Fast and Tiny Structural Self-Indexes for XML

Sebastian Maneth  
NICTA and UNSW  
Sydney, Australia  
sebastian.maneth@nicta.com.au

Tom Sebastian  
NICTA  
Sydney, Australia  
tom.sebastian@nicta.com.au

ABSTRACT

XML document markup is highly repetitive and therefore well compressible using dictionary-based methods such as DAGs or grammars. In the context of selectivity estimation, grammar-compressed trees were used before as synopsis for structural XPath queries. Here a fully-fledged index over such grammars is presented. The index allows to execute arbitrary tree algorithms with a slow-down that is comparable to the space improvement. More interestingly, certain algorithms execute much faster over the index (because no decompression occurs). E.g., for structural XPath count queries, evaluating over the index is faster than previous XPath implementations, often by two orders of magnitude. The index also allows to serialize XML results (including texts) faster than previous systems, by a factor of ca. 2–3. This is due to efficient copy handling of grammar repetitions, and because materialization is totally avoided. In order to compare with twig join implementations, we implemented a materializer which writes out pre-order numbers of result nodes, and show its competitiveness.

1. INTRODUCTION

An important task in XML processing is the evaluation of XPath queries. Such queries select nodes of an XML document and are used in many scenarios: embedded in larger XQueries, in XSL stylesheets, in XML policy specifications, in JavaScripts, etc. A common way of speeding up query evaluation is to use indexes. But conventional value indexes for XML tags and text values are not sufficient to answer XPath queries, because they do not capture the document’s hierarchical structure. Therefore a large number of structural XML indexes have been introduced (see 12 for a recent overview). The first one was the DataGuide 13. It stores a summary of all distinct paths of a document. Later the finer 1-index was proposed 25 which is based on node bisimulation. For certain XPath queries these indexes allow evaluation without previous access to the original data; e.g., for structural queries restricted to the child and descendant axes. More fine-grained structural indexes were considered but turned out to be too large in practice, see 17. As a compromise, the A(k)-index 18 was proposed which uses node bisimilarity of paths up to length k; the D(k) 28 and M(k)-

indexes 16 are A(k)-variants that adapt to query workloads. Updates for the A(k) and 1-indexes were studied in 33. Index path selection is considered e.g., in 29; but, their indexes are usually larger than the original documents (including data values). All indexes mentioned so far are approximative for full structural XPath, i.e., do not capture enough information to evaluate XPath’s twelve navigational axes. This is in contrast to the indexes introduced here.

A self index has the property that (1) it allows to speed up certain accesses, and (2) it can reproduce the original data (which therefore can be discarded after index construction). Moreover, such indexes are often based on compression and hence are small (typically smaller than the original data). For Claude and Navarro 7 a self-index for text must (at least) efficiently support the extract and find operations; these operations reproduce a portion of the text and find all occurrences of a substring, respectively. In XPath processing, more complex search than finding substrings is required. In fact, XPath search is comparable to regular expression search. Unfortunately, even for text, little is known about indexes that support arbitrary regular expression search (see, e.g., 2).

In 1, 24 it was observed that two particular navigational operations allow drastic speed-ups for XPath evaluation: taggedDesc and taggedFoll. Given a node and a label, these operations return the first descendant node and first following node with that label, respectively. During XPath evaluation these operations allow to jump to next relevant nodes; this cuts down the number of intermediate nodes to be considered during evaluation. The “QName thread” in MTree 27 is similar in spirit (it allows to jump to next descendants with a given label).

The idea of our new index is to use grammar-compressed trees (which typically are much smaller than succinct trees 23) and to add small data structures on top of these which support efficient taggedDesc and taggedFoll. Our contributions are

1. a self-index for trees, based on grammar-based compression
2. a generic sequential interface that allows to execute, over the new index, arbitrary algorithms on the original tree
3. a special evaluator for counting of XPath query results, and
4. special evaluators for serializing and materializing of XPath query results.

We tested the generic interface of Point 2 on two algorithms: on depth-first left-to-right (dflr) recursive and iterative full tree traversals, and, on the (recursive) XPath evaluator “SXSI” of 1. We obtain good time/space trade-offs. For instance, replacing SXSI’s tree store with our interface of Point 2 gives a slow-down of factor 4 while it slashes SXSI’s memory use by factor 3 (averaged over the 16 tree queries of 1 on a 116M XMark file). Our experiments
show that the evaluators of Points 3 and 4 are faster than existing XPath implementations, often by two orders of magnitude. Note that the indexes used by these evaluators are so tiny in space (see Figure 1) that any XML database can profit from them, by conveniently keeping them in memory. This allows, besides others, fast serialization and fast XPath selectivity computation, and therefore can replace structural synopses.

|       | XMark | Treebank | Sprot |
|-------|-------|----------|-------|
| Count | 116MB | 1GB      | 116MB | 83MB  | 437MB  |
| Mat. / Ser. | 1.1 | 5.5 | 7.9 | 5.5 | 1.3 |
|       | 1.7 | 1.9 | 11.6 | 5.5 | 2.5 |

Figure 1: TinyT Index Sizes (in MB)

While the generic interface causes a slow-down due to decompression, there are classes of algorithms (over the grammar) which allow considerable speed-ups. Essentially, the speed-ups are proportional to the compression ratio, because the compressed grammar need only be traversed once. For instance, tree automata and Core XPath can be evaluated in one pass over straight-line tree (SLT) grammars [20]. This idea was used in [11] for selectivity estimation of structural XPath. They study synopsis size and accuracy, but do not consider efficient evaluation. We combine the ideas of [24] with that of evaluating in one pass over the grammar. To this end, we augment the grammar with information that allows efficient taggedDesc and taggedFoll: for every nonterminal \( \tau \) and terminal symbol \( t \) of the grammar a bit is stored that determines whether \( \tau \) generates \( t \). If \( \tau \) does not generate \( t \), then it may be “jumped” during a taggedDesc call for \( t \). Our first structural index comprises this “jump table”, together with a compact representation of the grammar. The XPath count evaluator of Point 3 executes over this index. To obtain grammars from XML structure trees, we use the new TreeRePair compressor [21]. Due to compression, the resulting indexes are phenomenally small. For instance (cf. Figure 1), our index can store the half a billion nodes of an 11GB XMark tree in only 8MB! This means an astonishing 8.7 nodes per bit! Consequently, our evaluator (which, due to jumping, need not even visit the whole grammar) is extremely fast. Compared to the fastest known evaluators, MonetDB [4] and Qizx [32], we found that our XPath count evaluator is faster by 1–2 orders of magnitude, for essentially all queries we tested.

Motivated by our positive results from Point 3, the question arose whether the count evaluator can be extended to handle proper XPath semantics, i.e., to output XML subtrees of result nodes. Since serialization involves outputting of data values, all data values are now stored in a memory buffer. Additionally, a data structure is built that links the SLT grammar to the correct data values. This is achieved by storing for each nonterminal the number of text-values that it generates. In fact, since a nonterminal generates a tree pattern which has many “dangling subtrees”, we need to store tuples of such numbers: the first component is the number of text-values in the first “chunk” of the nonterminal, i.e., in the tag sequence (of the nonterminal) before the first dangling subtree; next is the number of text-values in the second chunk, i.e., between the first and second dangling subtrees, etc. Evaluation is still done in one pass through the grammar, but, this time must follow a strict different traversal (which makes it slower than for count queries). A nice bonus is the possibility to make clever use of hashing: we remember the “chunks” of XML markup produced by each nonterminal. This greatly speeds up serialization. Moreover, it turned out that materialization of result nodes can be totally avoided. Thus, neither expensive grammar node IDs need to be stored, nor their translation to pre-order numbers is needed. Rather, whenever a result node is encountered during evaluation, we start a serialization process which works in parallel with evaluation. The resulting system outperforms by a factor of 2–3 the fastest known system SXXI (which on its own outperforms MonetDB and Qizx, see [1]).

About the comparison: it can be argued that comparing our rudimentary XPath evaluator with full-blown XML databases is unfair, because these larger systems have more overhead (such as locking, transaction handling, updates). On the other hand, these systems are highly optimized and therefore could exploit their best available algorithm for simple queries. We therefore believe that the comparison is relevant. Note that we also compare with specialized implementations which handle smaller or incomparable XPath fragments. For instance, we compared to the fastest available implementations of twig joins [14, 15]. Since these algorithms materialize result nodes, we implemented an experimental materializer (Point 4). Interestingly, it often outperforms these twig implementations (which represent state-of-the-art of many years of research on twigs). We also compare to the index of [9, 10] which handles simple paths (XPaths with one // followed by /’s); our experiments show that for selective queries this index is faster than ours (by a factor of 10–20), while for non-selective queries our index is faster.

Related Work. Compression by SLT grammars was used in [11] for selectivity estimation of structural XPath. They study the space efficiency of binary encoded grammars with respect to other XML synopses, but do not study run times. It is also shows that updates can be handled incrementally with little space overhead; this is important also for our work, because we would like to support incremental updates in the future. The minimal DAGs used by Koch et al. [5] can be seen as the first grammar-compressed approach to XML trees (a DAG naturally corresponds to a regular tree grammar). For usual XML document trees, minimal DAGs only exhibit 10% of the original number of edges. More powerful grammar-compressors such as BPLEX [6] further reduce this number to 5% and the recently introduced TreeRePair [21] to less than 3%. An SLT grammar generalizes DAGs from sharing of repeated subtrees to sharing of repeated tree patterns (connected subgraphs of the tree). They are equivalent to the sharing graphs used by Lamp for optimal lambda calculus evaluation [19]. A self-index for grammar-compressed strings was presented in [7]. They show efficient support for extract and find. It can be shown, but goes beyond the scope of this paper, that the extract operation can be generalized from their string grammars to our SLT grammar, with the same time bounds as in their result. In [1] they use the succinct tree data structures of [31] and add explicit copies for each label, using compressed bit-arrays [26]. This allows constant time access to taggedDesc and taggedFoll (using rank and select over bit-arrays), but becomes fairly memory heavy (for a 116M XMark document with 6 million nodes, they need 8MB for the tree, and additional 18MB to support taggedDesc and taggedFoll in constant time).

2. XML TREE COMPRESSION

An XML document naturally corresponds to an unranked ordered tree. For simplicity, we only focus on element nodes, attributes, and text values, and omit namespaces, processing instructions, and comments. Our data model assumes that the attribute and text values are stored separately from the tree structure (in a “text collection”), and that they can be addressed by a function getText(n) that returns the n-th text or attribute value (in pre-order appearance). In our tree model, text nodes of the document are represented by placeholder leaf nodes labeled with the special label _T. Similarly, attribute definitions are represented by “attribute place-
holder nodes” labeled _A; such a node has children nodes which are labeled by the names of the attributes (prepended by the symbol “@”) in their appearance order, which themselves have a single “attribute-text placeholder node” (labeled _AT). For instance the XML element <name id="9" re="4">Text</name> is represented, in term syntax, by this tree: name(_A@id(_AT),@r(_AT)),_T). For a given XML document, such a tree is called its XML structure tree. Obviously, these trees are larger than pure element-

| Name          | Element Count | Max Depth | Non-Text (MB) |
|---------------|---------------|-----------|---------------|
| XMark116M     | 1,735,083     | 12        | 34            |
| XMark1G       | 16,703,210    | 13        | 325           |
| XMark11G      | 167,095,844   | 13        | 3246          |
| Treebank83M   | 2,437,666     | 22        | 25            |
| Sprot437M     | 10,903,568    | 38        | 154           |

**Figure 2:** Datasets used in experiments

| Name          | Element Count | Size (MB) |
|---------------|---------------|-----------|
| XMark116M     | 6,074,297     | 59        |
| XMark1G       | 58,472,941    | 559       |
| XMark11G      | 588,961,650   | 5579      |
| Treebank83M   | 7,312,615     | 48        |
| Sprot437M     | 27,035,515    | 231       |

**Figure 3:** Sizes of XML structure trees

In an SLT grammar, sharing is not restricted to subtrees, but arbitrary tree patterns (connected subgraphs) can be shared. In the example tree t of above, the tree pattern consisting of an f-node and right subtree a(c, c) appears twice. As we can see, this tree pattern has one “dangling edge”, namely, to the second-child of the f-node. In SLT grammar notation, a tree pattern is written as a tree in which special placeholders, called parameters, are inserted at dangling edge positions. The parameters are denoted y1, y2, ..., and are numbered in the order of appearance of dangling edges. An SLT grammar that represents t has these productions:

\[ S \rightarrow A(A(a(b, c))) \]
\[ A(y) \rightarrow f(f_1, a(c, c)) \]

The nonterminal A uses one parameter y1 for the single dangling edge of the pattern mentioned above. The size of this grammar is still 8. The number of parameters of a nonterminal A is called its rank and is denoted rank(A). The maximal number of parameters of the nonterminals of a grammar is called the rank of the grammar. Another important aspect of a grammar is its depth, which is the length of the longest sequence of nonterminals \( A_1, A_2, \ldots, A_d \) such that \( A_{i+1} \) appears in the right-hand side of \( A_i \), for all \( 1 \leq i < d \). Since all our grammars produce one tree only, the depth is bounded by the number of nonterminals. Given a nonterminal A (of rank \( k \)), its pattern tree, denoted \( t_A \), is the tree over terminal symbols and parameters \( y_1, \ldots, y_k \), obtained from \( A(y_1, \ldots, y_k) \) by applying grammar productions (until no production can be applied anymore).

While the minimal DAG of a tree is unique and can be found in linear time, the minimal SLT grammar is not unique, and finding one is NP-complete [6]. The BPLEX approximation algorithm [6] generates SLT grammars that are ca. half the size of the minimal DAG. The new TreeRePair algorithm [21] improves this by another 20%–30% (while improving run time by a factor of about 30). The above example grammar for \( t \) was produced by TreeRePair. Note that the rank of a grammar is important, because it influences the run-time of algorithms that directly execute on the grammar, such as executing tree automata or Core XPath [20]. Both BPLEX and TreeRePair take a user specified “maximal rank number \( m \)”, and produce grammars of rank \( \leq m \).

3. STRUCTURAL SELF-INDEX

We call our XML self-index “TinyTree” or simply “TinyT”. The first layer of storage in TinyT consists of a small memory representation of the grammar. The second layer consists of additional mappings that support fast XPath evaluation.

3.1 Base Index

The base index consists of a small memory representation of the grammar. Start production right-hand sides are usually large trees (they represent the incompressible part of the XML structure tree) and are coded succinctly, using two alternative ways. All other productions are transformed into a normal form, so that each resulting production fits into a single 64-bit machine word. We experimented with two variants of representing the start rhs:

- (bp) the succinct trees of Sadakane and Navarro [31] (ex) a naive custom representation.

Both of these use \( s \lceil \log \sigma \rceil \) many bits to represent the tag sequence of the tree, where \( s \) is the number of nodes in the start rhs. The first one uses the “moderate size” trees of [31], requiring additional \( 2s + O(s/polylog(s)) \) bits of space. Our implementation of (bp) uses approximately 2.5 bits per node. The second one (ex) stores an
explicit mapping called “find-close” which for every node records the number of nodes in its subtree. This is sufficient for our XPath evaluators because they only need pre-order access to the grammar, plus, the ability to “skip” a subtree. The find-close table allows to skip a subtree, by simply moving ahead in the tag-list by the number of nodes specified in the table. It requires \( s \lfloor \log s \rfloor \) bits. Clearly, this is rather wasteful compared to (bp), see the third column in Figure 4, but can make a large speed difference: e.g., our XPath count evaluator for the query \( Q_{06} = \text{site/regions/*item} \) over XMark1G takes 3.5ms with (ex) and 4.7ms with (bp). Observe also the difference in loading time of the two variants shown in Figure 6.

We bring the remaining productions into binary Chomsky Normal Form (bCNF). A production is in bCNF if it contains exactly two non-parameter nodes in its right-hand side. The bCNF can be obtained following exactly the same procedure as for ordinary CNF, see [22]. A grammar is in bCNF, if every production except the start production is in bCNF. For instance, in our example cft grammar of before, the \( A \)-production is not in bCNF. We first change its right-hand side to \( f(y_1, B) \) (which is in bCNF) and add the new production \( B \rightarrow a(c, c) \). The latter is not in bCNF and therefore is changed to \( B \rightarrow C(c) \). The final grammar, called \( G_1 \), is:

\[
\begin{align*}
    S & \rightarrow A(\alpha(b, c)) \\
    A(y_1) & \rightarrow f(y_1, B) \\
    B & \rightarrow C(c) \\
    C(y_1) & \rightarrow a(c, c)
\end{align*}
\]

Note that the size of this grammar is 9, thus has grown by one. In general, the size of a grammar can grow by a factor \( r \), where \( r \) is the rank of the original grammar. The rank of the grammar can grow by \( \max(r, 1) \), and the number of nonterminals can become at most two times the size of the original grammar, as implied by Proposition 3 of [22]. If we transform the DAG grammar for \( t \) of before into bCNF, then a grammar is obtained of rank one and of size 9; consider \( t' = f(t, a(c, c)) \), then the minimal DAG in bCNF is of size 11 (because two edges are added in the start rhs), while our cft grammar has size 10 (we simply add another \( A \)-node in the start production).

In practice, we do not observe a large size increase; the largest was 79\%, see the last column of Figure 4.

**Figure 4: Impact of bCNF**

| size (start-rhs) | #rules (before) | #rules bCNF | depth (before) | depth bCNF | size-diff (in %) |
|------------------|-----------------|-------------|----------------|-------------|-----------------|
| XMark16M         | 88299           | 10738       | 36             | 361         | 21%             |
| XMark1G          | 64313           | 31684       | 284944         | 10381       | 79%             |
| XMark1G          | 105893          | 62604       | 408485         | 3915        | 70%             |
| Treebank83M      | 470568          | 35352       | 37540          | 20          | 0.1%            |
| Sprot437M        | 246970          | 20410       | 23484          | 128         | 0.3%            |

...more entries...

Increase can be large as shown in the figure. The rhs of each bCNF rule if of the form \( X(y_1, \ldots, y_{t-1}, Y(y_1, \ldots, y_r)) \) and thus is characterized by the triple \((X, i, Y)\), where \( X \) and \( Y \) are nonterminals or terminals, and \( i \) is a number between 1 and the rank of \( X \). We represent one bCNF rule by a single 64-bit machine word, using 28 bits per nonterminal, 4 bits for the number \( i \), and 4 bits for the rank of the nonterminal. Our experiments show that setting the maximal rank of BPLEX and TreeRepPair to 8 and 2, respectively, gave best results for our XPath evaluators over the corresponding indexes. Thus, limiting our memory representation to grammars of rank 15 (4 bits) is justified. We are now ready to calculate the space requirement of the grammar representation: \#CNF-rules \times 8\,\text{Bytes} + \text{space}(start-rhs).

As an example, for XMark116M we calculate, according to Figure 4, 39631 productions in bCNF, multiplied by 8 bytes equals 309.6KB. For the tag sequence of the start rhs we need \( 88299 \cdot \log 39631 + 89 \approx 172.5\,\text{KB} \) (there are 89 different labels for XMark). For (bp) our implementation uses 27KB, while (ex) uses 183.2KB. Thus, the total sizes for (bp) and (ex) are, 509KB and 665KB, respectively (the sum of the first three columns in Figure 5 up to rounding).

**Figure 5: Sizes of TinyT’s components (in KB)**

### 3.2 Auxiliary Indexes

There are two well-known principles of XPath optimization: (1) jumping and (2) skipping. Here, jump means to omit internal nodes of the document tree. In our setting, the “jumped” nodes will be those represented by a nonterminal of our grammar. We also say that the nonterminal is “jumped”. Note that after a jump we still need to continue evaluating in the subtrees below the jumped pattern. Skipping means to omit a complete subtree. Thus, no evaluation is needed below skipped nodes. We now introduce the jump table which allows to jump nonterminals; this table suffices for our XPath count evaluator. To jump or skip during serialization and materialization we need further tables (the “pre and text mappings”). Lastly, we mention the start-skip table which supports fast skipping of subtrees.

**Jump Table**

As mentioned in the Introduction, it was observed in [1, 24] that the two operations taggedDesc and taggedFoll allow drastic speed-ups for XPath evaluation. The SXSI system [24] keeps a large data structure (about 2.25 times larger than the rest of their tree store) in order to give constant time access to these operations. We try to add very little extra space to our (so far tiny) index, and still be able to profit from these functions. We build a “jump table” which keeps for every nonterminal \( X \) and every terminal symbol \( b \) a bit indicating whether or not \( X \) generates a \( b \)-labeled node. When executing a taggedDesc-call (with label \( b \)), we try to derive the first descendant node with label \( b \); if a nonterminal during this derivation does not generate \( b \)'s (according to our jump table), then we do not expand it, but “jump” it (by moving to its first parameter position).

The taggedFoll function is realized similarly. For the sequential interface (Point 2 in the Introduction) plugged into SXSI [1] our experiments show that the speed-up through taggedDesc/taggedFoll is comparable to the speed-up obtained in SXSI. This is surprising, because the space overhead for our jump table is small: 65\% of extra space, compared to the 225\% in SXSI.

The jump table is not only useful to realize taggedDesc and taggedFoll, but also allows speed-ups in all our XPath evaluators, see e.g. \( q_1 \) in Figure 15. The size of the jump table (in bits) is the number of nonterminals multiplied by the number of different (terminal) labels. For instance, XMark uses 89 labels; thus, the jump table for XMark116M is 39631 * 89 bits = 431KB. For Treebank83M which has 257 labels we obtain 37540 * 257 bits = 1177.7KB, see the fourth column in Figure 5. In fact, our XPath count evaluator only loads the base index plus the jump table, which implies the total index sizes as shown in Figure 1 as sum of the first four columns in Figure 5 (taking "ex").
Pre and Text Mappings

In order to be able to materialize pre-order node numbers, or to access the text collection (needed for serialization), we need to calculate, during evaluation, the numbers of nodes/texts that have appeared until the current node following a dflr traversal. However, if we “jump” a nonterminal using our jump table, then we do not see its terminals. Therefore we need another table which records for each nonterminal the number of element nodes that it generates, and similarly for the number of text nodes. In fact, the situation is more complicated: we actually need to store several numbers per nonterminal, as many as the rank of that nonterminal, plus one. With respect to evaluation in dflr order, jumping a nonterminal means to move to its first parameter position and continue evaluation there. Thus, we must know how many element symbols are on the path from \( t_X \)'s root to the first parameter, where \( t_X \) is the tree generated by \( X \); note that \( t_X \) contains exactly one occurrence of each parameter of \( X \). Similarly, once returned from \( X \)’s first parameter position, we will want to jump to the second parameter. We thus need to know the number of element nodes that are on the path between \( y_1 \) and \( y_2 \) in the tree \( t_X \). The size of the corresponding table “prMap” is \( \sum_{X \in NT} \log(k) \), where \( k \) is the maximal number of element nodes on such paths, for any nonterminal. In fact, in our implementation we simply use a 4-Byte integer per value. For instance, our grammar for XMark116M has 14057 nonterminals of rank zero, 14475 of rank one, 9311 of rank two, and 1768 of rank three. The size of the resulting prMap is \( (14057 + 14475 + 2 + 9311 + 3 + 1768 + 4 = 78084) + 4B = 305KB \); this explains the column “prMap” in Figure 5. The corresponding table with numbers of text nodes is called “textMap” table.

Start-Skip Table

If, during materializing or serializing we want to skip a subtree, then we still need to traverse that subtree of the grammar, in order to sum all numbers of element nodes/texts, respectively (using the pr and text mappings). To short-cut this calculation, the start-skip table is added. It stores for every node of the start rhs, the total number of element nodes/texts in its subtree. The size of this table is the number of nodes in the start rhs multiplied by \( \log(n) \), where \( n \) is the total number of element nodes/texts. In our implementation we simply use 4 bytes per such number. The corresponding table for the numbers of text nodes is called “textSSkip”.

3.3 Index Generation

The generation of the base index consists of the following steps

1. generate XML structure tree (MakeSTree),
2. compress via TreeRePair,
3. transform into bCNF, and
4. build in-memory representation of TinyT components and save to file (BuildTinyT).

Technically speaking, Steps (1) and (3) are not necessary but can be incorporated into the TreeRePair compressor. In Step (1) we merely replace all text and attribute values by placeholder nodes. This can be incorporated into the parsing process of TreeRePair. Similarly, TreeRePair can be changed so that it produces grammars that are already in bCNF. Since we also wanted to experiment with other compressors such as DAG and BPLEX, we implemented small programs for (1) and (3). Our program for (1) is a naive java implementation using SAX which is quite inefficient. Therefore the times for MakeSTree in Figure 6 should be ignored and the table should be read as: indexing time is dominated by grammar compression time. The times in Figure 6 for step (4) are for generating the base plus the jump index, i.e., the first four columns of Figure 5.

The time for generating the two additional tables prMap and SSkip (columns 5 and 6 in Figure 5) is negligible, as it is is proportional to a “chunk-wise” traversal of the grammar (see Sections 4.3 and 6.1).

|                  | XMark116M | XMark1G | XMark11G | Treebank83M |
|------------------|-----------|---------|----------|-------------|
| MakeSTree        | 0:26      | 3:45    | 38:38    | 0:14        |
| TreeRePair       | 0:33      | 8:58    | 52:10    | 1:14        |
| bCNF             | 0:01      | 0:08    | 0:07     | 0:03        |
| BuildTinyT       | 0:01      | 0:04    | 0:06     | 0:02        |
| Total            | 1:01      | 12:55   | 91:01    | 1:49        |
| Peak memory      | 182M      | 728M    | 270M     | 336M        |
| Loading (bp)     | 44ms      | 45ms    | 62ms     | 209ms       |
| (ex)             | 2ms       | 14ms    | 18ms     | 4ms         |

Figure 6: Times (min:sec) for index generation and loading

4. THE THREE VIEWS OF A GRAMMAR

An SLT grammar can be seen as a factorization of a tree into its (repeated) tree patterns. Each tree pattern is a connected subgraph of the original tree and is represented by a nonterminal. In our algorithms we found a hierarchy of three different views of the grammar:

1. node-wise (the slowest),
2. rule-wise (the fastest), and
3. chunk-wise.

The node-wise view is a proxy to the original tree and allows to execute arbitrary algorithms (using the first-child, next-sibling, and parent functions). This is the most “detailed” view, but causes a slow-down (comparable to the space improvement of the grammar, when compared to succinct trees). The rule-wise view is the most abstract and high-level view; it means to move through the grammar in one pass, rule by rule. Specialized algorithms such as executing finite-state automata can operate in this view. For strings this idea is well studied [20]. We show in Section 4.2 that the “selecting tree automata” of [24] can be executed in the rule-wise view in order to count selected nodes. This is applied to XPath in Section 5 by compiling queries into selecting automata. The chunk-wise view is slightly more detailed than the rule-wise view. It means that the grammar is traversed (once) in a strict dflr order. This allows to keep track of pre-order numbers and text numbers, by keeping a global counts of element nodes/texts. Through the prMap and SSkip tables we can apply jumping in this view which allows to build fast XPath evaluators for serialization and materialization. Processing in the chunk-wise view is slightly slower than rule-wise (proportional to the rank of the grammar), because the rule of a nonterminal of rank \( k \) is now processed \( k + 1 \) times (instead of only once in rule-wise).

4.1 Node-Wise View

The node-wise interface allows to execute arbitrary algorithms over the original tree (see, e.g., [6]). Inside the interface, a node is represented by a sequence of pairs which shows the productions that were applied to obtain the node. The length of such sequences is at most the depth of the grammar, which be as large as 10000 (see Figure 3). Thus, even if one pair fits into a single bit (which can be done) then this is large compared to the 32 or 64 bits for a pre-order node ID. We observe a slow-down of the original algorithm of
approximately the same factor as the compression. Recursive tree algorithms need a lot of memory due to the size of these sequences. For iterative algorithms we obtain very good time/space trade-offs, see Figure 9.

The node-wise interface provides the functions find-root, first-child, next-sibling, and parent (plus checking the current label of course). A node of the original tree is represented as a sequence of pairs \( \eta = (\text{Start}, p_0)(A_1, p_1) \cdots (A_j, p_j) \), where \( p_0 \) is a node of the start rhs, \( A_1, \ldots, A_j \) are nonterminals, and \( p_1, \ldots, p_j \) are nodes such that rhs(S) at node \( p_0 \) is labeled \( A_1 \), and for every \( 1 \leq i < j \), the rhs for \( A_i \) at node \( p_i \) is labeled \( A_{i+1} \). Moreover, it must hold that the rhs of \( A_j \) at node \( p_j \) is labeled by a terminal symbol, say \( b \). The node ID \( \eta \) is labeled by \( b \), denoted \( \text{lab}(\eta) = b \). The first child (fc), next sibling, and parent functions are realized as in Section 6.2 of [6]. For instance, \( \text{fc}(\eta) \) is the following node ID: we first move to the first child of \( p_1 \) in the rhs of \( A_j \), if it exists. There are three possibilities: (1) \( \text{fc}(p_j) \) is labeled by a terminal symbol. In this case we are finished and return \( \eta' = (A_j, p_j) \rightarrow (A_j, p_{j+1}) \), i.e., \( \eta \) with the last pair replaced by \((A_j, p_{j+1})\). (2) \( \text{fc}(p_j) \) is labeled by a nonterminal \( A_{j+1} \). Let \( \eta' = \eta((A_j, p_j) \rightarrow (A_{j+1}, \epsilon)) \). If the rhs of \( A_{j+1} \) has a terminal at its root node, then the process is finished and return \( \eta' \). Otherwise, more nonterminals (\( A_{j+2}, \ldots, A_{j+k} \)) are added until \( A_{j+k} \) has a terminal root node (and all \( A_{j+1}, \ldots, A_{j+k-1} \) do not). (3) \( \text{fc}(p_j) \) is labeled by a parameter \( y_i \). We remove the last pair from \( \eta \) and consider the \( i \)-th child of the node \( p_{j-1} \) in the rhs of \( A_j \). If it is a terminal, then we are finished. If it is a nonterminal, then we expand as in Step 2. If it is again a parameter, then the pair is removed again, until a non-parameter last pair is found. This terminates with the desired node ID of the first-child node.

As an example, the node ID \( \eta_0 = (S, \epsilon)(A, \epsilon) \) represents the \( f \)-labeled root-node of the tree represented by our example grammar \( G_1 \). To compute \( \text{fc}(\eta_0) \) we move to the first child of \( f \) in \( A \)'s rhs. This is the parameter \( y_1 \). Thus, we pop \( \eta_0 \) and move to the second \( A \) of the start rhs, \((S, 1)\). We expand the \( A \) in one step and obtain the result \((S, 1)(A, \epsilon)\).

### 4.2 Rule-Wise View

The rule-wise view means that the grammar is traversed only once, rule by rule, and in each step only little computation takes place which is "compatible" with the grammar. A classical example of this kind of "computing over compressed structures" is the execution of a finite-state automaton over a grammar compressed string, i.e., over a straight-line context-free grammar (see, e.g., Theorem 9 of [20]). The idea is to memoize the "state-behaviour" of each nonterminal. For tree automata over SLT grammars, the problem was studied in [20] from a complexity theory point of view. We use selecting tree automata as in [24] and build a "count evaluator" which executes in one pass over the grammar. It counts the number of result nodes of the given XPath query.

The new aspect is to combine this evaluator with the jump table. Intuitively, if in a certain state only a given label \( b \) is relevant (meaning that only for that label the automaton changes state or selects the node), then we can jump over nonterminals that do not produce this label \( b \) (determined by the jump table). For instance, consider the query //\( f // b \) which selects all \( b \)-descendants of \( f \)-nodes. It should be intuitively clear that this query can be answered by considering only the \( f \) and \( b \)-nodes of the document (and their relationship). This means that during top-down evaluation we may jump nonterminals which do not produce \( f \) or \( b \) nodes. We now introduce, by example, selecting tree automata (\( ST \) automata), and discuss how they can be executed for result-counting over a grammar. We then show how jumping can be incorporated into this procedure. Here is an example of an \( ST \) automaton:

\[
\begin{align*}
q_0, f \rightarrow q_1, q_0 \\
q_0, L = \{ f \} \rightarrow q_0, q_0 \\
q_1, b \rightarrow q_1, q_1 \\
q_1, L = \{ b \} \rightarrow q_1, q_1
\end{align*}
\]

The first rule says that if in state \( q_0 \) the automaton encounter an \( f \)-labeled node, then it moves to state \( q_1 \) at the first child, and to state \( q_0 \) at the second child. The second rules says that, in state \( q_0 \) and for all labels (denoted by \( L \)) except \( f \), it stays in state \( q_0 \) at both children nodes. In state \( q_1 \) the current node is selected if it is labeled \( b \) (denoted by the double arrow ‘\( \Rightarrow \)’ in the third rule). The automaton realizes the XPath query //\( f // b \) over our binary tree representation of XML trees. We now want to execute this automaton over the grammar \( G_1 \) of Section 4.4 in “counting mode”, i.e., producing a count of the number of result nodes. It starts in state \( q_0 \) processing the start rhs of the grammar. Its root node is labeled \( A \), so the automaton moves to the \( A \)-production (still in state \( q_0 \)). The first automaton rule applies at the \( f \)-labeled node, meaning to process the first child \( (y_1) \) in state \( q_1 \) and the second child \( B \) in state \( q_0 \). The latter means to process \( C \) in state \( q_0 \) which gives state \( q_0 \) at \( y_1 \). We are now finished with processing the nonterminal \( A \) in state \( q_0 \). In summary: no result node was encountered, and the state has moved from \( q_0 \) to state \( q_1 \) at the first parameter \( y_1 \). This "behaviour" of \( q_0 \) on \( A \) is hashed as \((0, q_1)\). Of course, during this computation, the corresponding behaviors for \( C \) and \( B \) are hashed too, i.e., for \( q_0 \) on \( C \) the value \((0, q_0)\) and for \( q_0 \) on \( B \) the value \((0, q_1)\). The automaton continues in state \( q_1 \) at the second \( A \)-node of the start rule. Unfortunately, no hash for \( q_1 \) on \( A \) exists yet, so the automaton needs to be run. Again no result node is encountered and it stays in state \( q_1 \) at \( y_1 \). Thus, \((0, q_1)\) is hashed for \( q_1 \) on \( A \). Finally, it processes the \( a \)-node of the start production, in state \( q_1 \). It gives \( q_1 \) at the \( b \)-node. This node is selected according to the third rule of the automaton and therefore our global result count is increased, to its final value of one. Observe that if there was a third \( A \)-node in the start rhs, such as for the slightly larger tree \( t' \) mentioned before, then hashing is already useful because there will be a hash-hit for the third \( A \). It should be clear that, in the same way, any \( ST \) automaton can be processed in one pass through the grammar (see also [11][20]). Note that we only evaluate \( ST \) automata that are deterministic; it means that for every state \( q \) and every label \( a \) there is at most one transition with left-hand side \( "q, a" \).

### Adding Jumping

Consider the example automaton of before. It should be clear that in state \( q_0 \) the automaton only cares about \( f \)-labeled nodes, i.e., it can omit all other-labeled nodes and safely proceed to the first \( f \)-labeled descendant node (if such a node exists). In the terminology of [24], the omitable nodes are "not relevant". Here we say that a node is relevant if the automaton either selects the node, or changes state, i.e., applies a transition with rhs \( (q', q'') \), where \( q' \neq q \) or \( q'' \neq q \). Note that in [24] relevance is defined based on minimal automata; we have dropped this restriction and define it for arbitrary (but deterministic) \( ST \) automata. We further say that for state \( q \), \( u \) is a relevant label if the automaton's transition for \( q \) and \( u \) is selecting, or changes state, i.e., has rhs \( (q', q'') \) with \( q' \neq q \) or \( q'' \neq q \). Obviously, during the run of the automaton, the relevant labels allow to determine the next relevant node.

We use the jump table in order to omit ("jump") nonterminals which do not contain relevant nodes for the current state \( q \); if the jump table indicates that a nonterminal does not produce nodes labeled \( U = u_1, \ldots, u_k \), and the relevant labels of the current state are in \( U \), then the nonterminal may be jumped. By our definition of
relevance this implies that all parameters of of jumped nonterminal will all be processed in state $q_1$. Back to the example: Since $f$ is a relevant label for $q_0$, we cannot jump the first $A$-node of the start rhs. Hence, the automaton proceeds as before and eventually the entry $(0, q_1)$ is hashed for $q_0$ and $A$. The automaton proceeds in state $q_1$ at the second $A$-node of the start rhs. The only relevant label for $q_1$ is $b$. The jump table tells us that $A$ does not generate $b$’s. Thus, we jump this $A$ and continue evaluating at its child node. This saves a lot of computation (roughly half of before). But, in which state is the automaton supposed to continue? It must be state $q_1$ because, by definition of relevance, the state never changes on all non-relevant nodes. Thus, parameter $y_1$ must be reached in state $q_1$. We proceed, to the $b$-node of the start rhs and compute the correct final count of 1.

As another example, imagine the start rhs was $A(A(b))$ and we execute a query that selects all $b$-children of the root node. In XPath $/b$ (let us ignore that in XML the root node has only one child). An automaton for this query is:

$$q_0, b \Rightarrow q_1, q_0$$
$$q_0, L - b \Rightarrow q_1, q_0$$
$$q_1, L \Rightarrow q_1, q_1.$$ 

In state $q_0$, all labels are relevant, because there is a state change in all transitions for $q_0$. We therefore process as before, eventually hash the entry $(0, q_1)$ for $q_0$ and $A$, and determine that $y_1$ of $A$ need to be processed in state $q_1$. For this state, no label is relevant. Thus, the second $A$ may be jumped. We arrive at the $b$-node of the (new) start production of above, and terminate (with count zero).

### Adding Skipping

When an automaton is in its universal state $q_U$, we may skip the entire subtree because it contains no relevant nodes. For the count evaluator this is done by omitting all recursive calls to state $q_U$. This holds for terminal nodes, as well as for the hashed behavior of nonterminal nodes. For the materialize and serialize evaluators, it is necessary to know the number of element nodes/text nodes of the skipped subtree to correctly continue evaluating. During recursion these numbers are determined by the prMap/textMap tables. If we are in the start rhs, then we use the SSkip/textSSkip tables.

### 4.3 Chunk-Wise View

We now wish to serialize XML result subtrees of the nodes selected by an automaton. Additional to the grammar, we need access to the text values of the XML document. We assume a function getText(i) which returns the $i$-th text or attribute value of the document (starting from zero). For instance, getText(6) returns the 7-th text value, i.e., the string serialization for this example document:

```xml
<q><f><f><a><a><b></b></a></a><c>is</c><c>/b</c></a><c><a>test</a></c><a></a></f></f><a><c>document</c><c>for the purpose</c></a></f></f><a><c>of explaining</c><c>serializatio</c></a></f></f><a><c>/a</c></a></g>
```

A faithful grammar representation of the XML structure tree of this document is:

\[
\begin{align*}
S & \rightarrow \text{g(}_T, A(A(a(b(}_T, c(}_T)))) \\
A(y_1) & \rightarrow f(y_1, B) \\
B & \rightarrow C(}_T) \\
C(y_1) & \rightarrow a(y_1, c(}_T))
\end{align*}
\]

For simplicity we do not transform this grammar into bCNF. We would like to serialize (using the jump table) the nodes selected by this automaton:

\[
q_0, c \Rightarrow q_0, q_0 \\
q_0, L - c \Rightarrow q_0, q_0
\]

The automaton begins in state $q_0$ at the root of the start rhs. The recursive algorithm over grammar rules is shown in Figure 7 exactly the same algorithm is used over the start rhs (but iteratively, using stacks). During a dfir traversal the global counter num_T stores the number of _T nodes seen so far. Thus, at $q_1$’s first child num_T is set to 1. We now process, still in state $q_0$. $A$’s first chunk (that is: the sequence of tags from $q_0$’s root node to its first parameter node). This is done by first calling the rule-wise evaluator of Section 4.2 in order to compute and hash the parameter states for $A$ and the information whether a parameter is inside a result subtree (see the $A$’s in the algorithm of Figure 7). This will add the hashes ($q, q_0$) for $q_0$ on $C$, and (2) for $B$, and (2,$q_0$) for $A$. The first chunk only contains $<c>$ and therefore the empty tag sequence (0, 0) is hashed for $A$’s first chunk in $q_0$, i.e., for the triple ($q_0$, $A$, 1). Further, ($q_0$, $A$, 2) is pushed onto our “pending computation” (PC) stack. The next step in dfir is the first chunk of the second $A$-node. Both rule-wise and chunk-wise behaviors are hashed already, so nothing needs to be computed and again ($q_0$, $A$, 2) is pushed onto the PC stack. The dfir traversal continues at the subtree $a(b(}_T, c(}_T))$. The $a$ and $b$ nodes do not cause state changes or node selection. At the first $T$-node, num_T is set to 2. At the $c$-node a selecting transition fires. Thus, we now start appending tags to the “intermediate result tag” (IRT) sequence, first the tag $<c>$. We also append the pair of start position and current num_T value to the “final result list”. Moreover, $</c>$ is pushed onto the PC stack. Evaluation
continues at the _T-child (thus num_T is increased to 3). We append <T/> to the IRT sequence. Since _T is a leaf, the PC stack is popped and therefore add </A> to the IRT sequence. This finishes the result subtree. At the next step we return to the a-node of the start rhs. We return to the second A-node and pop the PC stack. This gives (q₀, A, 2). No (begin, end)-pair is hashed for this triple, so A’s second chunk is processed in state q₀. Recursion continues to B and C and finally move to the parameter tree c(T) of C in B’s rhs. This causes to append </A><T/> to the IRT sequence, to append (4, 2) to the final result list, and to increase num_T (to 4). We proceed at the second chunk of C, ignore </A>, and append </A><T/> to </A> and (7, 3) to the IRT sequence and final result list, respectively, and increment num_T (to 4). The pair (7, 9) is now hashed for the triple (q₀, C, 2). The grammar recursion continues at B and A, so (4, 9) is hashed for (q₀, B, 1) and (4, 9) for (q₀, A, 2). The dfr run continues at the first A of the start rhs and pop the PC stack. This gives (q₀, A, 2). We now have our first hash-hit and happily retrieve the (begin,end)-pair (4, 9). This is interpreted as a “copy instruction”: append to the IRT sequence (currently at position 10) its own content from position 4 to 9. During this copying we observe that 4 and 7 are final result begin-positions, and that their corresponding num_T-values are 5 and 6, respectively (by incrementing num_T during copying). The content of the final result list is (1, 2)(4, 3)(7, 4)(10, 5)(13, 6). The IRT sequence contains five copies of </A><T/> to </A>. In a final step we print correct XML document fragments for each result. This is done by copying from the IRT sequence while inserting for each <T/> the correct text value.

To see how jumping works, consider the query /b over this grammar. Now A’s first chunk can be jumped. During the rule-wise traversal, jumping takes place as discussed in Section 4.2. Next, the first chunk of the second A-nodes in the start rhs is jumped. The first hit for /b is obtained at the b-node of the start rhs. The dfr traversal jumps the second chunks of both A-nodes of the start rhs, and is finished. The final result list is (1, 1) and the IRT sequence is <boomla> to </boomla>. Thus, we print <boomla> to </boomla>.

Comments to Figure 7: In Line 2 we calculate rule-wise the parameter states s₁, ..., sₙ, and the Booleans u₁, ..., uₙ, which determine if a parameter is inside of a result subtree. Line 3: if p = 0 then yₙ refers to the root node and if p = rank(N) then yₙ₊₁ also refers to the root node. The Xᵢ and pᵢ are determined by the shape of rhs(N).

5. XPATH EVALUATION

We built rudimentary XPath evaluators that compile a given XPath query into an ST automaton. Our current evaluator only works for the //, and following-sibling axes and does not support filters. The count evaluator is based on the rule-wise view of Section 4.2 while the materialize and serialize evaluators are based on the chunk-wise view of Section 4.3. For the small XPath fragment considered here, the translation into ST automata is straightforward and similar to the one of [24] (essentially, the automaton is isomorphic to the query). First, an automaton is built which uses nondeterminism for the //axis. For instance, we first obtain an automaton similar to the one shown in the beginning of Section 4.2 but with L = { } replaced by L and L = { } replaced by L. Different from [24] which work on-the-fly, we fully determinize the automaton before evaluation. For the example, this gives precisely the automaton as shown. We can prove that determinization of our ST automata does not cause an exponential blow up. This is due to the simple form of our queries. Moreover, the determinization procedure always produces minimal automata. In fact, it can be shown that for a given XPath query with m-axes (over //, and following-sibling) the resulting deterministic ST automaton has at most 2m states. Note that in terms of the transitions along a first-child path, our deterministic ST automata behave exactly in the same way as “KMP-automata” (see, e.g., Chapter 32 of [3]), i.e., matching along a path works very much in the same way as the well-known KMP-algorithm. What is the time complexity for counting, i.e., of executing a deterministic ST automaton over an SLT grammar? As mentioned already in [20], even for general context-free tree grammars, a deterministic top-down tree automaton can be executed in polynomial time. We make this more precise: for each nonterminal (of rank k) of the grammar and state of the automaton, we need to compute only at most one k-tuple of parameter states. Hence, the following holds.

**Lemma 5.1.** Let G be an SLT grammar in which every production is in bCNF and let M be an ST automaton. Let n be the number of nonterminals of G, k the rank of G, and m the number of states of M. The automaton M can be executed rule-wise (e.g., for counting) over the grammar G in time O(mnk).

Note that an alternative way is to first reduce the number of parameters of the grammar to one, using the result of [22]. For a binary ranked alphabet (as we are using here for XML), the size of the resulting grammar is O(2(G)), where |G| denotes the size of G. We then apply the above theorem in time O(mn′k), where n′ is the number of nonterminals of the new grammar. It remains to be seen in practice which of the two approaches give better running times.

6. EXPERIMENTS

All experiments are done on a machine featuring an Intel Core2 Xeon processor at 3.6GHz, 3.8GB of RAM, and an S-ATA hard drive. The OS is a 64-bit version of Ubuntu Linux. The kernel version is 2.6.32 and the file system is ext3 with default settings. All tests are run with only the essential services of the OS running. The standard compiler and libraries available on this distribution are used (namely g++ 4.4.1 and libxml2 2.7.5 for document processing). Each query is run three times and of the running times select the fastest one. We only count query execution time, i.e.,

```c
function recPrint(int N, state s, chunkNo p, bool u) {
    let S = (X₁, s₁, p₁, u₁), (X₂, s₂, p₂, u₂) . . . (Xₙ, sₙ, pₙ, u₀)
    be the of TNT-chunks in rhs(N) between “yᵢ” and “yᵢ₊₁”;
    int curLength = IRT_length;
    for (i = 1 to n do
        if (Xᵢ is nonterminal) then
            if (hash(Xᵢ, sᵢ, pᵢ, uᵢ) = (z₁, z₂)) then
                for j = 1 to z₂ do
                    append(IRT, IRT(z₁ + j));
                    if (IRT(z₁ + j) is a result) then
                        append(FRL, (IRT_length, num_T));
                else recPrint(Xᵢ, sᵢ, pᵢ, uᵢ);,
            else if (pᵢ = 0) then
                if (sᵢ, tag(Xᵢ)) is selecting or uᵢ = 1) then
                    append(IRT, “<tag(Xᵢ)>”);
                if (sᵢ, tag(Xᵢ)) is selecting ) then
                    append(IRT, (IRT_length, num_T),
                if (tag(Xᵢ) = T) then num_T++;
                if (pᵢ = 1 and (sᵢ, tag(Xᵢ)) is selecting or uᵢ = 1) then
                    append(IRT, “</tag(Xᵢ)>”);
            hash(N, s, p, u) = (curLength, IRT_length - curLength); } }

Figure 7: Grammar-recursive case of the print function
```
do not take into account query translation times etc. For experiments that involve serialization the programs are directed to write to /dev/null.

**MonetDB.** We use Server version 4.38.5, release Jun2010-SP2. This contains the MonetDB/XQuery module v0.38.5. We compare pure query execution time, so report the “Query” time reported.

**Qizx.** Version 4.0 (June 10th, 2010) of the free engine is used. The “-v + 2” switches are used. For count queries we use the “evaluation time:”-number reported by Qizx. For serialization the sum of the “evaluation time:” and “display time:”-numbers are used. For a few count queries Qizx executed faster over the XML structure tree than over the original XML document. This is indicated by a footnote in Figure ref:fig:xmarkrun.

**SXSI.** The version used for [11] was supplied to us by the authors.

### 6.1 Traversal Access

To investigate the speed of our node-wise view, we consider fixed traversals: depth-first left-to-right (dflr) and dfrl traversals, both recursively and iteratively. Dflr traversals are common access pattern for XPath evaluation.

The speed of our interface is lower-bounded by the speed of the start rhs representation. Since it takes time to get a single pair out of our node ID sequence data structure, a plain traversal through the (bp)-start rhs is slower than a traversal through “Succinct” (=the whole XML structure tree represented in (bp)). To see this, we built grammars that have no nonterminals (except Start) but store the complete tree in their start rhs. The full traversal speed of these grammars is shown as OneRule in Figures 8 and 9. It is also possible to transform the Start rhs into bCNF. Intuitively, this will introduce as many new nonterminals as there are nodes in the Start rhs. If we apply this to the OneRule grammars of before, then we obtain NoStartRule grammars in which each node is explicitly represented by a nonterminal. The traversal speed over such grammars should be comparable to that of pointers, because this is similar to a pointer-based representation. Again, this is not exactly the case, because of the additional overhead implied by our node ID data structure. Finally, compressed grammars: we test (binary tree) same. The only big difference is that for recursive, the DAG line is in between the TreeRePair and the NoStartRule lines. Note that for recursive traversals we added a data structure called “node pool” which realizes prefix-sharing of node IDs. Without such a data structure, recursive traversals are roughly ten times slower (due to dynamic allocation of node IDs). Through profiling we found the

![Figure 8: Recursive tree traversals over XMark](image)

DAGs, BPLEX, and TreeRePair grammars. The resulting traversal speeds for iterative full traversals are shown in Figure 9. For recursive traversals the graph looks similar: all run times are about twice as fast as in the iterative graph, except for “Pointer” which stays the

![Figure 9: Iterative tree traversals over XMark](image)

![Figure 10: Space requirement for iterative traversals](image)

reason why DAG traversals are much slower in the iterative case: DAG grammars have about eight times more calls to the parent function (in the start rhs). The number of these calls is approximately equal to the number of nodes in the start rhs. As shown in Figure 11 the size of the start rhs is about eight times more than those of BPLEX and RePair. Note that for NoStartRule grammars the number of nonterminals equals two times the number of non-_T-nodes of the document, plus the number of _T-nodes of the document. This is because we use one fixed nonterminal to represent _T-nodes, i.e., we hash-cons all _T-subtrees. The OneRule grammars have (2n − 1)-many nodes in the start rhs, because for every binary node there is an additional null-tree.

To summarize the time/space trade-off: for recursive traversals, compared to succinct trees our interface (using TreeRePair grammars) is 5–6 times slower and uses 3 times less space, while it is 12 times slower and uses 18 times less space when compared to pointers. For iterative traversals we are 7 times and 15–16 times slower compared to succinct and pointers, respectively, and use 9–15 and 167–309 times less space, respectively.

### 6.2 Counting

Figure 12 shows timings for XPath counting over 116MB, 1GB, and 11GB XMark files. The queries Q01–Q08 and Q13–Q16 are shown in Figure 11 while queries X1–X3 (taken from [14]) are shown in Figure 14. For TinyT we load our base index plus the jump table. For SXSI and MonetDB it was beneficial to load the entire original document: this gave faster counting times than pro-
Label Queries

A label query is of the form //a1/a2/···/an, where a1, ..., an are label names, is called simple path. Note that each ai must be an element name, i.e., the wildcard-star (*) is not allowed. Such queries can be handled by the specialized index of Ferragina et al. [9, 10]. In fact, that index can even materialize result nodes, but not by pre-order numbers. We therefore did not include it in Section 4.4. We use our own implementation of [9, 10], called “Fer+”. It is optimized for speed, not for size. Their own java implementation [http://www.d1.unipi.it/~ferragin/Libraries/xbwt-demo.zip] produces much smaller indexes, see column “Fer-j” in Figure [14] but also performs much slower. For instance, it uses 476ms for query q1 and takes 106ms for the query q2.

Simple paths

An XPath query of the form //a1/a2/···/an, where a1, ..., an are label names, is called simple path. Note that each ai must be an element name, i.e., the wildcard-star (*) is not allowed. Such queries can be handled by the specialized index of Ferragina et al. [9, 10]. In fact, that index can even materialize result nodes, but not by pre-order numbers. We therefore did not include it in Section 4.4. We use our own implementation of [9, 10], called “Fer+”. It is optimized for speed, not for size. Their own java implementation [http://www.d1.unipi.it/~ferragin/Libraries/xbwt-demo.zip] produces much smaller indexes, see column “Fer-j” in Figure [14] but also performs much slower. For instance, it uses 476ms for query q1 and takes 106ms for the query q2.

```
q1 = //site/categorie/category
q2 = //open_auctions/open_auction/annotation
q3 = //description/text/text()
q4 = //text/text()
```

Figure 15: Simple path queries, counting (in ms)

Our “Fer+” implementation is fast for queries with low selectivity and slow for those with large selectivity. This can be seen in Figure 15 for q1 which has the lowest selectivity, Fer+ is 15-times faster than TinyT, while for q4 TinyT is slightly faster than Fer+. For larger XMark sizes the relative performance of TinyT is better, due to compression: for XMark1G, TinyT is already faster for q3, and is faster by a factor of > 4.5 for query q4.

6.3 Serialization

Figure 12 shows timings for serialization over 116MB and 1GB XMark files. TinyT gave the fastest times for all our serialization experiments. For printing a single subtree (e.g., Q01 and Q02)
|   | Q01 | Q02 | Q03 | Q04 | Q05 | Q06 | Q07 | Q08 | X1  | X2  | X3  | Q13 | Q14 | Q15 | Q16 |
|---|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
|XM|   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Mark 16M, counting | TinyT | **0.002** | SXSI | 1 | 14 | 16 | 26 | 12 | 36 | 32 | 22 | 18 | 18 | 309 | 313 | 330 |
|       | Monet | 8 | 24 | 24 | 33 | 16 | 18 | 22 | 23 | 12 | 15 | 56 | 196 | 476 | 760 |
|       | Qizx | 1 | 17 | 26 | 33 | 6 | 137 | 53 | 1 | 130 | 28 | 112 | 3954 | 20457 | 21014 |
|XM|   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Mark 16M, serialization | TinyT | 141 | 46 | 19 | 26 | 9 | 168 | 66 | 66 | 9 | 22 | 18 | 2507 | 2247 | 1773 | 312 |
|       | SXSI | 199 | 67 | 22 | 56 | 3 | 138 | 85 | 15 | 44 | 44 | 6821 | 6267 | 5063 | 1038 |
|       | Monet | 750 | 238 | 30 | 110 | 38 | 757 | 85 | 49 | 48 | 53 | 10977 | 9406 | 6095 | 1636 |
|       | Qizx | 2550 | 839 | 58 | 456 | 74 | 2721 | 257 | 18 | 56 | 179 | 84 | 45157 | 44264 | 8181 | 21680 |
|XM|   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Mark 1G, counting | TinyT | **0.004** | SXSI | 2 | 107 | 207 | 79 | 665 | 342 | 156 | 146 | 174 | 4376 | 4371 | 4382 | 4500 |
|       | Monet | 11 | 311 | 353 | 399 | 1191 | 2480 | 1238 | 365 | 295 | 332 | 3026 | 4370 | 6973 | 9673 |
|       | Qizx | 1 | 89 | 98 | 115 | 32 | 1266 | 412 | 1107 | 195 | 1015 | ++ | ++ | ++ | ++ |
|XM|   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| Mark 1G, serialization | TinyT | 1225 | 408 | 91 | 201 | 67 | 1454 | 412 | 398 | 68 | 152 | 139 | 23987 | 21213 | 18109 | 2339 |
|       | SXSI | 1922 | 639 | 214 | 597 | 325 | 3598 | 2381 | 1606 | 304 | 393 | 258 | 73126 | 67169 | 55564 | 112229 |
|       | Monet | 13903 | 4287 | 3484 | 4099 | 4014 | 12429 | 25557 | 12618 | 4178 | 4043 | 4130 | 115327 | 98622 | * | 112212 |
|       | Qizx | 24478 | 7767 | 189 | 203 | 4015 | 24080 | 2432 | 1632 | 520 | 375 | 562 | 325794 | ** | ** | ** |

|   | Monet | 11 | 11 | 3386 | 4017 | 5214 | 10570 | 24815 | 11079 | 4111 | 3576 | 3802 | 103401 | 397319 | 879057 | ++ |

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

|   | TinyT<br> serialization is 5-8 times faster than SXSI. | For larger result sets the time difference is bigger. The largest time difference is for Q07 (over XMark1G): TinyT serializes 1.5-times faster than the next fastest system (SXSI). For larger result sets the time difference is bigger. The largest time difference is only 2.1. This suggests that the speed-up is related to the compression in our index. This is interesting, because one would expect that the pure XML serialization time will dominate query evaluation and book keeping times. For TinyT, we load all indexes shown in Figure 5, together with a “text collection”. The latter gives access to getText(i), the i-th value (text or attribute) of the document. Our text collection stores all text (consecutively) in a huge memory buffer. This takes space (size of the file minus “Non-Text” value in Figure 2, e.g., 82MB for XMark116M). We use a simple data structure to map from text numbers to begin positions in the buffer. During serialization we opted for speed, not space. Recall from Section 4.3 our serialization process: we first build tag sequences of all document subtrees to be output, together with copy instructions that point into those sequences. These tag sequences still contain _T tags. After evaluation, XML serialization starts by (1) writing full XML subtrees by correctly replacing _T nodes by their text values and also (2) replacing copy instructions by their correct serialization. |

6.4 Materialization

Initially TinyT was built for fast evaluation of XPath count queries. Later we realized its usefulness for fast serialization; the key idea was to avoid materialization of result nodes and to print directly in parallel with query evaluation. The running times for both counting and printing are highly competitive, as can be seen in Figure 12. We also wanted to compare to specialized systems such as implementations of twig queries.Twig queries have been studied extensively both from a theoretical and an implementation view point. They belong to the most highly optimized XPath queries (see, e.g., [14] and the references those articles) Twig query implementations materialize several context nodes per query result. This is different from XPath semantics in which one node only is selected at a time. Clearly it would not be fair to compare our count evaluator with a twig implementation that materializes (even multiple nodes per result). We decided to build a materializer for TinyT which produces pre-order numbers of the result nodes. This was done in short time, by essentially reusing the code of the serializer, and indeed, doing a fair amount of serialization in memory during materialization. Certainly, this implementation is far from optimal; it would be much more efficient to work over node offset numbers, rather than serialized XML tag sequences. As the experiments in Figure 16 show, TinyT is the fastest only for query X3, while for X1 and X2 XLeaf and TJStrictPre are the fastest, respectively. We believe that a more efficient implementation of materializing over TinyT can be considerable faster, ca. 2–3 times slower than counting. |

6.5 Compression Behavior

Our algorithms that execute without decompression directly on the grammar such as rule-wise XPath counting or chunk-wise XPath serialization, both do one pass through the grammar. Thus, the running time of these algorithms is strongly influenced by the size of the grammar. TreeRePair generates smaller grammars than BPLEX (about half the size, in terms of numbers of edges) [21], which itself makes smaller grammars than DAGs [6]. Therefore, our count and serialize XPath evaluators run fastest over grammars produced by TreeRePair. The size of the start rhs is important too, because access is slower and more complicated than over the recursive rules (compare run times of OneRule with NoStartRule in Figures 8 [15]).
We presented a new structural index for XML and evaluated its performance for XPath evaluation. The index is based on a grammar compressed representation of the XML structure tree. For common XML documents the corresponding indexes are minuscule. When executing arbitrary tree algorithms over the index, a good time-space trade-off is obtained. For certain simple XPath tasks such as result node counting, impressive speed-ups can be achieved. Our rudimentary XPath implementation over this index outperforms the fastest known systems (MonetDB and Qizx), both for counting and for serialization. We built and experimental materializer which is competitive with the state-of-the-art twig query implementations. We believe that our system is useful for other XPath evaluators and XML databases. It can be used for selectivity computation of structural queries, and for fast serialization. It will be interesting to extend our current XPath evaluators to handle filters, and also to handle data value comparisons. For the latter bottom-up evaluator as in [24] could be built, which first searches over the text value store, and then verifies paths in the tree, in a bottom-up way. For such queries the SXSI system [1] is highly efficient. We do not expect to achieve faster run times with our index, but think that run times similar to those of SXSI can be achieved. This is a large improvement, because the space requirement of our index is much smaller than that of SXSI.

It would be interesting to add specialized indexes which allow more efficient running times for simple queries, such as simple path queries of the form \(\text{//a}_1/\text{//a}_2/\ldots/\text{//a}_m\). Over strings, the self-index of [7] allows to find occurrences of such queries in time logarithmic in the number of rules of the grammar. Can their result be generalized to the tree case? In terms of extraction (decompression) there are new results for DAGs [6] that run in time logarithmic in the number of edges of the DAG. Can this result be generalized from DAGs to our SLT grammar?

### Acknowledgments

We are thankful to Francisco Claude for providing to us an alternative implementation of [8][10]. Many thanks to the authors of [14][15] and to Nils Grimmelmo in particular, for providing to us their implementations used for those articles. Finally, we thank Xavier Franc from XMLmind for providing to us their implementation of Qizx, and for assisting us in testing against their system.

### References

[1] 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.

[2] R. A. Baeza-Yates and G. H. Gonnet. Fast text searching for regular expressions or automaton searching on tries. J. ACM, 43(6):915–936, 1996.

[3] P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings. In SODA, 2011. To appear.

[4] P. A. Bonec, 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 Conference, pages 479–490, 2006.

[5] P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In VLDB, pages 141–152, 2003.

[6] G. Busatto, M. Lohrey, and S. Maneth. Efficient memory representation of XML document trees. Inf. Syst., 33:456–474, 2008.

[7] F. Claude and G. Navarro. Self-indexed grammar-based compression. To appear in Fundamenta Informaticae, 2010.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Introduction to Algorithms, Second Edition. The MIT Press

### XMark1G (millisecond)

| RePair | 12MB | 116MB | 224MB |
|--------|------|-------|-------|
| TinyT  | 25   | 52    | 34    |
| TinyT+jump | 36  | 38    | 33 |
| SXSI   | 183  | 174   | 164   |
| MonetDB| 235  | 180   | 199   |
| TJS respectively | 145 | 21 | 40 |
| XLeaf  | 3.6  | 47    | 47    |

**Figure 16**: Twig queries, materialization (in ms)

| XMark     | 12MB | 116MB | 224MB |
|-----------|------|-------|-------|
| BPLEX     | 1724 |       |       |
| DAG       | 194032 | 192029 | 628 |
| PRJ       | 189683 | 894695 | 116801 |

**Figure 17**: Size start rhs and number rank-k nonterminals
[9] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In FOCS, pages 184–196, 2005.

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

[11] D. K. Fisher and S. Maneth. Structural selectivity estimation for XML documents. In ICDE, pages 626–635, 2007.

[12] G. H. L. Fletcher, D. V. Gucht, Y. Wu, M. Gyssens, S. Brenes, and J. Paredaens. A methodology for coupling fragments of XPath with structural indexes for XML documents. Inf. Syst., 34:657–670, 2009.

[13] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In VLDB, pages 436–445, 1997.

[14] N. Grimsmo, T. A. Bjørklund, and M. L. Hetland. Fast optimal twig joins. PVLDB, 3(1):894–905, 2010.

[15] N. Grimsmo, T. A. Bjørklund, and Ø. Torbjørnsen. Xleaf: Twig evaluation with skipping loop joins and virtual nodes. In DBKDA, pages 204–213, 2010.

[16] H. He and J. Yang. Multiresolution indexing of XML for frequent queries. In ICDE, pages 683–694, 2004.

[17] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth. Covering indexes for branching path queries. In SIGMOD Conference, pages 133–144, 2002.

[18] R. Kaushik, P. Shenoy, P. Bohannon, and E. Gudes. Exploiting local similarity for indexing paths in graph-structured data. In ICDE, pages 129–140, 2002.

[19] J. Lamping. An algorithm for optimal lambda calculus reductions. In POPL, pages 16–30, 1990.

[20] M. Lohrey and S. Maneth. The complexity of tree automata and XPath on grammar-compressed trees. Theor. Comput. Sci., 363(2):196–210, 2006.

[21] M. Lohrey, S. Maneth, and R. Mennicke. Tree structure compression with RePair. CoRR, abs/1007.5406, 2010. Short version to appaer as paper in DCC 2011.

[22] M. Lohrey, S. Maneth, and M. Schmidt-Schauß. Parameter reduction in grammar-compressed trees. In FOSSACS, pages 212–226, 2009.

[23] S. Maneth, N. Mihaylov, and S. Sakr. XML tree structure compression. In DEXA Workshops, pages 243–247, 2008.

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

[25] T. Milo and D. Suciu. Index structures for path expressions. In ICDT, pages 277–295, 1999.

[26] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In ALENEX, 2007.

[27] P. M. Pettovello and F. Fotouhi. MTree: an XML XPath graph index. In SAC, pages 474–481, 2006.

[28] C. Qun, A. Lim, and K. W. Ong. D(k)-index: An adaptive structural summary for graph-structured data. In SIGMOD Conference, pages 134–144, 2003.

[29] K. Runapongs, J. M. Patel, B. Bordawekar, and S. Padmanabhan. XIST: An XML index selection tool. In XSym, pages 219–234, 2004.

[30] W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In ICALP, pages 15–27, 2004.

[31] K. Sadakane and G. Navarro. Fully-functional succinct trees. In SODA, pages 134–149, 2010.

[32] XMLmind Products. Qizx, a fast XML database engine fully supporting XQuery. http://www.xmlmind.com/qizx/qizx.html.

[33] K. Yi, H. He, I. Stanoi, and J. Yang. Incremental maintenance of XML structural indexes. In SIGMOD Conference, pages 491–502, 2004.