected nodes. Thus, for T the strings "Soon discontinued", "blue", "40", and "30" are returned (plus the whitespace text nodes, cf. the discussion at the beginning of Section 2).

The string held by every text and attribute node is called its *value*. If a subquery only selects text and attribute nodes, then we call it a *value expression*. The "Core+" non-terminals mentioned on the right hand side of our last "Pred" rule must correspond to value expressions (we have avoided to complicate the grammar to enforce that). As an example, the query

```
/descendant::text()[contains(self::node(),'and')]
```

is in Core+; note that "self" here is a value expression, because `/descendant::text()` is.

5.2. From XPath to Automata

We use "marking tree automata" as our execution model for Core+ queries. Before we formally define these automata in the next section, we explain here by means of an example how a Core+ query is translated into an automaton. The translation of an XPath query to an automaton is a simple syntax-directed translation that can be carried out in one pass through the parse tree of the query. Roughly speaking, the resulting automaton is "isomorphic" to the original query. Consider the query

```
/descendant::listitem/descendant::keyword[child::emph] .
```

This query selects all `keyword`-nodes that are descendants of `listitem`-nodes, and that have a child node tagged `emph`. Formally speaking, the query starts with the expression "/" which selects the root node of the document (in our model, the &-node), then applies the descendant axis to that, and so on. The automaton has four states, q_0, q_1, q_2, q_3 which correspond to the four steps in the query (namely, "/", two descendants, and the child one). Its transitions are given in Figure 3 The automaton has start state q_0, and end states q_1, q_2. At first we can consider that this automaton is a classical *non-deterministic* alternating tree automaton (e.g. as the ones documented in [18] and [35]). The transitions have the form: "*state, set of labels $\rightarrow$ formula*" In the transitions above, $\mathcal{L}$ denotes the whole alphabet of the automaton and in the formulas, $\top$ denotes the Boolean "true", and $\downarrow_1 q$ (resp. $\downarrow_2 q$) is true if there exists an accepting run from state q on the first child (resp. next sibling) of the current node. The notion of (accepting) run for a given input tree is again the usual one:

- the root must be in a start state (here q_0)
- the leaves must be in an end state (here q_1 or q_2)
- a node x of the tree is in state q if there exists a transition $q, L \rightarrow \phi$, if the label of x, is in L and if the formula ϕ holds (which possibly requires some conditions on the left or right child of x).

The novelty here is the presence of the `mark` predicate (Transition 4) whose intuitive meaning is to remember the nodes in which the transition containing `mark` (here Transition 4) was valid.

Let us now describe informally the correspondence between the XPath formula and the automaton. The latter starts at the root of the input tree in state q_0. Here only Transition 1 can be satisfied. That is the case if the first child of the root is in state q_1. That state has two corresponding transitions (recall that our automaton is *non-deterministic*). Transition 2 requires that (i) the current label is `listitem` and q_2 hold for the first child and q_1 hold for both the first child and next sibling. Transition 3 has no requirement on the label (it can be anything in $\mathcal{L}$) but q_1 must hold for both the first child and next sibling of the current node. This self reference to q_1 in both directions simply encodes the recursion performed by the `descendant` axis in the query. Likewise for q_2 (and in general for any step of the query but the initial "/") there are two transitions. Transition 5 handles the recursion in case the label of the current node is not `keyword` or if the current node is a `keyword` which has no `emph` in child position. Transition 4 requires that the current node has label `keyword`, that q_2 holds for both first child and next sibling (recursion) *and* that q_3 holds for the first child. If that is the case, the node is *marked*. Lastly q_3 encodes the filter `[child::emph]`. If the current label is `emph`, then the transition is satisfied (there is no need to iterate in that case, since only one `emph`-node is sufficient for the filter to be true). Or the current node is not `emph`, and the automaton looks for an `emph` on the next sibling (recursion on $\downarrow_2$ only, which encodes a `child` axis).

There are of course several accepting runs for a given input tree (since the automaton is non-deterministic), but if a node is *marked* during a run, then it is a `keyword` node which has a `listitem`-node above it and an `emph` node amongst its children.

It is well-known that all the runs of a *non-deterministic* automaton can be simulated in a one pass traversal of the input (this holds for word automata as well as tree automata). Essentially, one maintains for each node a *set of states* (all those in which the non-deterministic automaton can be) instead of a single state. We show, after introducing formally our automata model, how we can compute the set of nodes marked during any non-deterministic run for a given automata, first using only FirstChild and NextSibling move, in one traversal of the tree. Then we show several optimizations techniques that allow us to leverage the speed of the low-level tree and text indices and compute efficiently the set of marked nodes.

5.3. Tree Automata Representation

Tree automata are a well-known and popular tool for reasoning about XML, see, for example, [57, 63, 29, 44]. Only seldom have they been used as a tool for query evaluation. In [32] automata are used to evaluate, on an XML stream, many (very simple) XPath queries in parallel. It is well-known that Core XPath can be evaluated using tree automata; see, for example, [39] and [9]. Here we use alternating tree automata (as in [18] and [35]). Such automata work with Boolean formulas over states, which must become satisfied for a transition to be triggered. This allows a much more compact representation of queries through automata, than ordinary tree automata (without formulas). Our tree automata are defined over a binary tree view of the XML tree where the left child is the first child of the XML node and the right child is the next sibling of the XML node.

Definition 5.1
A non-deterministic marking automaton $\mathcal{A}$ is a tuple $(\mathcal{L}, \mathcal{Q}, \mathcal{T}, \mathcal{B}, \delta)$, where:

- $\mathcal{L}$ is a countable (possibly infinite) set of tree labels;
- $\mathcal{Q}$ is a finite set of states;
- $\mathcal{T} \subseteq \mathcal{Q}$ is a set of top states (that is, states that must be satisfied at the root node);
- $\mathcal{B} \subseteq \mathcal{Q}$ is a set of bottom states (that is, states that must be satisfied at the leaves);
- $\delta : \mathcal{Q} \times 2_f^{\mathcal{L}} \cup 2_{cof}^{\mathcal{L}} \to F$ is a transition function, where F is the set of Boolean formulas, $2_f^{\mathcal{L}}$ is the set of finite subsets of $\mathcal{L}$, and $2_{cof}^{\mathcal{L}}$ is the set of co-finite subsets of $\mathcal{L}$. A *Boolean formula* ϕ is produced by the grammar:

$$
\begin{array}{llll}
\phi & ::= & \top \mid \bot \mid \texttt{mark} \mid \phi \vee \phi \mid \phi \wedge \phi \mid \neg\phi \mid a \mid p & \text{(formula)} \\
a & ::= & \downarrow_1 q \mid \downarrow_2 q & \text{(atom)}
\end{array}
$$

where $p \in P$ is a built-in *predicate* and q is a state.

Before explaining in detail the use of formulas, we motivate our use of finite or co-finite sets as guards for transitions. While traditionally automata transitions are guarded by a state and a single label, this would make the encoding of XPath into automata very tedious and needlessly complicate the algorithms. Indeed, one of the features of XPath is a wildcard element test, namely "*". One solution could be to suppose that for a given automaton the set of labels of the input document is known in advance and that this set is used as alphabet for the automaton. Unfortunately, this does not accurately reflect the semantics of XPath in which a query can be defined independently of any document and can even be executed on *any* document (it might not yield any result but its application is valid). Another solution (as in [32]) is to equip automata with a special "default" transition, labeled for instance "_", which is taken if in the current state no other transition can be evaluated. This has two drawbacks. Firstly, it is only well-defined for deterministic tree automata (our encoding makes heavy use of non-determinism). Secondly, the evaluation function is polluted by the special cases which handle this default transition. Our solution is more blunt. We guard transitions by finite or co-finite sets of labels, and a transition is taken if the label of the current node is a member of that set. For instance, the "*" XPath test is encoded as a transition guarded by the set $\mathcal{L} - \{@, \#\}$, where "@" and "#" represent labels of subtrees containing attribute nodes and text nodes in our encoding. This allows us to give a very straightforward evaluation function for tree automata, which relies on the evaluation of Boolean formulas, presented next.

Definition 5.2 (Evaluation of a formula)
Given an automaton $\mathcal{A}$ and an input tree t, the evaluation of a formula is given by the judgment $\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi = (b, R)$ where $\mathcal{R}_1$ and $\mathcal{R}_2$ are mappings from states to sets of nodes of t, t' is a node of t, ϕ is a formula, $b \in \{\top, \bot\}$, and R is a set of nodes of t. We define the semantics of this judgment by the means of the inference rules given in Figure 4.

$$\frac{}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \top = (\top, \emptyset)}\textbf{(true)} \qquad \frac{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi = (b, R)}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \neg\phi = (\bar{b}, \emptyset)}\textbf{(not)}$$

$$\frac{\begin{array}{c}\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_1 = (b_1, R_1)\\\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_2 = (b_2, R_2)\end{array}}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_1 \vee \phi_2 = (b_1, R_1) \oslash (b_2, R_2)}\textbf{(or)} \qquad \frac{\begin{array}{c}\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_1 = (b_1, R_1)\\\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_2 = (b_2, R_2)\end{array}}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_1 \wedge \phi_2 = (b_1, R_1) \oslash (b_2, R_2)}\textbf{(and)}$$

$$\frac{q \in \mathrm{dom}(\mathcal{R}_i)}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}}\downarrow_i q = (\top, \mathcal{R}(q))}\text{for } i \in \{1, 2\}\textbf{ (left,right)} \qquad \frac{}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \mathtt{mark} = (\top, \{t'\})}\textbf{(mark)}$$

$$\frac{EvalPred(p, t') = b, R}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} p = (b, R)}\textbf{(pred)} \qquad \frac{\text{when no other rule applies}}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi = (\bot, \emptyset)}$$

where:
$$\overline{\top} = \bot \quad \text{and} \quad \overline{\bot} = \top$$

$$(b_1, R_1) \oslash (b_2, R_2) = \begin{cases} \top, R_1 & \text{if } b_1 = \top, b_2 = \bot \\ \top, R_2 & \text{if } b_2 = \top, b_1 = \bot \\ \top, R_1 \cup R_2 & \text{if } b_1 = \top, b_2 = \top \\ \bot, \emptyset & \text{otherwise} \end{cases}$$

$$(b_1, R_1) \oslash (b_2, R_2) = \begin{cases} \top, R_1 \cup R_2 & \text{if } b_1 = \top, b_2 = \top \\ \bot, \emptyset & \text{otherwise} \end{cases}$$

Figure 4. Inference rules defining the evaluation of a formula.

These rules are straightforward and combine the rules for a classical alternating automaton, with the rules of a marking automaton. Rules **(or)** and **(and)** implement the Boolean connective of the formula and collect the marking found in their true sub-formulas. Rules **(left)** and **(right)** (written as a rule scheme for conciseness) evaluate to true if the state q is in the corresponding set. Intuitively, $\mathcal{R}_1$ (resp. $\mathcal{R}_2$) is the set of states recognizing the left (resp. right) subtree of the input tree. Rule

(pred) assumes the existence of an evaluation function for built-in predicates. Among the latter, we assume the existence of a special predicate `mark`, which evaluates to $\top$ and returns the singleton set containing the current node.

We now give the semantics of an automaton by means of the *run function* **TopDownRun** (see Figure 5). This algorithm is based on the textbook algorithm for recursive bottom-up evaluation of

---

Input An automaton $\mathcal{A} = (\mathcal{L}, \mathcal{Q}, \mathcal{T}, \mathcal{B}, \delta)$, a node x (of the implicit *Tree*), and a set of states $Q_{\mathrm{td}} \subseteq \mathcal{Q}$.

Output A mapping $\mathcal{R}$ from states to sets of nodes, such that $\mathcal{R}(q)$ is the set of nodes in the subtree rooted at x that were marked during an accepting run from x starting in state q.

TopDownRun$(\mathcal{A}, x, Q_{\mathrm{td}})$
1.   **If** x is `Nil` **Then**
2.   **Return** $\{q \to \varnothing \mid q \in \mathcal{B} \cap Q_{\mathrm{td}}\}$
3.   **Else**
4. $trans := \{(q, \ell \to \phi) \mid q \in Q_{\mathrm{td}}, Tag(x) \in \ell, (q, \ell \to \phi) \in \delta\}$
5. $Q_{\mathrm{td}}^i := \{q \mid \downarrow_i q \text{ occurs in } \phi, (q', \ell \to \phi) \in trans\}$ for $i \in \{1, 2\}$
6. $\mathcal{R}_1 := $ **TopDownRun**$(\mathcal{A}, \text{FirstChild}(x), Q_{\mathrm{td}}^1)$
7. $\mathcal{R}_2 := $ **TopDownRun**$(\mathcal{A}, \text{NextSibling}(x), Q_{\mathrm{td}}^2)$
8.   **Return** $\{q \mapsto R \mid \mathcal{R}_1, \mathcal{R}_2, x \vdash_{\mathcal{A}} \phi = (\top, R), (q, \ell \to \phi) \in trans\}$

Figure 5. Evaluation function for tree automata.

tree automata (see, e.g., [35]). The algorithm performs a recursive first child/next sibling traversal of the tree until a leaf is reached (base case for the recursion). When returning form the recursive evaluation on the left and right subtrees (Lines 6 and 7, Figure 5) the function evaluates the set of transitions for the current node, based on the set of states recognizing the left and right subtree. However, instead of blindly doing a recursive descent from the root to the leaves and evaluating when returning from the recursive calls, the transitions are restricted by the set of states Q_{td} (Line 4). This technique is dubbed "bottom-up evaluation with top-down preprocessing" in [35]. We therefore named the run function **TopDownRun** to differentiate it from a real bottom-up run (starting from the leaves of the tree) that we present in Section 5.4.2. The novelty is our use of maps from states to nodes instead of only sets of states. The resulting map associate any state q which has an accepting run from x with the set of nodes that were marked during that run.

5.4. *Leveraging the Speed of the Low-Level Interface*

We have seen how to evaluate an XPath query by compiling it into a tree automaton and running the latter on the input document. We present now several techniques that make use of the tree and text index presented in Sections 4 and 3. These are the techniques that make our SXSI prototype competitive in speed with state-of-the-art XML databases.

5.4.1. Jumping to relevant nodes Conventionally, the run of a tree automaton visits every node of the input tree. This is, for instance, the behaviour of the tree automata presented in [39], which perform two scans of the whole XML document (the latter being stored on disk in a particular format). However, for typical queries, most of the nodes are "useless" in the sense that the automaton only loops through them staying in the same set of states. In other words, the automaton *ignores* most of the nodes. To restrict the run to interesting nodes, we use the notion of *relevant nodes* introduced in [49]. While the full characterization is out of the scope of this paper, we give a flavor of relevant

nodes, using an example. Consider again the query

$$\texttt{/descendant::listitem/descendant::keyword[child::emph]} \ .$$

whose corresponding automaton is given in Figure 3. If we consider the starting transition, (Line 1) in Figure 3 we can see that at the root node, (labeled &) the automaton will first go on the first child, in state $\{q_1\}$. Then, it will loop, going down on the first child and next sibling of each node until it reaches a `listitem` element, on which it changes state and goes in $\{q_1, q_2\}$. Here we see, first, that there are no relevant nodes between the root and the first `listitem`. Indeed, the tree could be of any shape, and labeled with any tag (besides `listitem`), and the result of the query would be the same. Second, we see this `listitem` node is reached through the following sequence of moves (informally):

$$\downarrow_1 q_1 \cdot (\downarrow_1 q_1 | \downarrow_2 q_1)^*$$

(one first child move followed by an arbitrary sequence of first child and next sibling moves). Note, that all nodes that can be reached by such a sequence of move are *descendants* of the root node.

Our **TopDownRun** algorithm is therefore specialized as follows.

1. We compute from the formulas of the transitions we consider (Line 4, Figure 5) the set of states Q_1 (resp. Q_2) that are reached by a $\downarrow_1$ move (resp. $\downarrow_2$ move).
2. For Q_1 (resp. Q_2) we compute which set of labels cause a transition from one of the states in Q_1 to go in a state not in Q_1. These labels make the node that have them relevant.
3. We test whether the sequence of moves that can make the automaton leave the set of state Q_1 (resp. Q_2) has a specific pattern, for which our low-level library has an optimized call: TaggedDesc, TaggedFoll, ...
4. We use that special call instead of FirstChild (resp. NextSibling)

In the above example, we are initially in state $\{q_0\}$ with label &. We have $\{q_1\}$ as set of states Q_1. We see that a label `listitem` will make the automaton go from Q_1 to $Q_1' = \{q_1, q_2\}$. We therefore perform a TaggedDesc(_, `listitem`) to reach the next node (Line 6 of Figure 5)). The same process occurs recursively at each node, (e.g. after finding a `listitem` node, we look for a `keyword` node in the same fashion, jumping using low-level calls. Note that for the algorithm to remain sound, we cannot always jump in this fashion. For instance, once reaching a `keyword` node, we cannot jump arbitrarily, since e.g. on its first child, several situations may happen (all in parallel):

- the first child is labeled with `emph`, in which case the automaton will change back into a state that looks for a descendant `keyword`
- the first child is labeled with `keyword`, in which case the automaton will change into a state that (1) looks for a sibling of that node labeled `emph` and check below this second keyword node if there is a child labeled `emph`
- the first child is labeled by neither tag, then the automaton looks for a `emph` sibling and a `keyword` descendant

This behaviour is similar to the idea of "partitioning and pruning" in the staircase join [34], but here achieved by means of automata. One advantage of using automata instead of working directly on the syntactic form of the XPath query, is that some simplification occur "for free". For instance, query /descendant::*/descendant::a, /descendant::*/descendant::a, /descendant::*/child::a and /child::*/descendant::a are all executed in the same, efficient fashion: move one level down from the root then jump to all the `a` descendants. This happens because the for all these queries, the automata perform the same state change during the top-down phase.

Input An automaton $\mathcal{A} = (\mathcal{L}, \mathcal{Q}, \mathcal{T}, \mathcal{B}, \delta)$ and a sequence s of nodes in pre-order.

Output A mapping $\mathcal{R}$ from states to sets of nodes, such that $\mathcal{R}(q)$ is the set of nodes in the tree that were marked during an accepting bottom-up run that maps q to the root.

BottomUpRun$(\mathcal{A}, s)$
1. **If** s is empty **Then**
2.   **Return** $\varnothing$
3. **Else**
4. $x, s' := \text{Head}(s), \text{Tail}(s)$
5. $\mathcal{R} := \textbf{TopDownRun}(\mathcal{A}, x, \mathcal{Q})$
6. $\mathcal{R}', s'' := \textbf{MatchAbove}(\mathcal{A}, x, s', \mathcal{R}, \texttt{Nil})$
7.   **Return** $\mathcal{R}'$

Input An automaton $\mathcal{A} = (\mathcal{L}, \mathcal{Q}, \mathcal{T}, \mathcal{B}, \delta)$, a node x, a sequence s of nodes (in pre-order) occurring after x in pre-order, a mapping $\mathcal{R}_1$ from states to sets of descendant nodes of x, and a node *stop* that is an ancestor of x (or $\texttt{Nil}$).

Output A mapping $\mathcal{R}$ from states to sets of nodes and a sequence s' of nodes such that $\mathcal{R}(q)$ is the set of nodes that were marked during an accepting bottom-up run that maps q to the node *stop* and a sequence s of nodes that are not descendants of *stop*.

MatchAbove$(\mathcal{A}, x, s, \mathcal{R}_1, stop)$
8. $p := \text{Parent}(x)$
9. **If** $p = stop$ **Then**
10.   **Return** $\mathcal{R}_1, s$
11. **Else**
12.   **If** s is empty or not($\text{IsAncestor}(p, \text{Head}(s))$) **Then**
13. $\mathcal{R}_2, s'' := \varnothing, s$
14.   **Else**
15. $x', s' := \text{Head}(s), \text{Tail}(s)$
16. $\mathcal{R} := \textbf{TopDownRun}(\mathcal{A}, x', \mathcal{Q})$
17. $\mathcal{R}_2, s'' := \textbf{MatchAbove}(\mathcal{A}, x', s', \mathcal{R}, p)$
18. $trans := \{(q, \ell \to \phi) \in \delta \mid \exists i \in \{1, 2\}, \exists q' \in \text{dom}(\mathcal{R}_i), \downarrow_i q' \text{ occurs in } \phi, \text{ Tag}(p) \in \ell\}$
19. $\mathcal{R}' := \{q \mapsto R \mid \mathcal{R}_1, \mathcal{R}_2, p \vdash_{\mathcal{A}} \phi = (\top, R), (q, \ell \to \phi) \in trans\}$
20.   **Return MatchAbove**$(\mathcal{A}, p, s', \mathcal{R}', stop)$

Figure 6. Bottom-up evaluation function.

5.4.2. Bottom-Up Runs While the previous technique works well for tree-based queries it still remains slow for very selective value-based queries. For instance, consider the query

```
/descendant::listitem/descendant::keyword[contains(.,"Unique")] .
```

The text interface described in Section 3 can answer the text predicate very efficiently, returning the set of text nodes matching this *contains* query. If the number of occurrences is low, and in particular smaller than the number of `listitem` or `keyword` tags in the document (which can also be determined efficiently through the tree structure interface), then it would be faster to take these text nodes as starting points for query evaluation and test if their path upward to the root matches the XPath expression before the filter. This scheme (already mentioned in [52]) is particularly useful for text oriented queries with low selectivity text predicates. However, it also applies for tree only
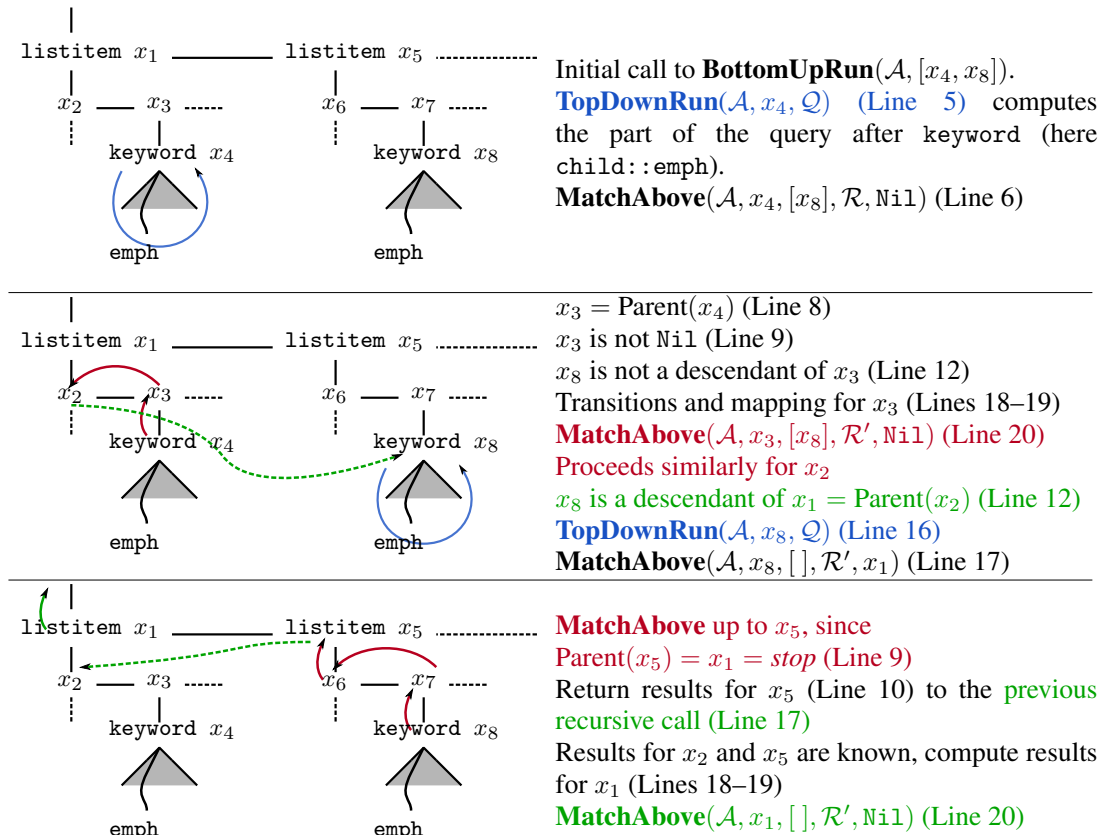
Initial call to **BottomUpRun**$(\mathcal{A}, [x_4, x_8])$.
TopDownRun$(\mathcal{A}, x_4, \mathcal{Q})$ (Line 5) computes the part of the query after `keyword` (here `child::emph`).
MatchAbove$(\mathcal{A}, x_4, [x_8], \mathcal{R}, \mathtt{Nil})$ (Line 6)

$x_3 = \mathrm{Parent}(x_4)$ (Line 8)
x_3 is not `Nil` (Line 9)
x_8 is not a descendant of x_3 (Line 12)
Transitions and mapping for x_3 (Lines 18–19)
MatchAbove$(\mathcal{A}, x_3, [x_8], \mathcal{R}', \mathtt{Nil})$ (Line 20)
Proceeds similarly for x_2
x_8 is a descendant of $x_1 = \mathrm{Parent}(x_2)$ (Line 12)
TopDownRun$(\mathcal{A}, x_8, \mathcal{Q})$ (Line 16)
MatchAbove$(\mathcal{A}, x_8, [\,], \mathcal{R}', x_1)$ (Line 17)

MatchAbove up to x_5, since
$\mathrm{Parent}(x_5) = x_1 = stop$ (Line 9)
Return results for x_5 (Line 10) to the previous recursive call (Line 17)
Results for x_2 and x_5 are known, compute results for x_1 (Lines 18–19)
MatchAbove$(\mathcal{A}, x_1, [\,], \mathcal{R}', \mathtt{Nil})$ (Line 20)

Figure 7. Illustration of the bottom-up run.

queries; consider again the query

$$\texttt{/descendant::listitem/descendant::keyword[child::emph]}$$

on a tree with many `listitem` nodes but only a few `keyword` nodes. We can start by jumping to the `keyword` nodes, run the remainder of the query (`child::emph`) on the subtree rooted at that node and then check its ancestors for `listitem` nodes. Note that with the tree index described in Section 4, we cannot directly jump to all bottom-most `keyword` nodes. We would need to iterate through all `keyword` nodes. Direct access could be provided through additional `sarrays` storing for each label its bottom-most nodes.

We now devise a real bottom-up evaluation algorithm of our automata. The algorithm takes an automaton and a sequence of potential match nodes (in our example, `keyword` labeled nodes). It then moves up to the root, using the Parent function and checks that the automaton arrives at the root node in a top state $q \in \mathcal{T}$. Note that, if naively done, such a bottom-up run will visit many nodes repeatedly: if a node is the common ancestor of m potential match nodes, then it would be visited m times. Instead, we move bottom-up left-to-right, and only move upwards from the left-most potential match until we reach its lowest common ancestor with the next potential match. This technique is similar in spirit to shift-reduce parsing (see [1]). Our bottom-up matching algorithm is given in Figure 6.

The behaviour of this algorithm is explained in detail on an example in Figure 7 (in this figure, vertical lines denote FirstChild edges, horizontal lines denote NextSibling edges, and dashed lines represent skipped subtrees). Intuitively, it takes as input a sequence of potential matches. For each of them it runs first the **TopDownRun** function to verify the downward context of the query (Lines 5 and 16). It then proceeds to walk upward (using Parent(_)) from a potential match node but

stops (*shift*) when it reaches an ancestor of the next match. The following matches are recursively handled and the algorithm can restart (*reduce*) when all the descendants of the current node have been treated.

For our algorithm to be sound, we need to be sure that the subtrees that are not visited (e.g. left subtree of x_2 and x_6 right subtrees of x_3, x_5 and x_7 in Figure 7) can safely be ignored. This is the case if the query at issue has the form

$$/axis :: step/ \ldots /axis :: step[pred]$$

(as in our previous example). Indeed, if a query has predicates in intermediate steps e.g.:

```
/descendant::listitem[descendant::bold]/descendant::keyword[child::emph]
```

then we might need to explore the whole subtree of a `listitem` if no `bold` node is found on the path between a `keyword` node and its `listitem` ancestor. We chose not to explore such extensions to the **BottomUpRun** algorithm since it would, in our opinion not yield a significant speed-up with respect to the **TopDownRun** algorithm. On the contrary the speed-ups provided by the version of **BottomUpRun** we presented in this section were sufficient to justify the use of a dedicated procedure for low-selectivity queries of a particular shape.

5.5. General Optimizations, On-the-fly Determinization

While the optimizations presented in the previous sections give the most important speed-up we describe hereafter a series of implementation techniques used for the efficient evaluation of automata.

5.5.1. Hash consing of data-structures

We use hash consing for all critical data-structures: sets of states, formulas, sets of transitions, sets of labels and so on. Hash consed values have the following two properties. First, structurally equal values share the same representation in memory. Therefore testing for equality of such values (e.g., testing that two sets of transitions are equal) consists in comparing their memory address (which is cheap). Second, to each such value we can associate a unique integer id (this can be its memory address, e.g., but more interestingly a small integer assigned at the creation of the value). These two properties —especially the second one— are instrumental to the other optimizations. Indeed, as described in [19], we can memoize (or cache) the results of expensive computations and reuse them when needed instead of recomputing them. We can associate to each function a table, indexed by the argument's id. While the first computation might be expensive, its result is stored once and for all in the table and can be retrieved with one pointer indirection later on, when the same computation is requested. We explain now how this generic technique comes into play for automata evaluation.

5.5.2. Just-in-time compilation of automata

In the **TopDownRun** algorithm (Figure 5) the most expensive operations are in Lines 4, 5, and 8. By expensive we mean that they take time $O(m)$ where m is the number of steps in the original query. At Line 4, we gather all the transitions that can be selected from the current label ℓ and set of states Q_{td}. From these we compute, at Line 5, the new set of states Q_{td}^1 and Q_{td}^2 onto which we will launch the recursive call. As explained in Section 5.4, from the set of states Q_{td}^1 (resp. Q_{td}^2) we compute the "jump" moves that the automaton will do to reach the next node in the left (resp. right) subtree. If none of the formulas requires the evaluation of a value predicate (such as `contains`, for instance) then we can see that this whole computation of Lines 4 and 5 can be cached in a 2-dimensional array, using only ℓ (the current label, identified by a small integer) and Q_{td} (a hash-consed set of states with a unique small id) as a key. In practice we store in this table a small sequence of instructions that are computed *at run time* and which represent the behaviour of the automaton for the next step (e.g., "jump to the next `keyword` label in state $\{q_0, q_1\}$"). This just-in-time compilation scheme absorbs in practice most of the overhead caused by the automaton machinery and makes running an automaton almost as fast as executing

a hand-written, precompiled function. In the same fashion, the computation of the judgment $\vdash_{\mathcal{A}}$ can be memoized, this time in two parts. First, the sets of states (that is the domain of the resulting mapping) is stored once and for all, and second, a sequence of instructions telling how to propagate the results from the left and right subtrees is stored and evaluated for each node.

5.5.3. Handling of result sets Maintaining sets of (result) nodes can be expensive. Our efficient management of sets of nodes relies on the following two observations. First, note that only the states outside of filters actually accumulate nodes. All other states always yield empty bindings. Thus we can split the set of states into marking and regular states. This reduces the number of $\varovee$ and $\varowedge$ operations on result sets. Note also that given a transition $q_i, \ell \to \downarrow_1 q_j \wedge \downarrow_2 q_k$ where q_i, q_j, and q_k are marking states, all nodes accumulated in q_j are in the left subtree of the current node. Likewise, all the nodes accumulated in q_k are subtrees of the right subtree of the current node. Thus both sets of nodes are disjoint and we do not need to keep sorted sets of nodes but only need sequences which support $O(1)$ concatenation. Computing the union of two result sets R_j and R_k can therefore be done in constant time and consequently $\varovee$ and $\varowedge$ can be implemented in constant time. Furthermore, if we are only interested in getting the number of results of a query, these sets can be replaced by integer counters. Marking a node corresponds to incrementing a counter and merging two sets (that is performing $\varovee$ or $\varowedge$) corresponds to adding two counters. The evaluation of formulas as well as the pre-order traversal we perform guarantees that marked node are not counted twice (that is, in our algorithms the results $\mathcal{R}$ becomes mapping from states to integers rather than mapping from states to sets of nodes).

5.5.4. Lazy result sets Another way to leverage the speed and jumping capabilities of our tree index is by making use of a lazy result set. Consider the query `/descendant::listitem/descendant::keyword` . When reaching a `listitem` node, the automaton is in a state which encodes the following behaviour: "accumulate all `keyword` nodes below this node". Therefore instead of having the automaton jump through the subtree to individually put each `keyword` node in the result set, we only store the `listitem` node (i.e., the current node during evaluation) and a flag to remember that during serialization, it is not the `listitem` node which should be printed but rather all its `keyword` descendants. Since our tree index allows us to reach each such node using a constant time jump operation, we delay the process of getting all the final result nodes until serialization, therefore speeding up the marking process. This not only saves time but also memory since the full set of nodes does not have to be materialized. When evaluating the query in *counting* mode (as described in the previous point), then we replace the lazy result set by a single call to e.g. SubtreeTags(_, `keyword`) which returns in constant time the number of `keyword`-labeled nodes. This number is then added to the counter of the corresponding state.

5.5.5. Early evaluation of formulas Another optimization consists in evaluating the Boolean formulas of the automaton as early as possible. First, remark that in the **TopDownRun** algorithm, a node is "visited" three times. Once when the automaton enters the node, during the top-down phase (Line 1). Here, we only know that at most all states in Q_{td} yield a successful run. Then when returning from the left subtree (Line 6), we know $\mathcal{R}_1$ that is, the states which yield an accepting run for the left subtree. The idea now is to perform a partial evaluation of formulas using only $\mathcal{R}_1$. If this happens to be sufficient to prove or disprove the states in Q_{td}, then the right subtree can be skipped altogether. This optimization is very important for filters as it insures that, for instance, in a query such as `/descendant::listitem[./descendant::keyword]` the run function only tests for the presence of the left-most `keyword` node below a `listitem` node.

5.5.6. Relative Tag position tables As explained earlier, the transitions for the query `.../descendant::keyword/...` would be (just-in-time) compiled into a piece of code performing a subtree traversal using TaggedDesc(_, `keyword`) and TaggedFoll(_ , `keyword`) instead of

FirstChild and NextSibling. This is already optimal for documents where `keyword` nodes may appear arbitrarily. However, it is often the case that labels are not recursive (that is, nodes with a label l do not occur below other l-labeled nodes). To further optimize the compilation of the automaton, we build –while indexing the document– four relative position tables, telling for each label l in the document the sets of labels that occur respectively in child position, descendant position, following-sibling position and following position. When compiling at run time the automaton and generating a call to TaggedDescendant for a label l, we check that this l label can indeed appear as descendant of the label of the current node (and similarly for other jumping functions). If the label does not occur, then the TaggedDescendant call is replaced by a constant function returning directly the correct sets of states for the left subtree as well as an empty result set, as if the automaton had made a full run on this subtree.

6. EXPERIMENTAL RESULTS

This section presents our experimental results and is organized as follows. We first describe our experimental settings, test machine, and benchmark data. We then provide a first round of experiments illustrating the raw performances of the tree and text index: indexing time and resulting index size, direct querying of the text index, and performing full pre-order traversal using FirstChild and NextSibling moves. A third subsection illustrates how the tree index and automata-based engine work together to achieve very fast tree-oriented query evaluation (in particular using the jumping moves described in Section 4.1.2). We then show how the automaton machinery can leverage the speed of both the text and tree index by evaluating queries containing both text and tree predicates. Lastly we illustrate the versatility of our approach: our engine is easily extended to support querying of XML document storing bio-genetic data as well as natural language.

We have implemented a prototype XPath evaluator based on the data structures and algorithms presented in the previous sections. Both the tree structure and the FM-index were developed in C++, while the XPath engine was written using the OCaml language.

6.1. Protocol

To validate our approach we benchmark our implementation against two well established XQuery implementations, MonetDB/XQuery and Qizx/DB. We describe our experimental settings hereafter.

Test machine Our test machine features an Intel Core i5 platform featuring eight 3.33Ghz cores, 16 GB of RAM and a S-ATA hard drive. The OS is a 64-bit version of Ubuntu Linux (11.04). The kernel version is 3.0 and the file system used to store the various files is ext4, with default settings. All tests are run on a minimal environment where only the tested program and essential services were running. We use the standard compiler and libraries available on this distribution (namely g++ 4.6.1, libxml2 2.7.8 for document parsing and OCaml 3.12.1).

Qizx/DB We use version 4.1 of Qizx/DB engine (free edition), running on top of the 64-bit version of the JVM (with the `-server` flag set as recommended in the Qizx user manual). The maximal amount of memory of the JVM is set to the maximal amount of physical memory (using the `-Xmx` flag). We also use the flag `-r` of the Qizx/DB command line interface, which allows us to re-run the same query without restarting the whole program (this ensures that the JVM's garbage collector and thread machinery do not impact the performances). We use the timing provided by Qizx debugging flags, and report the *serialization time* (which actually includes the materialization of the results in memory and the serialization).

MonetDB/XQuery We use version Oct2010-SP1 of MonetDB, and in particular, version 4.40.3 of MonetDB4 server and version 0.40.3 of the XQuery module (*pathfinder*). We use the timing

reported by the "`-t`" flag of MonetDB client program, `mclient`. We keep the materialization time and the serialization time separate.

Running times and memory reporting   Each query is executed eleven times in a row within the same program instance (using "`-r 11`" for Qizx and by evaluating the same query eleven times for MonetDB without restarting the server). Of these eleven runs, we discard the first and average the remaining ten, which we report as "running time" in the subsequent experiments. For all engines and all queries, the first run is always much slower (due to cold cache issues, garbage collector adjustments and so on). The running time does not take into account query parsing or the query optimization phases that take place before the actual query evaluation. We found these times to always be negligible for all queries and engines tested (around 1ms or less). We monitor the *resident set size* of each process, which corresponds to the amount of process memory actually mapped in physical memory. For the tests in which serialization is involved we serialize to the `/dev/null` device (that is, all the results are discarded without causing any output operation). We also ascertained that for all queries, all engines give the same node count and serialize *roughly* the same amount of data (small variations exist since, e.g., empty elements can be rendered `<a/>` or `<a></a>`).

Test data   Our test data is comprised of XMark documents [62] of various sizes (between 116MB and 1GB), a 83MB treebank document[†] and a 122MB Medline document[‡]. In Section 6.6 which investigates XPath text-oriented queries, we also experiment using a word-based FM-index within SXSI, and do this over a 2.3GB mediawiki document (part of the English "wiktionary"). Last, in Section 6.7 we experiment with a 132MB XML document composed of gene annotations and their DNA sequences.

Remarks   We also compared with Tauro [64]. Yet, as it uses a tailored query language, we could not produce comparable results.

6.2. Indexing

Our implementation features a versatile index. It is divided into three parts. First, the tree representation composed of the parenthesis structure, as well as the tag structure. Second, the FM-index encoding the text collection. Third, the auxiliary text representation allowing fast extraction of text content.

It is easy to determine from the query which parts of the index are needed in order to solve it, and thus load only those into main memory. For instance, if a query only involves tree navigation, then having the FM-index in memory is unnecessary. On the other hand, if we are interested in very selective text-oriented queries, then only the tree part and FM-index are needed (both for counting and serializing the results). In this case, serialization is a bit slower (due to the cost of text extraction from the FM-index) but remains acceptable since the number of results is low; see Table II.

Figure 8 reports the construction time and memory consumption of the indexing process, the loading time from disk into main memory of a constructed index, and a comparison between the size of the original document and the size of our in-memory structures.

For these indexes, a sampling factor $l = 64$ (cf. Section 3) was chosen. It should be noted that the size of the tree index plus the size of the FM-index is always less than the size of the original document.

It should further be noted that although loading time is acceptable, it dominates query answering time. This is however not a problem for the use case we have targeted: a main memory query engine where the same large document is queried many times. As mentioned in the Introduction, systems
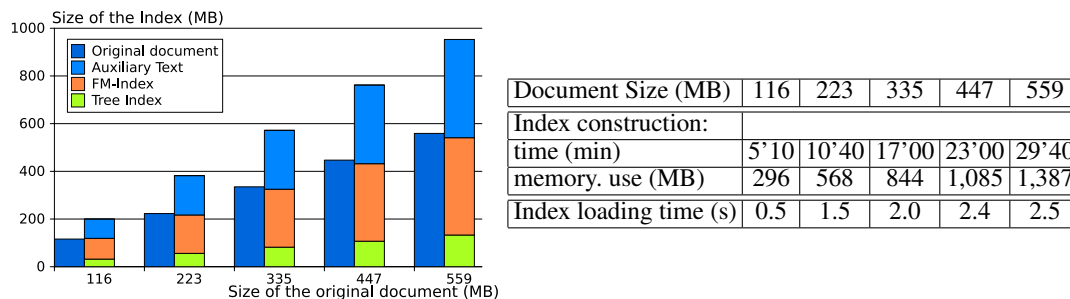

[†]`http://www.cs.washington.edu/research/xmldatasets`

[‡]http://www.ncbi.nlm.nih.gov/pubmed

Document Size (MB)	116	223	335	447	559
Index construction:					
time (min)	5'10	10'40	17'00	23'00	29'40
memory. use (MB)	296	568	844	1,085	1,387
Index loading time (s)	0.5	1.5	2.0	2.4	2.5

Figure 8. Indexing of XMark documents.

Table II. Search times of FM-index (in ms), sampling factor $l = 64$.

Query	Pattern	*GlobalCount* number	time	*ContainsCount* number	time	*Report-Contains*	mem (MB)
1	Bakst	1	.004	1	0.04	0.012	61
2	ruminants	22	.009	19	2.3	1.6	61
3	morphine	392	.009	144	4.6	4.5	61
4	AUSTRALIA	438	.009	438	29.9	32.7	61
5	molecule	1,472	.008	966	128.3	122.0	61
6	brain	2,685	.005	1,493	218.5	215.2	61
7	human	6,897	.005	4,690	553.5	548.0	62
8	blood	10,402	.005	8,534	401.2	399.7	62
9	from	20,859	.004	12,073	1,723	1,718	62
10	with	63,332	.004	22,974	5,084	5,084	63
11	in	238,638	.003	42,586	19,642	19,630	64
12	a	2,932,251	.001	595,716	189,299	188,377	93
13	\n	9,730,750	.001	5,870,474	132,780	132,241	86

such as MonetDB load their indexes only partially; this gives superior performance in a cold-cache scenario when compared with our system.

6.3. Raw Performance of Text Index

Here we give a short overview of the performance of our implementation of the FM-index. We present the search times for different versions of *contains*-queries:

1. *GlobalCount*(P): returns the global number of occurrences of the pattern P in all texts.
2. *ContainsCount*(P): returns the number of texts that contain P,
3. *ContainsReport*(P): returns the positions of all occurrences of P in the texts.

Our experiments are over the text collection obtained from a 122MB Medline XML document. The size of this text is around 82MB (if stored in one-byte per character ASCII format). Our "plain" alternative to the FM-index is a naive (byte-wise) string buffer (using precisely 82MB of memory). To search over the plain buffer, we use OCaml's regular string expression library. The naive search time is constant for all our queries at around 2,700ms. For both the naive and the FM-index, the result positions (32 bit integers) for *ContainsReport* queries are materialized in an array. Consider now the performance of our FM-index in comparison. First, at sampling factor $l = 64$, shown in Table II. As can be seen, the times for *ContainsCount* and *ContainsReport* for the word "from" are at around 1,720ms. Thus, in this case it is still faster to search over the FM-index. On the other hand, for the word "with" the search time is over 5,000ms, thus, here the plain search becomes faster. Hence, somewhere between 20,859 and 63,332 occurrences lies the cut-off point from which on searching over the plain text is faster than over the FM-index. Table III shows timings obtained with sampling factor $l = 4$. As can be seen the cut-off point is now much later, at a global count somewhere between 411,409 and 748,326. The last columns of Tables II and III show the maximal memory consumption

Table III. Search times of FM-index (in ms), sampling factor $l = 4$.

Query		GlobalCount number	time	ContainsCount number	time	Report-Contains	mem (MB)
1	Bakst	1	.005	1	0.049	0.013	100
2	ruminants	22	.010	19	0.156	0.086	100
3	morphine	392	.009	144	1.7	1.4	100
4	AUSTRALIA	438	.009	438	4.1	3.9	100
5	molecule	1,472	.009	966	6.2	5.9	101
6	brain	2,685	.006	1,493	12.2	11.6	101
7	human	6,897	.005	4,690	25.4	27.3	101
8	blood	10,402	.005	8,534	77.2	73.6	101
9	from	20,859	.003	12,073	84.0	78.7	101
10	with	63,332	.004	22,974	242.8	235.0	102
11	in	238,638	.002	42,586	1,105	1,091	103
12	b	411,409	.001	135,307	1,779	1,762	108
13	g	748,326	.001	320,440	3,412	3,379	119
14	a	2,932,251	001	595,716	13,183	13,173	133
15	\n	9,730,750	.001	5,870,474	87,771	88,230	126

Table IV. Construction times (in ms) for pointer versus SXSI tree store.

file	parse	pointers	parentheses	tags	tag-tabs
XMark116M	89,446	373	504	4,682	1,324
XMark223M	220,143	716	976	9,051	2,544
XMark559M	620,479	7,923	2,415	22,857	6,283
Treebank83M	67,412	465	615	14,067	18,867
medline122M	67,935	537	760	6,933	2,036

for these queries over the FM-index. As mentioned in the beginning of this section, we measure the maximum memory used by the process, as report by the operating system (this is a slight over-approximation of the actual memory). The memory overhead for queries with large cardinality, such as the last queries (q13 and q15), is explained by the size of the result array: for both sampling factors this is around 25MB. This query has around 6 million results (*ContainsCount*-number), each result is stored as a 4 Byte integer. Thus, 23MB are needed. However, additional memory overhead occurs when results are removed from the *GlobalCount* (because they occur in the same XML text node). For instance, in the second to last query (q12/q14) the ratio of *GlobalCount*-number to *ContainsCount*-number is much larger than for the last query (4.9 versus 1.7). This means that on average there are around 5 "a"-characters per text node, while there are only around 1.7 return-characters per text node. Correspondingly, the maximum memory consumption is much higher too.

6.4. Raw Performance of Tree Index

The performance of some low-level features of our tree index is compared with the corresponding performance of a standard pointer-based implementation of a tree. The latter provides for each tree node two 64-bit pointers to its first child and next sibling nodes (and does not store labels). We first compare construction times. Then we compare times for a full depth-first left-to-right tree traversal on the different structures. Finally, we test the speed of the taggedDesc and taggedFoll functions. We compare different traversals through all nodes with a given label: (i) using a pure C++ function, (ii) using our automata in counting mode, and (iii) using our automata in materialization mode.

Construction   As Table IV shows, the construction of the parentheses structure takes roughly 1.5-times the amount of time of allocation a pointer structure for the tree. Constructing the tag sequence is considerably slower, about ten times as much as building the parentheses structure. This is because, for each opening and for each closing tag, a separate `sarray` is constructed (see bottom left of Figure 1). The last column shows the time for building the four tag-to-tag tables described in

Table V. Traversal times (in ms), #nodes (in millions).

file	recursive, all nodes			element nodes, SXSI		
	#nodes	pointer	SXSI	#nodes	rec.	//*
XMark116M	6	33	109	1.7	71	153
XMark223M	12	63	209	3.3	137	296
XMark559M	30	164	535	8.4	345	756
Treebank83M	7	57	184	2.4	136	292
medline122M	9	48	164	2.9	112	244

Table VI. Times (in ms) for tagged traversals over XMark116M.

tag	#nodes	jump(C++)	//(cou)	//(mat)
category	1,040	1.2	1.6	1.7
price	10,141	2.3	2.9	3.1
listitem	63,179	16	22	24
keyword	73,070	11	12	14

Section 5.5.6. We also show the XML parsing time in the first column of the table, which dominates the rest of construction times.

Full Traversals The left part of Table V shows that a full tree traversal through all nodes is between 3.2 and 3.4 times slower with SXSI, than with a pointer tree data structure. Note that the pointers are allocated in pre-order too, giving optimal performance for pre-order traversal. As a comparison, if the pointers are allocated in post-order, then traversal time for the pre-order traversal is almost twice as slow as the numbers reported, and if pointers are allocated in in-order, then the times are a bit over twice as slow; see [3] for a discussion of the phenomenon. It should also be noted that for other access patterns, such as random root-to-leaf traversals, the time difference between pointer and succinct trees is much larger, factors of up to 100 are measured in [3].

In the right part of Table V we see the number of element nodes in these trees, and the time it takes for SXSI to recurse through those nodes: either using a small recursive C-function (column "rec."), or using the automaton for the XPath query //*, and executing in counting mode.

Tagged Traversals Here the speed of the TaggedDesc and TaggedFoll functions is investigated. Using these two functions, three different traversal through all nodes with a given label are considered: first, by a small C++ function, and second and third by our automata through a //label query in counting and materializing modes, respectively. For instance, Figure VI shows that iterating through all keyword-nodes of the 116MB Xmark document takes essentially the same time for all three methods (11–14ms). This is in contrast to some other labels: for listitem, for instance, the count-automaton traversal is 1.5-times slower than the C++ traversal. This can be explained by the fact that listitem is a recursive tag: there are in fact 23,298 listitem nodes that appear as descendants of listitem nodes. Hence, at each listitem node the automaton issues a taggedDescendant to search for further nodes. The other labels such as keyword and category do not appear recursively. Since this information is part of our tree index (cf. Section 5.5.6), the automaton run function avoids all these taggedDesc calls, which brings the speed almost up to the one of the C++ function.

6.5. XPath Tree Queries

We benchmark tree queries using the queries given in Figure 9. Queries X01 to X12 are taken from the XPathMark benchmark [27], derived from the XMark XQuery benchmark suite. X13 to X17 are "crash tests" that are either simple (X13 selects only the root since it always has at least one descendant in our files) or generate roughly the same amount of results but with various intermediate result sizes. Queries T01 to T05 work on the Treebank file.

```
T01  //NP
T02  //S[.//VP and .//NP]/VP/PP[IN]/NP/VBN
T03  //NP[.//JJ or .//CC]
T04  //CC[ not(.//JJ) ]
T05  //NN[.//VBZ or .//IN]/*[.//NN or .//_QUOTE_]

X01  /site/regions
X02  /site/regions/*/item
X03  /site/closed_auctions/closed_auction/annotation/description/text/keyword
X04  //listitem//keyword
X05  /site/closed_auctions/closed_auction[ annotation/description/text/keyword ]/date
X06  /site/closed_auctions/closed_auction[ .//keyword]/date
X07  /site/people/person[ profile/gender and profile/age]/name
X08  /site/people/person[ phone or homepage]/name
X09  /site/people/person[ address and (phone or homepage) and (creditcard or profile)]/name
X10  //listitem[not(.//keyword/emph)]//parlist
X11  //listitem[ (.//keyword or .//emph) and (.//emph or .//bold)]/parlist
X12  //people[ .//person[not(address)] and .//person[not(watches)]]/person[watches]
X13  /*[ .//* ]
X14  //*
X15  //*//*
X16  //*//*//*
X17  //*//*//*//*
```

Figure 9. Tree oriented queries based on Treebank (T01–T05) and XMark (X01–X17).

Query answering time   For this experiment we use the Treebank document (83MB) and XMark documents (116MB and 1GB). In the cases of MonetDB and Qizx, the files were indexed using the default settings. Let us first describe in detail Figure 10, which summarizes the running time for XMark queries. Each of the six graphs should be read as follows. For each query (X01 to X17), the graph reports as vertical bars the relative running time of the three engines with respect to SXSI's running time (therefore SXSI's score is always 100%). In these graphs a higher bar means that the engine was slower. We also give at the top of each bar the average running time for the query in milliseconds (or seconds, if the number is suffixed with an "s"). For instance, in the first graph — labelled "116 MB (counting)"— we can see that for query X01, SXSI evaluates the query in 1.8ms, MonetDB 7.4ms (or roughly 400% of SXSI's time) and QizX 3.6 ms (or roughly 200% of SXSI's time). For count queries, the timing for all three engines are given side by side (SXSI, MonetDB and QizX in that order). For full reporting queries however, we want to gauge precisely the amount of time spent during materialization and during serialization. The definition of materialization seems to fit the evaluation model of both MonetDB and SXSI: create a data structure in memory which holds the resulting nodes *in order* and *without duplicates* such that access of the first result in pre-order can be done in constant time, and accessing the next resulting node from the current one in pre-order can also be done in constant time. The timing for both SXSI and MonetDB are given in the graphs labelled "(materialization)". Since Qizx interleaves evaluation of the query and serialization, we only compared it to SXSI and MonetDB in the "(materialization+serialization)" series. We also checked that all three engines generated in the end the same amount of data while in serialization mode and that they generated valid XML documents (in particular, special characters were replaced by their corresponding XML entities —for instance "&" is rendered as"&"—).

From the results of Figure 10, we see how the different components of SXSI contribute to the efficient evaluation model. Fully qualified paths, such as queries X01–03 and X5 illustrate the sheer speed of the tree structure and in particular the efficiency of its basic operations (such as FirstChild and NextSibling, which are used for the `child` axis), as well as the efficient execution scheme provided by the automaton. The descendant axes (used, e.g., in X04, X06, X10–12) show the impact of the jumping primitives and the computation of relevant nodes. Complex filters (X06–12) show how the alternating automata can efficiently evaluate complex Boolean formulas corresponding to structural conditions over subtrees of a given node, including negations of paths and nested predicates. Finally, X12 to X17 illustrate the robustness of our automaton model. Indeed while
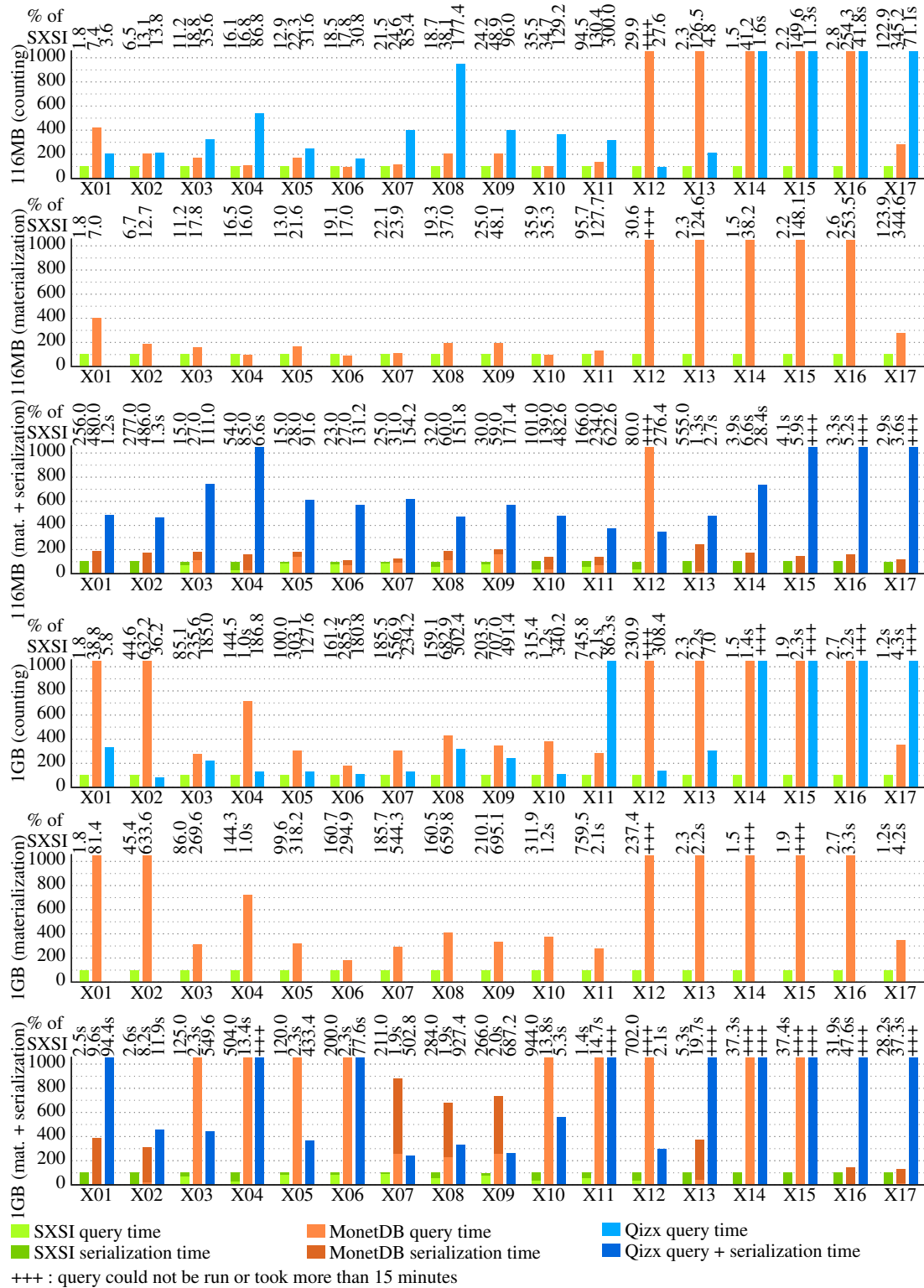
Figure 10. Running times for XMark queries, in milliseconds or seconds and as percent of SXSI's speed. Lower bars are better.

such queries might seem unrealistic, the good performances that we obtain are the combination of *(i)* using an automata based evaluator (which factors in the states of the automaton all the necessary

computation and thus do not materialize unneeded intermediate results) and *(ii)* our implementation of lazy result sets, which shifts the burden of walking through the document as much as possible to the serialization process.

As for Treebank queries, the results are reported in Figure 11. As one can see, SXSI behaves well for simple queries (such as T01) and for more complex ones (T02, T04 and T05). An important point is also the following one: the running times for all queries and for *all engines* are much worse than the ones obtained for larger XMark documents. It seems that, despite using three different approaches (tree automata+jump, staircase join or Qizx's proprietary one), a high number of distinct paths and labels impact query results much more than the final node count.
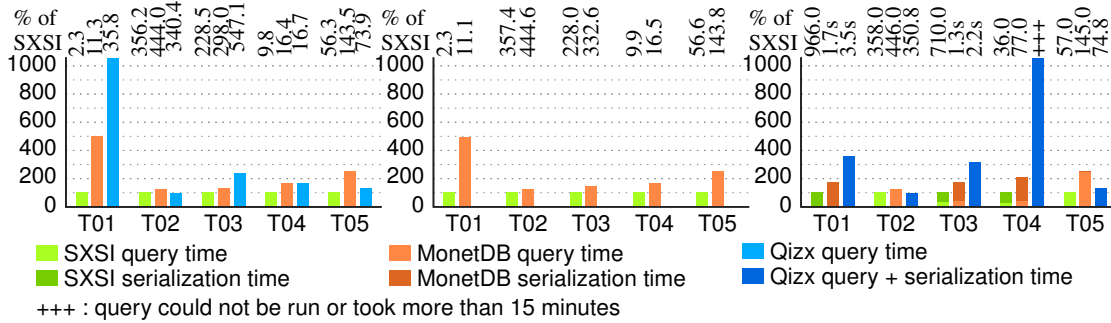
Figure 11. Running times for Treebank queries, in milliseconds or seconds and as percent of SXSI's speed. From left to right, counting, materialization, and serialization times. Lower bars are better.

Jumping and memoization of computations   As the experiments of the previous section show, SXSI often outperforms MonetDB and Qizx. To better highlight how the optimizations described in Sections 5.4 and 5.5 impact the running time of Algorithm 5, we selectively disabled some of them and executed queries X01 to X17 on a 116MB XMark document. The results are shown in Figure 12. The first bar of each cluster gives the running time of a naive version of the **TopDownRun** algorithm, where each node of the document is traversed and the computation of the next transitions to evaluate is re-computed for each node. In other words, in this run, the $|D|$ (size of the document) and $|Q|$ (size of the automaton/query) factors are paid in full. Here we can see that query answering time depends on both the size of the query (which increases from X01 to X12) but also the number
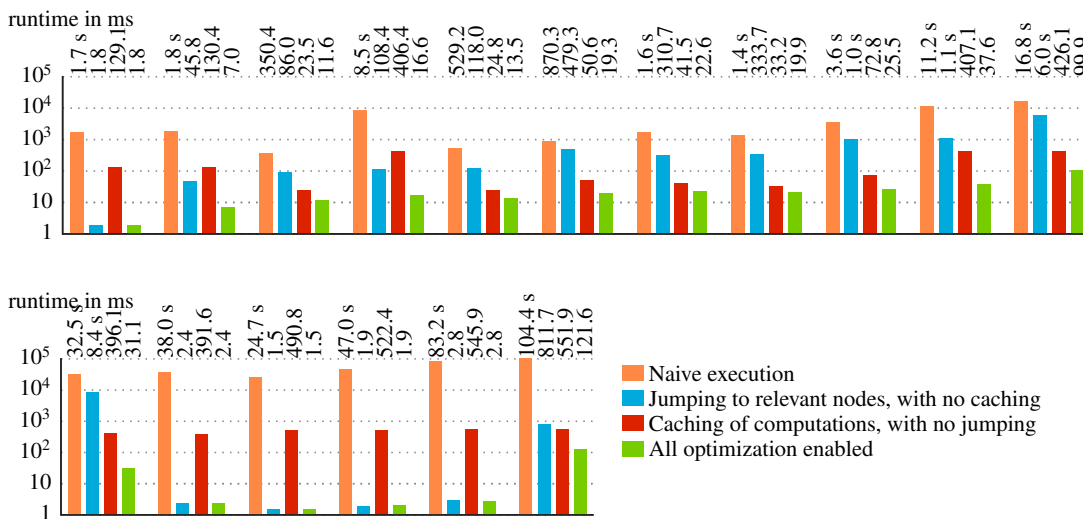
Figure 12. Impact of jumping and hash-consing on the running time of **TopDownRun** (Figure 5). Logarithmic scale, lower bars are better.
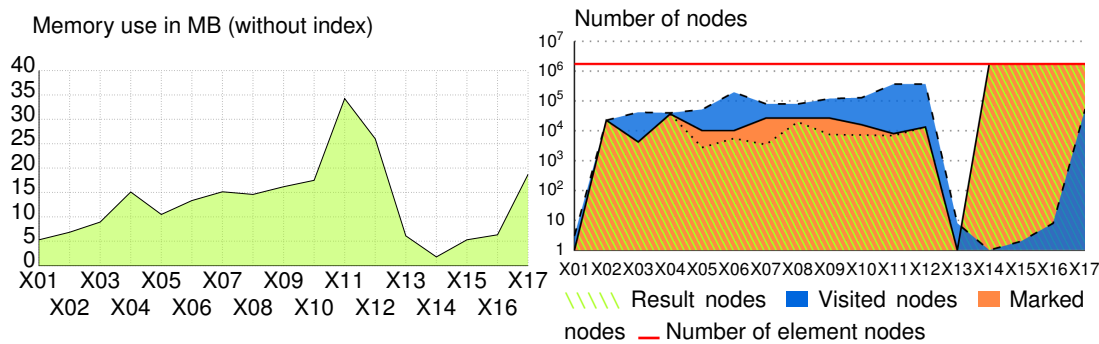
Figure 13. On the left, memory use in MB for XMark 116MB documents (excluding the index). On the right, comparison of visited, marked and result nodes for each query (logarithmic scale).

of selected nodes (e.g., X14 is a short query but selects all nodes, causing memory allocations and result sets book-keeping —which are expensive operations— for each node in the document).

The second bar in each cluster reports query answering time when computation and jumping to relevant nodes are performed (but these computations are not memoized). This includes not only jumping to relevant nodes, but also discarding whole subtrees and using the constant-time subtree counting function of the tree interface (Section 5.5.4). It comes to no surprise that skiping nodes aggressively yields important speed-ups in query answering time (some queries being translated into a single low-level function call). However queries such as X17 show the limit of the approach. For these queries (//*//*//* and //*//*//*//* resp.) the engine reaches all the element nodes at depth 3 (resp. 4) and select in constant time the set of elements in their subtrees. Because of the shape of the document, there are much more nodes at depth 4 than at depth 3, and therefore the automaton has to traverse more nodes to touch all those at depth 4. This explains why the inpact of jumping is much lower for query X17 than it is for X16.

The third bar in each cluster reports query answering time of a version of the **TopDownRun** that traverses the whole document, but memoizes the computations of Line 4, 5 and 8 in Figure 5 in look-up tables. This also decreases the query answering time greatly. Furthermore since this technique does not rely on a particular index, it shows that it would also benefit more traditional "DOM-based" query engines which lack the low-level jumping primitives.

Lastly, the fourth bar reports query answering time with both optimizations enabled. Here we see that the jumping optimization is further improved by the memoization: the somewhat expensive process that computes the next relevant node and the corresponding jump function, is stored in a look-up table and can be accessed in constant time when the automaton reaches again the same configuration.

Memory use and precision   While it is straightforward to predict the memory consumption of our engine with respect to the index part (the full index is mapped in memory excluding the auxiliary text, see Figure 8), the behaviour of the automaton evaluation function is unclear. Indeed, to speed-up the computation we create memoization tables, we handle partial result sets, and we perform recursive procedures that might be as deep as the *binary encoding* of the XML document (since we recurse on FirstChild and NextSibling move) thus increasing the size of the call stack.

We report in Figure 13 (left) the memory consumption for the automata evaluation of materialization queries. This includes the size of the recursive call stack, the size of OCaml's heap (which is grown dynamically by OCaml's garbage collector to accommodate the memory need). On the heap are allocated the memoization tables, intermediary structures and final result sets. As one can see, the memory use is very modest, peaking at 32 MB for query Q11. While we do not compare directly with MonetDB or QizX for memory consumption (since these engines try to maximize the memory used to achieve better speed) we see that we can reach comparable (if not greater) speed while being very conservative memory wise.

To gauge the precision of our automata based approach, we report in Figure 13 (right) for each query:

- the number of visited nodes (that is, the number of nodes onto wich the **TopDownRun** function is called);
- the number of marked nodes (that is, the number of nodes that are marked as potential results during the evaluation);
- and finally the number of result nodes for the query.

A first observation is that the number of marked nodes is almost always the same as the number of result nodes. In particular, for queries of the form:

$$/axis::step/\ldots/axis::step[pred]$$

we only mark nodes that are part of the final result (the first part of the path is resolved during the top-down phase, while the filter is validated during the bottom-up phase. Thus once reaching a potential result the automaton has recognized its path to the root as well as its subtree and can decide whether to select it or not). For general queries, the experimental results show that early evaluation of Boolean formulas, helps to decide early during query evaluation whether a node is indeed a result or not (even though some nodes are still selected and discarded later on). Another point of interest is that for several queries (X02, X04, X14–X17) we *only* visit result nodes. While this might be expected for queries X14–X17, for which virtually every node is a result, queries such as X02 or X04 are very selective. However, these queries provide enough information for the runtime analysis of relevant nodes to be *exact* and therefore only touch result nodes. In general of course, the number of traversed node is larger than the number of resulting nodes but always far less than the whole document. Queries X14 to X17 show the impact of lazy result sets where we mark several nodes (whole subtrees actually) in one function call, and therefore manage to return more nodes than we have actually visited.

Lastly, we can see that the shapes of the "Visited Nodes" curve in Figure 13 (right) and the memory use (Figure 13 left) are quite similar (the former being flattened by the logarithmic scale). This (quite expectedly) shows that the number of visited nodes (and not the number of result nodes or intermediary results) impacts directly the memory consumption of our query engine.

6.6. XPath Text Queries

In this section we illustrate how our approach can leverage the speed of the underlying text engine. Evaluation of text queries is performed as follows:

1. we determine during parsing whether the query can be run using the **BottomUp** algorithm (Section 5.4.2)
2. we determine whether the text predicates (`starts-with`, `ends-with`, `contains`, ...) is applied to a single text node; for example if the context node in the XPath expression is reached using "$axis$`::text()`" or if the content of a selected element is known to be `PCDATA` (this information is kept in the index). If that is the case we use the FMIndex and run the query bottom-up. Otherwise we revert to using the naive text representation to ensure that the semantics of the XPath text function is preserved, and run the query using the **TopDownRun** algorithm.

Here, the check in Step 2 is necessary to implement the semantics of XPath's text predicates over mixed content. Indeed, for such elements (containing both text and other elements) the semantics is to first create a text node resulting in the concatenation of all text elements and only then to perform the text predicate. For instance, on the document

```
<a>01<b>23</b>45</a>
```

the following query returns the root node as a result (i.e., the predicate evaluates to true):

```
/child::a[ contains( . , "1234") ] .
```

Since the text value of "`.`" (that is, the root node) is the concatenated string "`012345`" which indeed contains the substring "`1234`". While this "feature" of text predicates is rarely used in practice, its presence in the XPath specification can hinder the efficiency of "normal" text queries.

M01	//Article[.//AbstractText[contains (., "foot") or contains(. , "feet")]]	↓, ▲
M02	//Article[.//AbstractText[contains (. , "plus")]]	↑, ▲
M03	//Article[.//AbstractText[contains (. , "plus") or contains (. , "for")]]	↓, ▲
M04	//Article[.//AbstractText[contains (. , "plus") and not(contains (. , "for"))]]	↓, ▲
M05	//MedlineCitation/Article/AuthorList/Author[./LastName[starts-with(. , "Bar")]]	↑, ▲
M06	//*[.//LastName[contains(., "Nguyen")]]	↑, ▲
M07	//*//AbstractText[contains(., "epididymis")]	↑, ▲
M08	//*[.//PublicationType[ends-with(., "Article")]]	↑, ▲
M09	//MedlineCitation[.//Country[contains(. , "AUSTRALIA")]]	↑, ▲
M10	//MedlineCitation[contains(. , "blood cell")]	↓, ○
M11	//*/*[contains(. , "1999\n11\n26")]	↓, ○

↓: can be evaluated using the **TopDownRun** Alg. only
↑: can be evaluated using the **BottomUpRun** Alg.
▲: can use the FM-Index
○: can only use the naive index

Figure 14. Text oriented XPath queries over Medline and their evaluation strategies

6.6.1. FM-Index For this experiment, we compared the generic version of XPath text searching function (such as `contains`, `ends-with` and `start-with`) for all three engines on a 659MB Medline document (used to store biblographic data about medical publications). The queries we used for this experiment are given in Figure 14. In queries M01, M03 and M04, SXSI can use the FM-Index but cannot run bottom-up, since they feature complex filters. On the other hand, queries M02 and M05–M09 can be run purely bottom-up. Lastly, queries M10 and M11 need to use the "naive" text collection since the string that is searched for could overlap several text elements of the TextCollection (for M10, the `MedlineCitation` element has mixed content while for M11 it is not known at query compile time which element will be searched for the text).

The timings for these queries for all three engines are given in Figure 15.

The interpretation of the histograms is the same as for the tree oriented queries. For queries M01–M09, for which the FM-index can be used, we also give a table with the details of the amount of time spent querying the whole text collection for matches and the amount of time spent traversing the tree with the automaton. We also give the number of matches of the query.

As expected, in the case where queries can be evaluated in a bottom-up fashion the improvement in running time is of several orders of magnitude. Even when this is not the case, the improvement of using the FM-Index over the naive substring search completely justifies the somewhat longer indexing time and memory use. Lastly, for queries where the naive text representation must be used, the fact that our tree representation alows to return in constant time the set of texts that occurs as descendant of a given node allows us to avoid costly materialisation of large temporary text elements (as we can see in query M11, Qizx seems to also manage to avoid allocations, yielding a query answering time close to ours, while MonetDB seems to allocate large amounts of temporary character buffers, which impacts query answering time).

6.6.2. Word-based text index The loose coupling of SXSI's components (automata based query engine, succinct tree index and text index) allows us to readily plug other sorts of tree structures or text indices. The text collection presented in Section 3 achieves exact symbol-based text pattern-matching at the cost of indexing time and query time for large results. We show in this section how one can choose another trade-off, by pluging in the simple word-based text index of [20]. This index achieves high indexing and querying speed, with little memory by limiting queries at a word boundary. In this index, distinct words are treated as distinct symbols and the text collection is therefore viewed as a suffix array over a very large alphabet (the alphabet size is the number of distinct words in the original text). We compare SXSI equipped with this index to the full-text extension of Qizx. This extension implements the XQuery Full-Text facility [68] which allows to perform token-based (essentially word-based) queries. We used the queries in Figure 16 to test both

Figure 15. Running times for Medline queries, in milliseconds or seconds and as percent of SXSI's speed.
Lower bars are better.
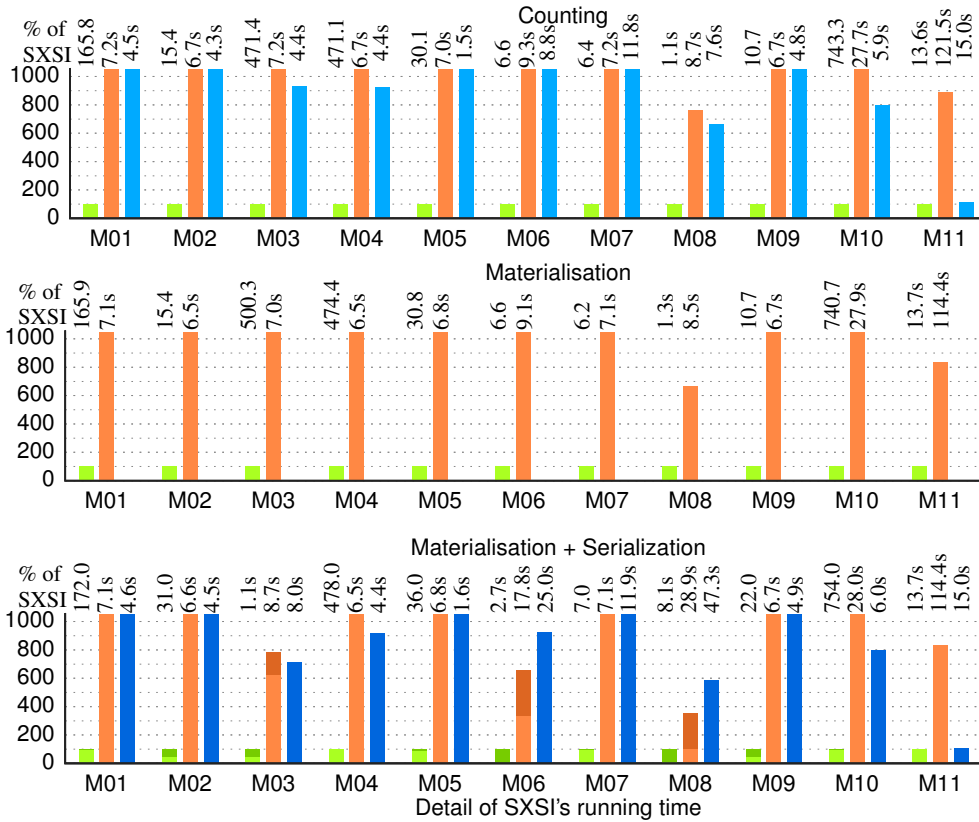
Query	# results	Text	Auto.	Total
M01	422	7.0	169.2	176.2
M02	1,009	6.2	9.7	15.9
M03	50,305	278.6	214.6	493.2
M04	219	272.3	210.0	482.3
M05	2,432	4.5	25.1	29.6

Query	# results	Text	Auto.	Total
M06	412	4.8	2.2	7.0
M07	48	2.8	5.3	8.1
M08	516,577	195.1	1,116	1,311
M09	832	3.2	7.1	7.3

indexes. While in this figure we used the "contains" function for conciseness, Qizx queries are
implemented using the special "ftcontains" operator (which performs word based queries).

We tested queries for both Qizx and SXSI on the 122MB Medline document and for SXSI only[§]
on a 2.3GB mediawiki document (a snapshot of the english "wiktionary" [66]). The results of the
experiment are given in Table VII. As we can see, even when both SXSI and Qizx are allowed
to make use of an efficient index, SXSI's bottom-up evaluation strategy clearly improves query
answering time (all queries but W03, W05 and W06 can be answered in 10ms or less). For queries
that involve several text predicates (i.e., that need to be evaluated in a top-down fashion) SXSI and
Qizx performs similarly (and efficiently, even for large documents).

[§]Unfortunately only the standard version of Qizx hits a hard-limit of around 2GB for a single document (while the
commercial "XL" version of the engine supports documents of 1TB).

```
W01    //Article[ .//AbstractText[ contains ( ., "blood sample") ] ]
W02    //Article[ .//AbstractText[ contains ( ., "is such that") ] ]
W03    //Article[ .//AbstractText[ contains( ., "various types of") and contains( ., "immune cells") ] ]
W04    //Article[ .//AbstractText[ contains( ., "of the bone marrow") ] ]
W05    //Article[ .//AbstractText[ contains( ., "cell") and not(contains( ., "blood")) ] ]

W06    //text[ contains ( ., "dark horse")]
W07    //text[ contains ( ., "horse") and contains( ., "princess") ]
W08    //page/child::title[ contains ( ., "crude oil") ]
W09    //page[.//text[ contains( ., "played on a board")]]/title
W10    //page[.//text[ contains( ., "whether accidentally or purposefully")]]/title
```

Figure 16. Word-based queries on Medline (W01–W05) and Wikimedia (W06–W10) documents

Table VII. Experimental results for word-based text queries

Query	W01	W02	W03	W04	W05	W06	W07	W08	W09	W10
SXSI (ms)	5.7	4.9	136.8	5.7	143.5	10.5	1600	10.6	7.8	10.2
Qizx (ms)	86.3	78.3	137.3	158.0	150.7	–	–	–	–	–

```
<!ELEMENT   chromosome   (name, gene*) >
<!ELEMENT   name         #PCDATA >
<!ELEMENT   gene         (name, strand, biotype, status,
      description?, promoter, sequence, transcript*) >
<!ELEMENT   strand       #PCDATA >
<!ELEMENT   biotype      #PCDATA >
<!ELEMENT   status       #PCDATA >
<!ELEMENT   description  #PCDATA >
<!ELEMENT   promoter     #PCDATA >
<!ELEMENT   sequence     #PCDATA >
<!ELEMENT   transcript   (name, start, end, exon*,
                          sequence, protein?) >
<!ELEMENT   start        #PCDATA >
<!ELEMENT   end          #PCDATA >
<!ELEMENT   exon         (name, start, end, sequence) >
```

Figure 17. DTD for bio-genetic data

6.7. Biological Sequence Queries

As a last experiment we demonstrate the versatility of SXSI by showing that it can be used as a very efficient biological database manager, answering queries which make use of both the tree structure and a tailored text index. More precisely, we create XML files that combine gene annotations with their DNA sequences. A sample DTD for these files is given in Figure 17. In this DTD, the elements promoter and sequence are of particular interest: they store the DNA represented as long sequences of A, T, C, G characters. The other #PCDATA elements store the gene annotation data such as positions, names and so on.

Our experiment data is composed from human chromosome five[¶] which contains 2,719 genes having in total 8,330 different transcripts. For each gene we include 1,000 base pairs of its upstream promoter sequence, the gene sequence itself (all exons and introns included), and annotation information such as gene's biotype and description. Additionally, we include all known transcripts of each gene, that is, sequences of the exons they contain as well as the concatenation of these exons. The resulting textual content is highly repetitive since each one of the exon sequences can appear in many transcripts. Highly repetitive data has been shown to compress well using certain run-length encoded text indexes [48], thus, here the text index implementation is switched to use

---

[¶]Ensemble Human genome release 59, August 2010.

query	# results	Text	Auto	Total
//promoter[PSSM(., M1)]	134	85.1	7.40	92.5
//promoter[PSSM(., M2)]	4	4.4	1.15	5.5
//promoter[PSSM(., M3)]	1	6.5	0.38	7.0
//exon[.//sequence[PSSM(., M1)]]	434	85.5	7.50	92.6
//exon[.//sequence[PSSM(., M2)]]	25	4.3	1.28	5.6
//exon[.//sequence[PSSM(., M3)]]	9	6.4	0.62	7.0
//*[PSSM(., M1)]	1,875	85.0	7.60	92.6
//*[PSSM(., M2)]	184	4.3	1.19	5.5
//*[PSSM(., M3)]	51	6.4	0.58	6.9

M1 : Jaspar ID = MA0031.1, length = 8, threshold = 5,000

M2 : Jaspar ID = MA0050.1, length = 12, threshold = 100,000

M3 : Jaspar ID = MA0017.1, length = 14, threshold = 300,000

Figure 18. Running times for PSSM queries (in ms).

RLCSA [48] instead of the FM-index. In this example, the final XML file[||] is 132 MB while the text index requires only 63 MB of memory plus 59 MB for the samples array. The full index, including tree and text, is around 135 MB, that is only as big as the original document. The resulting XML document contains 323,318 elements, of which 65,286 are either `promoter` or `sequence` nodes containing genetic data.

To do biologically relevant XML queries, we extend our engine to support PSSM queries (Position Specific Scoring Matrix) which allow us to search for transcription factor binding sites from genes' promoter regions. The input for this query is a Position Frequency Matrix (PFM) and a minimum threshold for a valid match. The matrices can be found from the Jaspar database [37].

In a nutshell, PFMs have one row for each symbol of the alphabet (in our case 4 rows A, T, C, G) and one column for each position in the pattern to search. For instance, the PFM:

$$\begin{bmatrix} A & 0 & 20 & 10 & 1 \\ T & 30 & 10 & 0 & 0 \\ C & 0 & 0 & 10 & 20 \\ G & 18 & 6 & 6 & 6 \end{bmatrix}$$

denotes patterns of length four, and the substring AGCT would get the score $0 + 6 + 10 + 10 = 26$. To form the PSSM query, the PFM matrix is first converted into log-odds form to take into account the uneven background distribution of nucleotide frequences. Then the PSSM query takes such a matrix as well as a threshold and returns all text elements whose content scores more than the given threshold. Figure 18 gives the running times for XPath queries using the PSSM predicates and RLCSA, with block size 128 and sample rate 16, as the text index. The table summarizes also the number of results, the length of the search pattern and the value of the threshold.

It is interesting to remark that since the document has a very flat and shallow structure, the automaton/tree part of the query evaluates always very quickly (7ms or under). The PSSM scheme also allows us to write biologically meaningful queries that would otherwise be impossible or very hard to write with regular expressions or a regular full-text extension. Yet, we did not have to modify our core engine, only the text index was modified in isolation to add PSSM capabilities; the automata and tree machinery remained unchanged.

7. CONCLUSIONS AND FUTURE WORK

We have presented SXSI, a system for compact in-memory representation of an XML collection and for fast indexed XPath queries over the representation. Even in its current prototype stage, SXSI is already competitive with well-known efficient systems such as MonetDB and Qizx. A number of avenues for future work are open. We mention the broadest ones.

[||] http://www.cs.helsinki.fi/group/suds/sxsi/data/

Handling updates to the collections is possible in principle, as there are dynamic data structures for sequences, trees, and text collections [16, 47, 61, 13, 56]. However, their practicality has not yet been established, nor how they relate to classical schemes that maintain a log of changes and reindex periodically.

The compact data structures used in SXSI support several fancy operations beyond those actually used by our XPath evaluator. A matter of future work is to explore other evaluation strategies that take advantage of those nonstandard capabilities. As an example, the current XPath evaluator does not use the range search capabilities of the structure *Doc* of Section 3. This could be useful in the case of top-down evaluation of queries that contain non-selective text searches; after a top-down phase the search on *Doc* could be restricted to the range of a particular subtree.

An interesting challenge is to support XPath string-value semantics, where strings spanning more than one text node can be searched for. This, at least at a rough level, is not hard to achieve with our FM-index, by removing the $-terminators and marking them on a separate bitmap instead.

We would like to extend our implementation to full XPath 1.0, and to add core functionalities of XQuery. A first step here is to add backward axes to our XPath fragment. Ideally, we would like to extend the lazy on-the-fly determinzation procedure of our automata, to a lazy remove-backward procedure which removes backward axes during the run of the automaton. We do however need to execute automata with backward axes too, because not every query can be rewritten into a forward one (see Section 3.2 of [7]). Even with backward axes and with the semi-joins of XPath, evaluation can be done in linear time (cf. [10]) and should be fast and predictable in SXSI. This efficiency will have to be given up in general, when we want to support more complex features such as numerical operations and loops of XQuery.

ACKNOWLEDGEMENTS

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.
2. D. Arroyuelo. An improved succinct representation for dynamic k-ary trees. In *CPM*, pages 277–289, 2008.
3. D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *ALENEX*, pages 84–97, 2010.
4. D. Arroyuelo, F. Claude, S. Maneth, V. Mäkinen, G. Navarro, K. Nguyen, J. Sirén, and N. Välimäki. Fast in-memory XPath search using compressed indexes. In *ICDE*, pages 417–428, 2010.
5. D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *CPM*, pages 319–330, 2006.
6. J. Barbay, T. Gagie, G. Navarro, and Y. Nekrich. Alphabet partitioning for compressed rank/select and applications. In *ISAAC (2)*, pages 315–326, 2010.
7. M. Benedikt and C. Koch. XPath leashed. *ACM Comput. Surv.*, 41(1), 2008.

8. D. Benoit, E. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

9. H. Björklund, W. Gelade, M. Marquardt, and W. Martens. Incremental XPath evaluation. In *ICDT*, pages 162–173, 2009.

10. M. Bojanczyk and P. Parys. Xpath evaluation in linear time. *J. ACM*, 58(4):17, 2011.

11. P. A. Boncz, T. Grust, M. van Keulen, S. Manegold, J. Rittinger, and J. Teubner. MonetDB/XQuery: a fast XQuery processor powered by a relational engine. In *SIGMOD*, pages 479–490, 2006.

12. A. Bonifati, G. Leighton, V. Makinen, S. Maneth, G. Navarro, and A. Pugliese. An in-memory XQuery/XPath engine over a compressed structured text representation. In *Structure-Based Compression of Complex Massive Data. Dagstuhl Seminar Proceedings 08261*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2008.

13. S. Böttcher, A. Bültmann, and R. Hartel. Search and modification in compressed texts. In *DCC*, pages 403–412, 2011.

14. S. Böttcher, R. Hartel, and C. Heinzemann. BSBC: Towards a succinct data format for XML streams. In *WEBIST*, pages 13–21 (vol. 1), 2008.

15. M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

16. H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Trans. Algorithms*, 3(2), 2007.

17. F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *SPIRE*, pages 176–187, 2008.

18. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, C. Löding, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. http://www.grappa.univ-lille3.fr/tata, 2007.

19. S. Conchon and J.-C. Filliâtre. Type-Safe Modular Hash-Consing. In *Proc. ACM SIGPLAN Workshop on ML*, 2006. http://www.lri.fr/~filliatr/ftp/publis/hash-consing2.ps.

20. A. Fariña, N. Brisaboa, G. Navarro, F. Claude, A. Places, and E. Rodríguez. Word-based self-indexes for natural language text. *ACM Trans. Inform. Syst.*, 30(1), 2012.

21. M. F. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax experience. In *VLDB*, pages 1077–1080, 2003.

22. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *FOCS*, pages 184–196, 2005.

23. P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Compressing and searching XML data via two zips. In *WWW*, pages 751–760, 2006.

24. P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 54(4):552–581, 2005.

25. P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Algorithms*, 3(2), 2007.

26. M. Franceschet. XPathMark: An XPath benchmark for the XMark generated data. In *XSym*, pages 129–143, 2005.

27. M. Franceschet. XPathMark: Functional and performance tests for XPath. In *XQuery Implementation Paradigms*, 2007. http://drops.dagstuhl.de/opus/volltexte/2007/892.

28. R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. In *CPM*, pages 159–172, 2004.

29. P. Genevès and N. Layaïda. XML reasoning made practical. In *ICDE*, pages 1169–1172, 2010.

30. A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *SODA*, pages 368–373, 2006.

31. G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30(2):444–491, 2005.

32. T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29:752–788, 2004.

33. R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA*, pages 841–850, 2003.

34. T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *VLDB*, pages 524–525, 2003.

35. H. Hosoya. *Foundations of XML Processing: The Tree Automata Approach*. Cambridge University Press, 2010.

36. G. Jacobson. Space-efficient static trees and graphs. In *FOCS*, pages 549–554, 1989.

37. JASPAR database. http://jaspar.genereg.net.

38. M. Kay. Ten reasons why Saxon XQuery is fast. *IEEE Data Eng. Bull.*, 31(4):65–74, 2008.

39. C. Koch. Efficient processing of expressive node-selecting queries on XML data in secondary storage: a tree automata-based approach. In *VLDB*, pages 249–260, 2003.

40. C. Koch, S. Scherzinger, and M. Schmidt. The GCX system: Dynamic buffer minimization in streaming xquery evaluation. In *VLDB*, pages 1378–1381, 2007.

41. T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008.

42. B. Langmead, C. Trapnell, M. Pop, and S. L. Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3), 2009. R25.

43. H. Li and R. Durbin. Fast and accurate short read alignment with burrows-wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

44. L. Libkin and C. Sirangelo. Reasoning about XML with temporal logics and automata. *J. Applied Logic*, 8:210–232, 2010.

45. V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *SPIRE*, pages 229–241, 2007.

46. V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theor. Comput. Sci.*, 387(3):332–347, 2007.

47. V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Trans. Algorithms*, 4(3), 2008.

48. V. Mäkinen, G. Navarro, J. Siren, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

49. S. Maneth and K. Nguyen. XPath whole query optimization. *PVLDB*, 3(1):882–893, 2010.

50. S. Maneth and T. Sebastian. Fast and tiny structural self-indexes for XML. *CoRR*, abs/1012.5696, 2010.

51. G. Manzini. An analysis of the Burrows-Wheeler transform. *J. ACM*, 48(3):407–430, 2001.

52. J. McHugh and J. Widom. Query optimization for xml. In *VLDB*, pages 315–326, 1999.

53. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *FOCS*, pages 118–126, 1997.

54. J. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM J. Comp.*, 31:762–776, 2001.

55. G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comp. Surv.*, 39(1), 2007.

56. G. Navarro and Y. Nekrich. Optimal dynamic sequence representations. In *SODA*, pages 865–876, 2013.

57. F. Neven. Automata theory for XML researchers. *SIGMOD Record*, 31:39–46, 2002.

58. D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *ALENEX*, 2007.

59. D. Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. Knowl. Data Eng.*, 19(7):934–949, 2007.

60. R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *SODA*, pages 233–242, 2002.

61. K. Sadakane and G. Navarro. Fully-functional static and dynamic succinct trees. In *SODA*, pages 134–149, 2010.

62. A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002.

63. T. Schwentick. Automata for XML - a survey. *J. Comput. Syst. Sci.*, 73:289–315, 2007.

64. Signum. Tauro. http://tauro.signum.sns.it/, 2008.

65. J. Sirén. Compressed suffix arrays for massive data. In *SPIRE*, pages 63–74, 2009.

66. English Wiktionary. `http://en.wiktionary.org/`.

67. XML Mind products. Qizx XML query engine. http://www.xmlmind.com/qizx, 2007.

68. XQuery and XPath Full Text 1.0. `http://www.w3.org/TR/xpath-full-text-10`.