Fast In-Memory XPath Search over Compressed Text and Tree Indexes

Diego Arroyuelo [#1], Francisco Claude [*2], Sebastian Maneth [+%3], Veli Mäkinen [†4], Gonzalo Navarro [$5], Kim Nguyễn [+6], Jouni Sirén [†7], Niko Välimäki [†8]

[#] *Yahoo! Research Latin America, Chile*
[1] darroyue@dcc.uchile.cl

[*] *David R. Cheriton School of Computer Science, University of Waterloo, Canada*
[2] fclaude@cs.uwaterloo.ca

[+] *NICTA, Australia*
[3] sebastian.maneth@nicta.com.au,
[6] kim.nguyen@nicta.com.au

[†] *Dept. of Computer Science, University of Helsinki, Finland*
[4] vmakinen@cs.helsinki.fi
[7] jltsiren@cs.helsinki.fi
[8] nvalimak@cs.helsinki.fi

[$] *Dept. of Computer Science, University of Chile, Chile*
[5] gnavarro@dcc.uchile.cl

[%] *CSE, University of New South Wales, Australia*

***Abstract*— A large fraction of an XML document typically consists of text data. The XPath query language allows text search via the equal, contains, and starts-with predicates. Such predicates can efficiently be implemented using a compressed self-index of the document's text nodes. Most queries, however, contain some parts of querying the text of the document, plus some parts of querying the tree structure. It is therefore a challenge to choose an appropriate evaluation order for a given query, which optimally leverages the execution speeds of the text and tree indexes. Here the SXSI system is introduced; it stores the tree structure of an XML document using a bit array of opening and closing brackets, and stores the text nodes of the document using a global compressed self-index. On top of these indexes sits an XPath query engine that is based on tree automata. The engine uses fast counting queries of the text index in order to dynamically determine whether to evaluate top-down or bottom-up with respect to the tree structure. The resulting system has several advantages over existing systems: (1) on pure tree queries (without text search) such as the XPathMark queries, the SXSI system performs on par or better than the fastest known systems MonetDB and Qizx, (2) on queries that use text search, SXSI outperforms the existing systems by 1–3 orders of magnitude (depending on the size of the result set), and (3) with respect to memory consumption, SXSI outperforms all other systems for counting-only queries.**

I. Introduction

As more and more data is stored, transmitted, queried, and manipulated in XML form, the popularity of XPath and XQuery as languages for querying semistructured data spreads faster. Solving those queries efficiently has proved to be quite 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 necessary. Each query must sequentially read the whole collection, 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 collection to build a data structure on it, so that later queries can be solved without traversing the whole collection. A serious challenge 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. Indexed schemes access the data more randomly than sequential ones, thus given the way disk costs favor sequential accesses, an indexed scheme can be outperformed by a streaming one even if the former accesses a relatively small fraction of the data. This is especially true in applications where the data itself can fit in main memory but the index cannot, so that streaming approaches do not need to access the disk at all. Those applications are becoming more common as current main memories become able of holding a few gigabytes of XML data. Examples of such systems are Qizx/DB [1], MonetDB/XQuery [2] and Tauro [3].

In this work we aim at an index for XML that uses little space compared to the size of the data, so that the indexed collection can fit in main memory for moderate-sized data, thereby solving XPath queries without any need of resorting to disk. An in-memory index should outperform streaming approaches, even when the data fits in RAM. Note that usually, main memory XML query systems (such as Saxon [4], Galax [5], Qizx/Open [1], etc.) use machine pointers to represent XML data; this blows up the memory consumption to about 5–10 times the size of the original XML document.

An XML collection 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 within compressed space is now well understood [6]–[8], and also much work has been carried out on compact data structures for trees [9]–[13]. In this paper we show how both 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 for XML data was recently proposed [14], [15], yet its support for XPath is reduced to a very limited class of queries that are handled particularly well.

The main value of our work is to provide the first practical and public tool for compressed indexing of XML data, dubbed *Succinct XML Self-Index* (SXSI), which takes little space, solves a significant portion of XPath (currently we support at least *Core XPath* [16], i.e., all navigational axes, plus the three text predicates $=$ (equality), *contains*, and *starts-with*), and largely outperforms the best public softwares supporting XPath we are aware of, namely MonetDB and Qizx. The main challenges in achieving our results have been to obtain practical implementations of compact data structures (for texts, 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 (this is a basic design decision, actually), that it is static (i.e., the index must be rebuilt when the XML data changes), and that it does not handle XQuery. The last two limitations are subject of future work.

II. Basic Concepts and Model

We regard an XML collection as (i) a 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 a dummy root so that we have a tree instead of a forest. Moreover, each text node is represented as a leaf labeled #. Attributes are handled as follows in this model. Each node with attributes is added a single child labeled @, and for each attribute `@attr=value` of the node, we add a child labeled `attr` to its @-node, and a leaf child labeled % to the `attr`-node. The text content `value` is then associated to that leaf. Therefore, there is exactly one string content associated to each tree leaf. We will refer to those strings as *texts*.

Let us call T the set of all the texts and u its total length measured in symbols, n the total number of tree nodes, Σ the alphabet of the strings and $\sigma = |\Sigma|$, t the total number of different tag and attribute names, and d the number of texts (or 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 and is lexicographically smaller than the other symbols in Σ. We can easily support multi-byte encodings such as Unicode.

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 collection (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. We created a dummy root labeled &, as well as dummy internal nodes #, @, and %. Note how the attributes are handled. There are 6 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, and will not be further mentioned.

Some notation and measures of compressibility follow, preceding a rough description of our space complexities. Logarithms will be in base 2. The *empirical k-th order entropy* [17] of a sequence S, $H_k(S) \leq \log \sigma$, is a lower bound to the output size per symbol of any k-th order compressor applied to S. We will build on self-indexes able of handling text collections T of total length u within $uH_k(T) + o(u \log \sigma)$ bits. On the other hand, representing an unlabeled tree of n nodes requires $2n - o(n)$ bits, and several representations using $2n + o(n)$ bits support many tree query and navigation operations in constant time. The labels require in principle other $n \log t$ bits. Sequences S can be stored within their zero-order entropy, $|S|H_0(S) + o(|S| \log \sigma)$, so that any element $S[i]$ can be accessed, and they can also answer queries $rank_c(S, i)$ (the number of c's in $S[1, i]$) and $select_c(S, j)$ (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) $uH_k(T) + o(u \log \sigma)$ 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) $2n + o(n)$ bits for representing the tree structure. This supports many navigational operations in constant time.
3) $d \log d + o(d \log d)$ bits for the string-to-text mapping, e.g., to determine to which text a string position belongs, or restricting string searches to some texts.
4) Optionally, $u \log \sigma$ or $uH_k(T) + o(u \log \sigma)$ bits, plus $O(d \log \frac{u}{d})$, to achieve faster text extraction than in 1).
5) $4n \log t + O(n)$ bits to represent the tags in a way that they support very fast XPath searches.
6) $2n + o(n)$ for mapping between tree nodes and texts.

As a practical yardstick: without the extra storage of texts (item 4) 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 store the memory consumption is between 1 and 2 times the size of the original XML file.

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

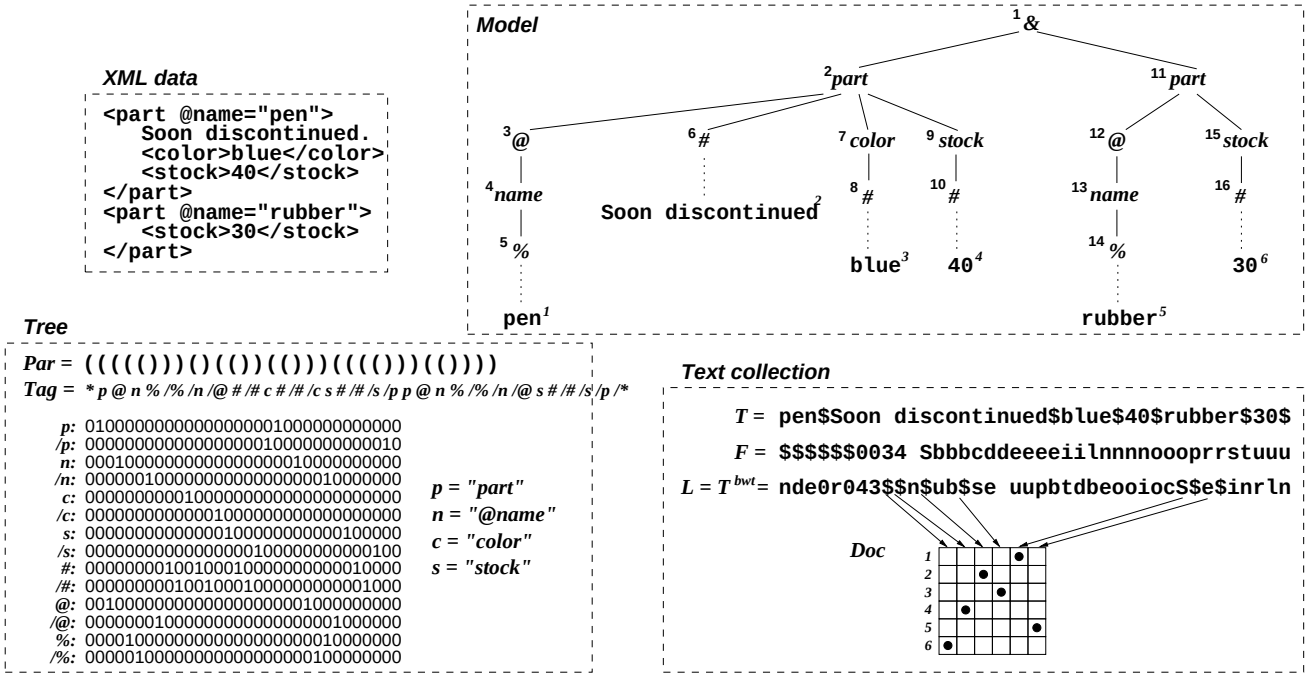


Fig. 1. Our running example on representing an XML collection.

way the correspondence between tree nodes and text identifiers works. This explains items 2, 5, and 6. Section V describes how we process XPath queries on top of these compact data structures. In Section VI we empirically compare our SXSI engine with the most relevant public engines we are aware of.

III. Text Representation

Text data is represented as a succinct full-text self-index [6] that is generally known as the *FM-index* [18]. The index supports efficient pattern matching that can be easily extended to support different XPath predicates.

A. FM-Index and Backward Searching

Given a string T of total length u, from an alphabet of size σ, the *alphabet-friendly FM-index* [19] requires $uH_k(T) + o(u \log \sigma)$ bits of space. The index supports counting the number of occurrences of a pattern P in $O(|P| \log \sigma)$ time. Locating the occurrences takes extra $O(\log^{1+\epsilon} u \log \sigma)$ time per answer for any constant $\epsilon > 1$.

The FM-index is based on the Burrows–Wheeler transform of a string T [20]. 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. Now the last column L of $\mathcal{M}$ forms a permutation of T which is the BWT string $L = T^{bwt}$. The matrix is only conceptual; the FM-index operates only on the actual T^{bwt} sequence. See Figure 1 (bottom right).

The resulting permutation is reversible. The first column of $\mathcal{M}$, denoted F, contains all symbols of T in lexicographic order. There exists a simple last-to-first mapping from symbols in L to F [18]: Let $C[c]$ be the total number of symbols in T that are lexicographically less than c. Now *LF-mapping* can be defined as $LF(i) = C[L[i]] + rank_{L[i]}(L, i)$. Symbols of T can be read in reverse order by starting from end-marker location i and applying $LF(i)$ recursively: we get $T^{bwt}[i], T^{bwt}[LF(i)], T^{bwt}[LF(LF(i))]$ etc. and finally, after u steps, get the first symbol of T. The values $C[c]$ can be stored in a small array of $\sigma \log u$ bits. The function $rank_c(L, i)$ can be computed in $O(\log \sigma)$ time with a *wavelet tree* data structure requiring only $uH_k(T) + o(u \log \sigma)$ bits [19], [21].

Pattern matching is supported via *backward searching* on the BWT [18]. Given a pattern P of length m, the backward search starts by finding the range $[sp, ep]$ of rows in $\mathcal{M}$ that have $P[m]$ as a prefix, say $sp = C[P[m]]$ and $ep = C[P[m] + 1]$. At each step $i \in \{m-1, m-2, \ldots, 1\}$ 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. New range $[sp', ep']$ is given by $sp' = C[P[i]] + rank_{P[i]}(L, sp - 1) + 1$ and $ep' = C[P[i]] + rank_{P[i]}(L, ep)$. Each step takes $O(\log \sigma)$ time [19], and finally $ep - sp + 1$ gives the number of times P occurs in T. To find out the location of an occurrence, the text is traversed backwards (virtually, using LF on T^{bwt}) until a sampled position is found. If every $l = \Theta(\log^{1+\epsilon} u)$ positions of T is sampled, locating takes $O(l \log \sigma)$ time per occurrence.

B. Text Collection and Queries

The textual content of the XML data is stored as \$-terminated strings so that each non-empty element corresponds to one text. Let T be the concatenated sequence of d texts. Since there are several \$'s in T, we fix a special ordering such that the end-marker of the i-th text will appear at $F[i]$ in $\mathcal{M}$. 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, denoted

Doc, that allows two-dimensional range searching [22] (see Figure 1). Mapping from a \$-terminator in position $T^{bwt}[i]$ to its entry in Doc can be calculated by $rank_{\$}(T^{bwt}, i)$. 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 $[sp, ep] \times [x, y]$ range in $O(\log d)$ time per answer. Doc requires $d \log d\,(1 + o(1))$ bits of space.

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:

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

2) ends-with$(P, [x, y])$*:* The given pattern is appended with \$. Backward searching is localized to texts $[x, y]$ by choosing $sp = x$ and $ep = y$ as the starting interval. After the backward search, the resulting range $[sp, ep]$ contains all possible matches, thus, existential and counting queries can be answered in constant time. To find out text identifiers for each occurrence, text must be traversed backwards to find a sampled position of the beginning of the current text. Cost is $O(l \log \sigma)$ per answer.

3) operator $=(P, [x, y])$*:* texts that are equal to P, and in range, can be found as follows. Do the backward search as in *ends-with*, then map to the \$-terminators like in *starts-with*. Time complexities are same as in *starts-with*.

4) 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, but the $O(|P| \log \sigma)$-time counting of all the occurrences of P can still be useful for query optimization.

5) operators $\leq, <, >, \geq$*:* The 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. If at some point there are no occurrences of c within the prefix $L[1, ep]$, we find those of smaller symbols in the range. This can be done by regarding the wavelet tree of the BWT as a range search data structure (as in previous work [22]). Other operators can be supported in similar fashion, and time complexities are the same as in *starts-with*.

The new XPath extension, *XPath Full Text 1.0*, suggests a wider selection of 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* [23]: 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 simply by computing $sp'_c = C[c] + rank_c(L, sp - 1) + 1$ and $ep'_c = C[c] + rank_c(L, ep)$ 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 against all substrings of the texts, allowing to search for approximate occurrences [23]. 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 [24], [25]. We omit further details here, since these extensions are out of the scope of this paper.

C. Implementation details

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 [26] (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 current implementation supports all the XPath text queries based on substring matching. We have also implemented approximate string matching and an experimental support for regular expressions. To enable fast text extraction from the collection, 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 [27]) using $uH_k(T) + o(u \log \sigma)$ bits. These secondary text representations are coupled with a delta-encoded bit vector storing starting positions of each text in T. This requires $O(d \log \frac{u}{d})$ more bits.

IV. Tree Representation

A. 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 Figure 1 (bottom left).

1) Par: The *balanced parentheses* representation [28] of the tree structure. This is obtained by traversing the tree in *depth-first-search* (DFS) order, writing a `"("` whenever we arrive at a node, and a `")"` when we leave it (thus it is easily produced during the XML parsing). In this way, every node is represented by a pair of matching opening and closing parentheses. A tree node will be identified by the position of its opening parenthesis in *Par* (in other words, in this representation a node is just an integer index within *Par*). In particular, we will use the balanced parentheses implementation of Sadakane [13], which supports a very complete set

of operations, including finding the i-th child of a node, in constant time. Overall *Par* uses $2n + o(n)$ bits. This includes the space needed for constant-time binary ranks on *Par*, which are very efficient in practice.

2) Tag: A 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 will also need $rank$ and $select$ queries on *Tag*. Several sequence representations supporting these are known [29]. 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 $\log(2t)$ 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 $M[1..2t][1..n]$ such that entry (i, j) is 1 if and only if *Tag*$[j] = i$. We represent each row of the matrix using Okanohara and Sadakane's structure `sarray` [30]. Its space requirement for each row i is $k_i \log \frac{2n}{k_i} + k_i(2 + o(1))$ bits, where k_i is the number of times symbol i appears in Tag. The total space of both structures adds up to $2n \log(2t) + 2nH_0(\textit{Tag}) + n(2+o(1)) \leq 4n \log t + O(n)$ bits. They support access and select in $O(1)$ time, and rank in $O(\log n)$ time.

B. Tree Navigation

We define the following operations over the tree structure, which will be 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. Let tag be a tag identifier.

1) Basic Tree Operations: These are direcly inherited from Sadakane's implementation [13]. We mention only the most important ones for this paper; x is a node (a position in Par).

- 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 nonlocal table accesses.
- Preorder$(x) = rank_(Par, i)$: Preorder number of x.
- SubtreeSize$(x) =$ (Close$(x) - x + 1)/2$: Number of nodes in the subtree rooted at x.
- IsAncestor$(x, y) = x \leq y \leq$ Close(x): Whether x is an ancestor of y.
- FirstChild$(x) = x + 1$: First child of x, if any.
- NextSibling$(x) =$ Close$(x) + 1$: Next sibling of x, if any.
- Parent(x): Parent of x. Somewhat costlier than Close(x) in practice, because the answer is less likely to be near x in Par.

2) Connecting to Tags: The following operations are essential for our fast XPath evaluation.

- SubtreeTags(x, tag): Returns the number of occurrences of tag within the subtree rooted at node x. This is $rank_{tag}(Tag,$ Close$(x)) - rank_{tag}(Tag, x - 1)$.
- Tag(x): Gives the tag identifier of node x. In our representation this is just *Tag*$[x]$.
- TaggedDesc(x, tag): The first node labeled tag with preorder larger than that of node x, and within the subtree rooted at x. This is $select_{tag}(Tag, rank_{tag}(Tag, x) + 1)$ if it is $\leq$ Close(x), and undefined otherwise.
- TaggedPrec(x, tag): The last node labeled tag with preorder smaller than that of node x, and not an ancestor of x. Let $r = rank_{tag}(Tag, x - 1)$. If $select_{tag}(Tag, r - 1)$ is not an ancestor of node x, then we stop. Otherwise, we set $r = r - 1$ and iterate.
- TaggedFoll(x, tag): The first node labeled tag with preorder larger than that of x, and not in the subtree of x. This is $select_{tag}(Tag, rank_{tag}(Tag,$ Close$(x)) + 1)$.

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, plus $o(n)$ extra bits to support rank/select queries. Bitmap B enables the computation of the following operations:

- 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 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).

C. Displaying Contents

Given a node x, we want to recreate its text (XML) content, that is, return the 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 III) and $O(1)$ per tag.

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

D. Handling Dynamic Sets

During XPath evaluation we need to handle sets of intermediate results, that is, global identifiers. Due to the mechanics of the evaluation, we need to start from an empty set and later carry out two types of operations:

- Insert a new identifier to the result.
- Remove a range of identifiers (actualy, a subtree).

To remove a range faster than by brute force, we use a data structure of $2n - 1$ bits representing a perfect binary tree over the interval of global identifiers, so that leaves of this binary tree represent individual positions and internal nodes ranges of positions (i.e., the union of their child ranges). A bit mark at each such internal node can be set to zero to implicitly set all its range to zero. A position is in the set iff all of its path from the root to it is not zero. Thus one can easily insert elements in $O(\log n)$ time, and remove ranges within the same time, as any range can be covered with $O(\log n)$ binary tree nodes.

V. XPath Queries

The aim is to support a practical subset of XPath, while being able to guarantee efficient evaluation based on the data structures described before. As a first shot we will support the "Core XPath" subset [16] of XPath 1.0. It supports all 12 navigational axes, all node tests, and filters with Boolean operations (and, or, not). In our prototype implementation, all axes have been implemented, but only the forward fragment (consisting of self, child, descendant, and following-sibling) has been fully optimized. We therefore focus here only on these two axes. A node test (nonterminal NodeTest below) is either the wildcard ('*'), a tagname, or a nodetype test, i.e., one of text() or node(); the node type tests comment() and processing-instruction() are not supported in our current prototype. Of course, we support all text predicates of XPath 1.0, i.e., the $=$, contains, and starts-with predicates. 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 ')'

A *data value* is the value of an attribute or the content of a text node. Here, all data values are considered as strings. If an XPath expression selects only data values, i.e., its final location step is the attribute-axis or a text() test, then we call it a *value expression*. Our XPath fragment ("Core+"), consists of Core XPath plus the following data value comparisons which may appear inside filters (that is, may be generated by the nonterminal Pred of above). Let w be a string and p a value expression; if p equals . (dot) or self and the XPath expression to the left of the filter is a value expression, then p is a value expression as well.

- p = w (equality): tests if a string selected by p is equal to w.
- contains(w, p): tests if the string w is contained in a string selected by p.
- starts-with(p, w): tests if the string w is a prefix of a string selected by p.

A. Tree Automata Representation

It is well-known that Core XPath can be evaluated using tree automata; see, e.g., [31]. Here we use alternating tree automata (as in [32]). Such automata work with Boolean formulas over states, which must become satisfied for a transition to fire. This allows much more compact representation of queries through automata, than ordinary tree automata (without formulas). Our tree automata work 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 (Non-deterministic marking automaton): An automaton $\mathcal{A}$ is a tuple $(\mathcal{L}, \mathcal{Q}, \mathcal{I}, \delta)$, where $\mathcal{L}$ is the infinite set of all possible tree labels, $\mathcal{Q}$ is the finite set of states, $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states, and $\delta : \mathcal{Q} \times 2^{\mathcal{L}} \to F$ is the transition function, where F is a set of Boolean formulas. A *Boolean formula* ϕ is generated by the following EBNF.

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

where $p \in P$ is a built-in *predicate* and q is a state. We call F the set of well-formed formulas.

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 judgement

$$\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi = (b, R)$$

where $\mathcal{R}_1$ and $\mathcal{R}_2$ are mappings from states to sets of subtrees of t, t' is a subtree of t ϕ is a formula, $b \in \{\top, \bot\}$ and R is a set of subtrees of t. We define the semantics of this judgment by the mean of inference rules, given in Figure 2.

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

$$\frac{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_1 = (b_1, R_1) \quad \mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_2 = (b_2, R_2)}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_1 \vee \phi_2 = (b_1, R_1) \veebar (b_2, R_2)}\textbf{(or)}$$

$$\frac{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_1 = (b_1, R_1) \quad \mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_2 = (b_2, R_2)}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \phi_1 \wedge \phi_2 = (b_1, R_1) \barwedge (b_2, R_2)}\textbf{(and)}$$

$$\frac{q \in \mathrm{dom}(\mathcal{R}_i)}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \downarrow_i q = (\top, \mathcal{R}(q))}\textbf{(left,right)}$$

$$\frac{}{\mathcal{R}_1, \mathcal{R}_2, t' \vdash_{\mathcal{A}} \texttt{mark} = (\top, \{t'\})}\textbf{(mark)}$$

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

where:

$$\overline{\top} = \bot \quad \overline{\bot} = \top$$

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

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

Fig. 2. Inference rules defining the evaluation of a formula

These rules are pretty straightforward and combine the rules for a classical alternating automaton, with the rules of a marking automaton. Rule **(or)** and **(and)** implements the Boolean connective of the formula and collect the marking found in their true sub-formulas. Rules **(left)** and **(right)** (written as a rule schema for concision) 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 accepted in the left (resp. right) subtree of the input tree. Rule **(pred)** supposes the existence of an evaluation function for built-in predicates. Among the latter, we suppose the existence of a special predicate, `mark` which evaluates to $\top$ and returns the singleton set containing the current subtree. We can now give the semantics of an automaton, by the means

of a *run function*.

Algorithm 5.1 (Top-down run function):
Input: $\mathcal{A} = (\mathcal{L}, \mathcal{Q}, \mathcal{I}, \delta)$,$t$,$r$ **Output:** $\mathcal{R}$
where $\mathcal{A}$ is the automaton, t the input tree, r a set of states and $\mathcal{R}$ a mapping from states of $\mathcal{Q}$ to sets of subtrees of t and such that $\text{dom}(\mathcal{R}) \subseteq r$.

```
1  function top_down_run A t r =
2    if t is the empty tree then return ∅
3    else
4    let trans = {(q, ℓ) → φ | q ∈ r and Tag(t) ∈ ℓ}
5    in
6    let r_i = {q | ↓_i q ∈ φ, ∀φ ∈ trans}
7    in
8    let R_1 = top_down_run A FirstChild(t) r_1
9    and R_2 = top_down_run A NextSibling(t) r_2
10   in return
11   {q ↦ R | R_1, R_2, t ⊢_A, φ = (⊤, R), ∀(q, ℓ → φ) ∈ trans }
```

The algorithm is straightforward. Although we called this function *top_down_run*, it is clear that it corresponds to the classical notion of *bottom-up* run for an automaton. Indeed, even though this function is called on the root node with the initial set of states, the sequence of recursive calls first reaches the leaves of the tree and starts evaluating the transitions while "returning" from a recursive call, hence when moving upward in the tree. This algorithm works in a very general setting. Considering any subtree t of our input tree, let $\mathcal{R}$ be the result of `top_down_run`$(\mathcal{A}, t, \mathcal{Q})$. Then $\text{dom}(\mathcal{R})$ is the set of states which accepts t and $\forall q \in \text{dom}(\mathcal{R})$, $\mathcal{R}(q)$ is the set of subtrees of t *marked* during a run starting from q on the tree t. It is easy to see that the evaluation of `top_down_run`$(\mathcal{A}, t, r)$ takes time $O(|\mathcal{A}| \times |t|)$, provided that the operations $\curlyvee$, $\curlywedge$ and `eval_pred` can be evaluated in constant time.

B. From XPath to Automata

The translation from XPath to alternating automata is simple and can be done in one pass through the parse tree of the XPath expression. Roughly speaking, the resulting automaton is "isomorphic" to the original query (and has approximately the same size). All our optimization discussed later are *on-the-fly* algorithms; for instance, we only determinize the automaton during its run on the input tree. Here, we only give an example of a query and its corresponding automaton: Consider the query `/descendant::listitem/descendant::keyword`. The corresponding automaton is $\mathcal{A} = (\mathcal{L}, \{q_0, q_1\}, \{q_0\}, \delta)$ where δ contains the following transitions:

1	$q_0, \{\texttt{listitem}\}$	$\to \downarrow_1 q_1$	4	$q_1, \{\texttt{keyword}\}$	$\to \texttt{mark}$
2	$q_0, \mathcal{L} - \{@, \#\}$	$\to \downarrow_1 q_0$	5	$q_1, \mathcal{L} - \{@, \#\}$	$\to \downarrow_1 q_1$
3	$q_0, \mathcal{L}$	$\to \downarrow_2 q_0$	6	$q_1, \mathcal{L}$	$\to \downarrow_2 q_1$

The automaton starts in state $\{q_0\}$ and traverses the tree until it finds a subtree labeled `listitem`. At such a subtree, the automaton changes to state $\{q_0, q_1\}$ on the left subtree (because it is nondeterministic and two transitions fire), looking for a tag `keyword` or possibly another tag `listitem` and it will recurse on the right subtree in state $\{q_0\}$ again. Transitions 2 and 5 make sure that, according to the semantics of the descendant axis, only element nodes (and not text or attributes) are considered. If, in state $\{q_0, q_1\}$ it finds a node labeled `keyword` then this node is marked as a result node.

C. General Optimizations, On-the-Fly Memoization

In Algorithm 5.1 the most expensive operation is in Line 11, which is evaluating the set of possible transitions and accumulating the mappings. 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 $\curlyvee$ and $\curlywedge$ operations on results sets. Note also that given a transition $q_i, \ell \to \downarrow_1 q_j \vee \downarrow_2 q_k$ where q_i, q_j and q_k are marking states, all nodes accumulated in q_j are subtrees of the left subtree of the input tree. Likewise, all the nodes accumulated in q_k are subtrees of the right subtree of the input tree. Thus both sets of nodes are disjoint. Therefore, we do not need to keep sorted sets of nodes but only need sequences which support $O(1)$ concatenation. Thus, computing the union of two result sets R_j and R_k can be done in constant time and therefore $\curlyvee$ and $\curlywedge$ can be implemented in constant time.

Another important practical improvement exploits the fact that the automata are very repetitive. For instance if an XPath query does not contain any data value predicate (such as `contains`) then its evaluation only depends on the tags of the input tree. We can use this to our advantage to *memoize* the results based on the tag of the input tree and the set r. Indeed, the set r and the tag of the input tree t uniquely define the set *trans* of possible transitions. So instead of computing such a set at every step, we can cache it in a hash-table where the key is the pair (Tag(t),r); this corresponds to an on-the-fly determinization of automata. Of course computing the hash of such a key must be fast for the operation to be beneficial. Labels are not a problem since, they are internally represented as integers. Sets however are trickier. We use the technique described in [33]. Basically, we represent sets of integers (which can be used for set of states, sets of tags, sets of transitions,...) as *hash-consed Patricia-trees*, which support $O(1)$ hashing and $O(1)$ equality checking. In practice the number of different values for the input set r is very small. We can further improve the running time by using an array of hashtables instead of a hashtable indexed by (`label`(t),r). We can apply a similar technique for the other expensive operation, that is, the evaluation of the set of formulas. This operation can be split in two parts: the evaluation of the formulas and the propagation of the result sets for the corresponding marking states. Again, if the formulas do not contain data value predicates, then their value only depends on the states present in $\mathcal{R}_1$ and $\mathcal{R}_2$, the results of the recursive calls. Using the same technique, we can memoize the results in a hash table indexed by the key $(\text{dom}(\mathcal{R}_1), \text{dom}(\mathcal{R}_2))$. This hash table contains the pair $\text{dom}(\mathcal{R})$ of the states in the result mapping and a sequence of affectation to evaluate, of the form [q_i`:=concat`$(q_j, q_k), \ldots$], which represents that need to be propagated between the different marking states. Another optimization is for the result set associated with the initial state of the automaton, which is answer of the query. This result

set is "final" in the sense that anything that was propagated up to it will be in the result set. We can exploit this fact and use a more compact data-structure for this set of results (for instance the one described in Section IV-D). Thus we can trade time complexity (since insertion is $O(log(n))$ in this structure) for space. Using this scheme, we are able to answer queries containing billions using little memory.

D. Leveraging the Speed of the Low-Level Interface

Conventionally, the run of a tree automaton visits every node of the input tree. For highly efficient XPath evaluation, this is not good enough and we must find ways to restrict the run to the nodes that are "relevant" for the query (this is precisely what is also done through "partitioning and pruning" in the staircase join [34]). Consider the query `/descendant::listitem/descendant::keyword` of before. Clearly, we only care about listitem and keyword nodes for this query, and how they are situated with respect to each other. This is precisely the information that is provided through the TaggedDesc and TaggedFoll functions of the tree representation. These functions allow us to have a "contracted" view of the tree, restricted to nodes with certain labels of interest (but preserving the overall tree structure). For instance, to solve the above query we can call TaggedDesc(Root,listitem) which selects the first listitem-node x. Now we can apply recursively TaggedDesc(x,keyword) and TaggedFoll(y,keyword) in order to select all keyword-descendants of x. We do this optimization of "jumping run" based on the automaton: for a given set of states of the automaton we compute the set of relevant transitions which cause a state change. For instance, in the automaton for the above query which is shown in Section V-B only transitions 1 and 4 are relevant. Thus, in state $\{q_0\}$ the automaton can use TaggedDesc to jump to listitem nodes, and in state $\{q_0, q_1\}$ it can jump to listitem or keyword nodes.

Bottom-up run: While the previous technique works well for tree-based queries it still remains slow for value-based queries. For instance, consider the query `//listitem//keyword[contains(.,"Unique")]`. The text interface described in Section III can answer the string query very efficiently returning the set of text nodes matching this *contains* query. It is also able to count globally the number of such results. If this number 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 point for query evaluation and test if their path to the root matches the XPath expression `//listitem//keyword`. This scheme is particularly useful for text oriented queries with low selectivity. However, it also applies for tree only queries: imagine the query `//listitem//keyword` on a tree with many listitem nodes but only a few keyword nodes. We can start bottom-up by jumping to the keyword nodes and then checking their ancestors for listitem nodes.

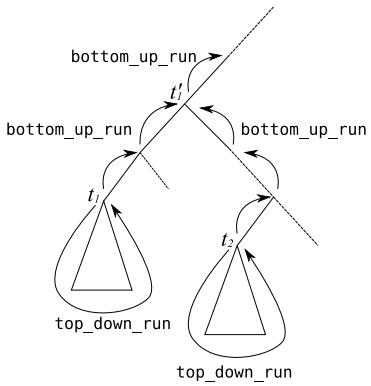


Fig. 3. Illustration of the bottom-up run

To achieve this goal, we devise a real `bottom-up` evaluation algorithm of an automaton. The algorithm takes an automaton and a sequence of potential matching nodes (in our example, the text nodes containing the string `"Unique"`). It then moves up to the root, using the `parent` function and checks that the automaton arrives at the root node in its initial state q_i. This scheme is illustrated in Figure 3. The technique used is similar to shift-reduce parsing. Consider a sequence `[`t_1,...,t_n`]` (ordered in pre-order) of potentially matching subtrees. In our previous example these were text nodes but this is not a necessary condition. The algorithm starts on tree t_1. First, if the tree is not a leaf, we call the `top_down_run` function on t_1 with $r = \mathcal{Q}$. This returns the mapping $\mathcal{R}_1$ of all states accepting t_1. We now want to move up to the root from t_1 in state $\text{dom}(\mathcal{R})_1$ and by taking the transitions upward. As illustrated in Figure 3 however, we do not want to move blindly from t_1 to the root. Indeed, once we arrive at a node t'_1 which is an ancestor of the next potential matching subtree t_2, then we stop at t'_1 and start the algorithm on t_2 until it reaches the lowest common ancestor t'_1. Once this is done, we can merge both mappings and continue upward on from t'_1 until we reach the root or a common ancestor of t'_1 and t_3 and so on. The idea of *merging* the runs at the lowest common ancestor makes sure that we never touch any nodes more than once, in a bottom-up move. We now give formally the bottom up algorithm.

Algorithm 5.2 (Bottom-up run function):
Input: $\mathcal{A}, s$ **Output:**$\mathcal{R}$
where $\mathcal{A}$ is an automaton, s a sequence of subtrees of the input tree, and R a mapping from states of $\mathcal{A}$ to subtrees of the input tree.

```
1  function bottom_up_run A s =
2   if s = [] then return ∅
3   else
4    let t,s' = hd(s), tl(s) in
5    let R = top_down_run A t Q in
6    let R', s'' = match_above A t s' R # 
7    in
8    R' ∪ (bottom_up_run A s'')
9
10 function match_above A t s R1 stop =
11   if t = stop then R1, s
```

```
12    else
13    let pt = Parent(t) in
14    let R_2, s' =
15       if s = [] or not (IsAncestor(pt,hd(s)))
16       then ∅, s
17       else
18       let t_2,s' = hd(s), tl(s) in
19       let R = top_down_run A t_2 Q in
20       match_above A _2 s' R pt
21    in
22    let trans = {q, ℓ → φ |  ∃q' ∈ dom(R_i) s.t. ↓_i q' ∈ φ
                               label(pt) ∈ ℓ                 }
23    in
24    let R' = {q ↦ R |  R_1, R_2, t ⊢_A, φ = (⊤, R),
                         ∀(q, ℓ → φ) ∈ trans              }
25    in
26    match_above A pt s' R' stop
```

In the light of Figure 3, it is easy to understand the two functions defined in Algorithm 5.2. The first one iterates the auxiliary function `match_above` on every tree in the sequence s. The `match_above` function is the one "climbing-up" the tree. We assume that the Parent(_) function returns the empty tree when applied to the root node. If the input tree is not equal to the tree *stop* (which is initially the empty tree #, allowing to stop only after the root node has been processed) then we first check whether the next (we use the function `hd` and `tl` which returns the first element of the list and its tail) potential tree is a descendant of our parent (Line 14). If it is so, then we pause for the current branch and recursively call *match_above* with our parent as *stop* tree. Once it returns, we compute all the possible transitions that the automata can take from the parent node to arrive on the left and right subtree with the correct configuration (Line 21). Once this is done, we *merge* both configuration using the same computation as in the top-down algorithm (Line 23). Finally, we recursively call `match_above` on the parent node, with the new configuration and sequence of potential matching nodes (Line 25).

VI. Experimental Results

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

A. Protocol

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

Test machine: Our test machine features an Intel Core2 Xeon processor at 3.6Ghz, 3.8 GB of RAM and a S-ATA hard drive. The OS is a 64-bit version of Ubuntu Linux. The kernel version is 2.6.27 and the file system used to store the various files is ext3, with default settings. All tests were run on a minimal environment where only the tested program and essential services were running. We used the standard compiler and libraries available on this distribution (namely g++ 4.3.2, libxml2 2.6.32 for document parsing and OCaml 3.11.0).

Qizx/DB: We used version 3.0 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 set to the maximal amount of physical memory (using the `-Xmx` flag). We also used 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 performance). We used the timing provided by Qizx debugging flags, and reported the *serialization time* (which actually includes the materialization of the results in memory and the serialization).

MonetDB/XQuery: We used version Feb2009-SP2 of MonetDB, and in particular, version 4.28.4 of MonetDB4 server and version 0.28.4 of the XQuery module (*pathfinder*). We used the timing reported by the "`-t`" flag of MonetDB client program, `mclient`. We kept the materialization time and the serialization time separated.

Running times and memory reporting: For each query, we kept the best of five runs. For Qizx/DB, each individual run consists of two repeated runs ("`-r 2`"), the second one being always faster. For MonetDB, before each of the five runs, the server was exited properly and restarted. We monitored the memory usage by reading, every 200 ms during the duration of the tested program, the `/proc/pid/statm` pseudo-file provided by Linux. More specifically, we monitored the so-called *resident set size*, which corresponds to the amount of process memory actually mapped in physical memory. For MonetDB, we kept track of the memory usage of both server and client. The peak of memory reported was the sum of client's peak plus server's peak.

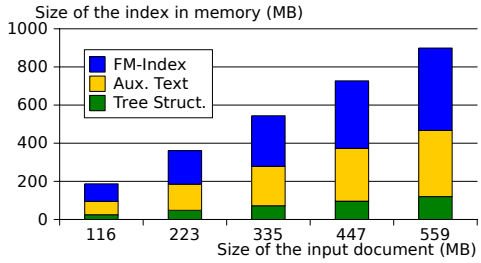
For the tests where serialization was involved, we serialized to the `/dev/null` device (that is, all the results were discarded without causing any output operation).

B. 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.

Figure 4 shows the construction time and the memory used during the indexing process. For these indexes, a sampling factor $l = 64$ (cf. Section III) was chosen. As we see, the *whole* index, including tree structure, FM-index and auxiliary



Document Size (MB)	116	223	335	447	559
Index Construction time (min)	9	30	42	54	67
Memory requirement of the build process (MB)	301	557	854	1069	1323

Fig. 4. Indexing of XMark documents

Q01 /site/regions
Q02 /site/closed_auctions
Q03 /site/regions/europe/item/mailbox/mail/text/keyword
Q04 /site/closed_auctions/closed_auction/annotation/description/parlist/listitem
Q05 /site/closed_auctions/closed_auction/annotation/description/parlist/listitem/parlist/listitem/*//keyword
Q06 /site/regions/*/item
Q07 //listitem//keyword
Q08 /site/regions/*/item//keyword
Q09 /site/regions/*/person[address and (phone or homepage)]
Q10 //listitem[.//keyword and .//emph]//parlist
Q11 /site/regions/*/item[mailbox/mail/date]/mailbox/mail
Q12 /*[descendant::*]
Q13 //*
Q14 //*//*
Q15 //*//*//*//*
Q16 //*//*//*//*//*//*//*

Fig. 5. Tree oriented queries

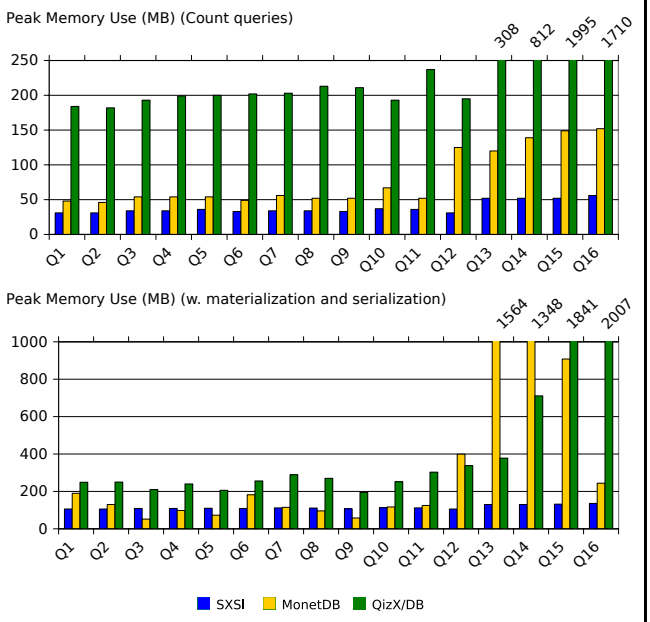


Fig. 7. Peak memory use of the three engines (116 MB XMark file)

text representation, is always smaller than twice the size of the original document. Since it is always possible to choose which text representation to use, the actual main-memory footprint of the index is close to the original document size.

C. Tree queries

We benchmarked tree queries using the queries given in Figure 5. Queries Q01 to Q11 were taken from the XPathMark benchmark [35], derived from the XMark XQuery benchmark suite. Q12 to Q16 are "crash tests" that are either simple (Q12 selects only the root since it always has at least one descendant in our files) or generate the same amount of results but with various intermediate result sizes. For this experiment we used XMark documents of size 116MB and 1GB. In the cases of MonetDB and Qizx, the files were indexed using the default settings. Figure 6 reports the running times for both counting and materialization+serialization. We report in Figure 7 the peak memory use for each query, for the 116MB document.

From the results of Figure 6, we see how the different components of SXSI contribute to the efficient evaluation model. First, queries Q01 to Q06 —which are fully qualified paths— 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. Query Q07 to Q11 illustrate the impact of the jumping. Moreover, it shows that filters do not impact the execution speed: the conditions they express are efficiently checked by the formula evaluation procedure. Finally, Q12 to Q16 illustrate the robustness of our automata model. Indeed while such queries might seem unrealistic, the good performances that we obtain are only the consequence of using an automata model, which factors in its states all the necessary computation and thus do not materialize uneeded intermediate results. This coupled together with the compact dynamic set of Section IV-D allows us to keep a very low memory foot-print even when the query generates lots of result or that each step as a lot of intermediate results (cf. Figure 7).

D. Text queries

We tested the text capabilities of our XPath engine against the most advanced text oriented features of other query engine.

Qizx/DB: We used the newly introduced *Full-Text* extension of XQuery available in Qizx/DB v. 3.0. We tried to write queries as efficiently as possible while preserving the same semantics as our original queries. The query we used always gave better results than their pure XPath counterpart. In particular, we used the `ftcontains` text predicate, introduced in [36] and implemented by Qizx/DB. The `ftcontains` predicate allows one to express not only *contains*-like queries but also Boolean operations on text predicates, regular expression

	Q01	Q02	Q03	Q04	Q05	Q06	Q07	Q08	Q09	Q10	Q11	Q12	Q13	Q14	Q15	Q16
116 MB Document, counting																
SXSI	**1**	**1**	**14**	**16**	**24**	**12**	36	31	**5**	70	**34**	**1**	309	309	**313**	**330**
Monet	7	7	28	24	40	16	24	**30**	87	**61**	60	183	**75**	**239**	597	957
Qizx	**1**	**1**	26	29	31	17	**19**	39	48	109	158	**1**	2090	8804	28005	34800
116 MB Document, materialising and serializing																
SXSI	**1**	**1**	**15**	**21**	**26**	120	64	65	**5**	**83**	**52**	**1**	974	975	987	**465**
	198	66	7	36	7	256	74	85	0.1	43	96	566	5847	5295	4076	573
Monet	7	7	28	27	40	**16**	**25**	**25**	29	88	60	179	**71**	**238**	**591**	966
	672	208	10	76	10	671	90	81	0.1	104	181	1653	10023	8288	4959	667
Qizx	3153	1260	65	567	103	3487	1029	307	50	991	1179	8387	45157	44264	8181	21680
1 GB Document, counting																
SXSI	2	2	**107**	149	**207**	79	665	342	**5**	990	317	**2**	**4376**	**4371**	**4382**	**4500**
Monet	8	8	519	576	597	1557	3383	1623	1557	3719	1799	16274	7779	25493	60555	77337
Qizx	**1**	**1**	185	**135**	230	**45**	**101**	**302**	291	**185**	**186**	14	17368	++	++	++
1 GB Document, materialising and serializing																
SXSI	**2**	**2**	**140**	**238**	**256**	**1110**	**1654**	**771**	**5**	**1372**	**543**	**2**	**15246**	**15254**	**15461**	**6567**
	1920	637	74	359	69	2488	727	835	0.1	411	927	5413	57880	51915	40103	5662
Monet	8	8	587	617	648	1554	3405	1710	1600	3739	1810	18203	⋆	⋆	⋆	80394
	20999	200770	22586	158548	37469	11740	53067	16360	0.1	43688	16882	26858	⋆	⋆	⋆	31818
Qizx	29998	9363	368	4517	417	29543	9061	1989	317	8452	9424	74843	414086	⋆⋆	⋆⋆	⋆⋆

++: Running time exceeded 20 minutes ⋆: MonetDB server ran out of memory. ⋆⋆: Qizx/DB ran out of memory.

We mark in **bold face** the fastest query execution time and we underline the fastest execution and serialization time.

Fig. 6. Running time for the tree based queries (in milliseconds)

matching and so on. It is more efficient than the standard `contains`. In particular we used regular expression matching *in lieu* of the `starts-with` and `ends-with` operators since the latter were slower in our experiments.

MonetDB: MonetDB supports some full-text capabilities through the use of the PF/Tijah text index ([37]). While more efficient than using the built-in string functions, the set of queries expressible with this index is quite limited. The tree navigation part is limited to the `descendant` and `self` axes, while the only text predicate available is `about`, which allows selecting nodes which are "relevant" with respect to a given string. We used it to express `contains` and used the built-in string functions for other queries.

Experiments were made against a 122MB Medline file. This file contains bibliographic information about life of sciences and bio medical publications. This test file featured 5,732,159 text elements, for a total amount of 95MB of text content. Figure 8 shows the text queries we tested. We used count queries for both MonetDB and Qizx —enclosing the query in a `fn:count()` predicate— while in our implementation we ran the queries in "materialization" mode but without serializing the output. The table in Figure 9 summarizes the running times for each query. As we target very selective text queries, we also give, for each query, the number of results it returned. Since for these queries our automata worked in "bottom-up" mode, we detail the two following operations:

- Calling the text predicate *globally* on the text collection, thus retrieving all the probable matches of the query (*Text query* line in the table of Figure 9)
- Running the automaton bottom up from the set of probable matches to keep those satisfying the path expression (*Auto. run* line in the table of Figure 9)

```
T1  //MedlineCitation//*/text()[contains( ., "brain")]
T2  //MedlineCitation//Country/text()[
                        contains(., "AUSTRALIA")]
T3  //Country/text()[ contains(. , "AUSTRALIA")]
T4  //*/text()[ contains( . , "1930")]
T5  //MedlineCitation//*/text()[ contains( . , "1930") ]
T6  //MedlineCitation/Article/AuthorList/Author/
                LastName/text()[startswith(., "Bar")]
T7  //MedlineCitation[ MedlineJournalInfo/
            Country/text()[ ends-with(.,"LAND")]]
T8  //*[ Year = "2001"]
T9  //*[ LastName = "Nguyen"]
```

Fig. 8. Text oriented queries

As it is clear from the experiments the bottom-up strategy pays off. The only down-side of this approach is that the automaton uses Parent moves, which are less efficient than FirstChild and NextSibling. This is clear in queries T7 and T8 where the increase in number of results makes the relative slowness of the automata more visible. However our evaluator still outperforms the other engines even in those cases.

E. Remarks

We also compared with Tauro [3]. Yet, as it uses a tailored query language, we could not produce comparable results.

We limited the experiments to natural language XML, although our engine (unlike the inverted file -based engines) supports as well queries on XML databases of continuous sequences such as DNA and proteins. Realistic queries on such

	T1	T2	T3	T4	T5	T6	T7	T8	T9
Text query	10	4	4	0.4	0.4	9	6	59	0.7
Auto. run	28	8	3	0.9	1.5	17	108	96	2
SXSI: Total	**38**	12	7	**1.3**	**1.9**	**26**	**144**	**175**	**2.7**
MonetDB	336	118	117	252	301	180	256	473	505
Qizx/DB	108	**10**	**6**	99	107	244	259	2469	1397
# of results	1493	438	438	32	32	680	6935	6685	36

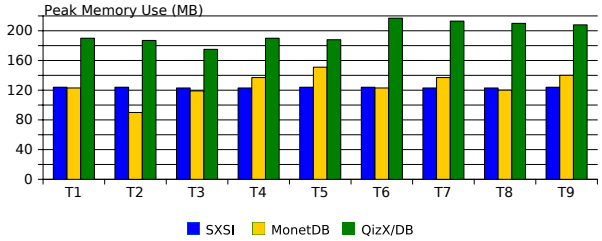


Fig. 9. Running times and memory consumption for the text-oriented queries

biosequence XMLs require approximate / regular expression search functionalities, that we already support but whose experimental study is out of the scope of this paper.

VII. Conclusions and Future Work

We have presented SXSI, a system for representing an XML collection in compact form so that fast indexed XPath queries can be carried out on it. Even in its current prototype stage, SXSI is already competitive with well-known efficient systems such as MonetDB and Quizx. As such, a number of avenues for future work are open. We mention the broadest ones here.

Handling updates to the collections is possible in principle, as there are dynamic data structures for sequences, trees, and text collections [7], [8], [13]. What remains to be verified is how practical can those theoretical solutions be made.

As seen, the compact data structures 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 structure *Doc* of Section III.

A clear direction for future work is to extend the current system to support XQuery operations. Even within full XPath 1.0 there are very sophisticated primitives such as data joins, which would be challenging to support efficiently.

References

[1] XML Mind products, "Qizx XML query engine," http://www.xmlmind.com/qizx, 2007.

[2] 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*, 2006, pp. 479–490.

[3] Signum, "Tauro," http://tauro.signum.sns.it/, 2008.

[4] M. Kay, "Ten reasons why Saxon XQuery is fast," *IEEE Data Eng. Bull.*, vol. 31, no. 4, pp. 65–74, 2008.

[5] M. F. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur, "Implementing XQuery 1.0: The Galax experience," in *VLDB*, 2003, pp. 1077–1080.

[6] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Comp. Surv.*, vol. 39, p. article 2, 2007.

[7] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane, "Compressed indexes for dynamic text collections," *ACM TALG*, vol. 3, 2007.

[8] V. Mäkinen and G. Navarro, "Dynamic entropy-compressed sequences and full-text indexes," *ACM TALG*, vol. 4, 2008.

[9] G. Jacobson, "Space-efficient static trees and graphs," in *FOCS*, 1989, pp. 549–554.

[10] J. Munro and V. Raman, "Succinct representation of balanced parentheses and static trees," *SIAM J. Comp.*, vol. 31, pp. 762–776, 2001.

[11] R. Geary, N. Rahman, R. Raman, and V. Raman, "A simple optimal representation for balanced parentheses." in *CPM*, 2004, pp. 159–172.

[12] D. Benoit, E. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao, "Representing trees of higher degree," *Algorithmica*, vol. 43, no. 4, pp. 275–292, 2005.

[13] K. Sadakane and G. Navarro, "Fully-functional static and dynamic succinct trees," arXiv, Tech. Rep. 0905.0768v2, 2008.

[14] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan, "Structuring labeled trees for optimal succinctness, and beyond," in *FOCS*, 2005, pp. 184–196.

[15] ——, "Compressing and searching XML data via two zips," in *WWW*, 2006, pp. 751–760.

[16] G. Gottlob, C. Koch, and R. Pichler, "Efficient algorithms for processing XPath queries," *ACM TODS*, vol. 30, no. 2, pp. 444–491, 2005.

[17] G. Manzini, "An analysis of the Burrows-Wheeler transform," *J. ACM*, vol. 48, no. 3, pp. 407–430, 2001.

[18] P. Ferragina and G. Manzini, "Indexing compressed text," *J. ACM*, vol. 54, no. 4, pp. 552–581, 2005.

[19] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "Compressed representations of sequences and full-text indexes," *ACM TALG*, vol. 3, 2007.

[20] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm." Digital Equipment Corporation, Tech. Rep. 124, 1994.

[21] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *SODA*, 2003, pp. 841–850.

[22] V. Mäkinen and G. Navarro, "Rank and select revisited and extended," *Theor. Comput. Sci.*, vol. 387, no. 3, pp. 332–347, 2007.

[23] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu, "Compressed indexing and local alignment of DNA," *Bioinformatics*, vol. 24, no. 6, pp. 791–797, 2008.

[24] 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*, vol. 10, no. 3, 2009, R25.

[25] H. Li and R. Durbin, "Fast and accurate short read alignment with burrows-wheeler transform," *Bioinformatics*, 2009, advance access.

[26] J. Sirén, "Compressed suffix arrays for massive data," in *SPIRE*, 2009, to appear.

[27] D. Arroyuelo, G. Navarro, and K. Sadakane, "Reducing the space requirement of LZ-index," in *CPM*, 2006, pp. 319–330.

[28] I. Munro and V. Raman, "Succinct representation of balanced parentheses, static trees and planar graphs," in *FOCS*, 1997, pp. 118–126.

[29] F. Claude and G. Navarro, "Practical rank/select queries over arbitrary sequences," in *SPIRE*, 2008, pp. 176–187.

[30] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary," in *ALENEX*, 2007.

[31] H. Hosoya, "Foundations of XML processing," in preparation, see http://arbre.is.s.u-tokyo.ac.jp/hahosoya/xmlbook/.

[32] 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.

[33] 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.

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

[35] M. Franceschet, "XPathMark: Functional and performance tests for XPath," in *XQuery Implementation Paradigms*, 2007, http://drops.dagstuhl.de/opus/volltexte/2007/892.

[36] "XQuery and XPath Full Text 1.0," http://www.w3.org/TR/xpath-full-text-10.

[37] H. E. Blok, V. Mihajlovic, G. Ramírez, T. Westerveld, D. Hiemstra, and A. P. de Vries, "The tijah XML information retrieval system," in *SIGIR*, 2006, p. 725.