An In-Memory XQuery/XPath Engine over a Compressed Structured Text Representation

Angela Bonifati Gregory Leighton Veli Mäkinen Sebastian Maneth
Gonzalo Navarro[*] Andrea Pugliese

Abstract

We describe the architecture and main algorithmic design decisions for an XQuery/XPath processing engine over XML collections which will be represented using a self-indexing approach, that is, a compressed representation that will allow for basic searching and navigational operations in compressed form. The goal is a structure that occupies little space and thus permits manipulating large collections in main memory.

1 Generalities

In principle we will aim at a static representation, because it will be significantly faster and easier to program (a good part already exists). Only for the text we will use a dynamic representation at construction time, so as to permit building the index in compressed form.

Let u be the total length of the collection (measured in symbols), n be the total number of nodes, Σ the collection alphabet and σ its size, t the total number of different tag and attribute names. In the mixed text model there are $2n-1$ texts, but say there are $d < 2n$ nonempty ones. Logarithms will be in base 2 and $H_k(S)$ will denote the empirical k-th order entropy of sequence S [Man01].

Σ will be regarded in this section as the set of byte values $\{1 \ldots 255\}$ and a special terminator will be called $\$ = 0$. Support for multibyte sequences will not be provided at this level, but those encodings guarantee that we can ignore them, and still a substring search for a valid multibyte string will not yield misaligned occurrences.

2 Basic Representation and Data Structures

For a survey of many compact data structures and compressed indexes see [NM07]. In what follows we cite the original papers only.

2.1 Data Representation

The representation of a structured text collection (XML) will be composed of the following compressed/compact data structures. These at the same time represent the collection and provide navigation and indexed access to it.

[*]Partially funded by Yahoo! Research project ".Compact Data Structures".

Par: A parentheses representation of the tree structure. In particular we will use the balanced parentheses representation [MR97] (which now has good implementations supporting i-th child query [Sad08]), obtained by traversing the tree in DFS order and writing a `"("` whenever we arrive at a node and a `")"` when we leave it. A tree node will be identified by its corresponding opening parenthesis in *Par*. There will be no support for attribute nodes at this level, so the upper level interface must convert them to normal tree nodes. Overall *Par* will need $2n + o(n)$ bits.

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 the range $[1, 2t]$. The sequence is aligned with *Par* so that the tag of node i is $Tag[i]$. *Tag* can be stored using a wavelet tree [GGV03], taking at most $2n \log(2t) + o(n) \log t$ bits, and even $2nH_0(Tag) + o(n) \log t$ if we use a compressed wavelet tree (e.g. giving it Huffman shape or representing the bitmaps using RRR structure [RRR02]). This would provide *rank*, *select* and *access* operations in $O(\log t)$ time. We can also use Golynski et al.'s representation [GMR06] to achieve $2n \log(2t) + n\, o(\log t)$ bits of space, *select* in constant time and *rank* and *access* in $O(\log \log t)$ time (which should be closer to constant in practice). We could replace this by two sequences over t symbols, sorted in preorder and in postorder, to convert $2n \log(2t)$ into $2n \log t$, but the extra $2n$ bits save us from *rank*/*select* to map from parentheses to tag sequence positions.

TagName: A simple table mapping tag names (strings) to tag identifiers, for displaying purposes. Space is proportional to t.

Text: A collection of at most $2n - 1$ texts (or "documents") of total length at most u. Those correspond to the textual contents at the tree leaves and also between consecutive children of each node (and before/after the first/last child). Their position in the collection (in a left to right read of the XML) will be their document identifier (a number in $[1, 2n - 1]$). The collection will be represented as the wavelet tree of the Burrows-Wheeler Transform (BWT) [BW94] of the set of texts. The texts will be inserted into the collection as independent strings, ending with a terminator \$ (if one tries to move backward from the first symbol, one ends up in the \$ terminator of some other text). The insertion will be so that the terminator of the i-th text will appear at $F[i]$ in the BWT, and thus it would be easy to extract it given a document identifier i. We note that the mapping of tree node i to the range of document identifiers below it is simply $[i, i+2 \cdot SubtreeSize(i)-2]$. The space will be $uH_k(T)+o(n \log \sigma)$ bits, being T the collection of all the texts.

Doc: The document number corresponding to the ranks of the \$'s of the BWT. Given a \$ at $L[i]$, $rank_\$(L, i)$ gives the rank, and the mapping gives the document number (note the terminators \$ are not ordered by document number). Needs $2n \log(2n)$ bits, or $2n + d \log(2n)$ if we do not index empty texts (in which case a bitmap of length $2n - 1$ indicating the valid documents is added).

2.2 Construction

Construction will operate so that the XML collection can be read in one pass from disk (in DFS order), yet the data structures must fit in memory in compressed form.

The parentheses *Par* are built by appending bits as we read tags, in left-to-right, order. Extra parentheses structures can be built on them at the end, or even at loading time if this does not need much extra space.

For *Tags*, we could first store a sequence of integers (maybe with byte codes as we do not know in the beginning how many different tags are there), and at the end we can build the wavelet tree or Golynski et al.'s structure on it.

Thus as the parser sends events of starting/ending of nodes, one just appends to those sequences. Instead, we should maintain a dynamic (and hopefully compressed) wavelet tree for inserting the consecutive texts that the parser sends. The parser should notify even about the empty texts between consecutive tags. We insert the new text into the collection *Text*, right to left, starting by inserting its last symbol at $bwt(i)$, where i is the new document number. Should we decide that there are too many empty texts, we add 0/1 to a bitmap marking the nonempty texts (to be later provided with *rank* operation) and insert only nonempty texts to the collection (and increasing text identifiers only when inserting them).

At the end, the interface sends the tag name table, the structures are made static, and saved to disk (together with the name tables). Sublinear extra structures that can be easily built on the fly (*rank*, *findclose*, etc.) are not saved. The text collection can be discarded at this point, as it is represented by the structures just built.

The *Doc* mapping can be stored initially as a dynamic sequence supporting insertions, and storing the document identifiers ordered by their positions in L. If we insert document i and its terminator \$ falls at $L[p]$, then we insert i at position $rank_{\$}(L, p)$ in this sequence.

2.3 Loading

The index is saved under a given file name, using different extensions, and is loaded with the same file name. Some sublinear structures are built on the fly upon loading, transparently.

Once loaded in static form, the data structures will be able of supporting various tree navigation and text searching operations.

2.4 Tree Navigation

These are operations to move around the tree and make some queries on it. Recall that the node identifier is the same as a parenthesis position in *Par*, although the interface should not need to know that in principle. All these operations are constant time unless otherwise noted (but we mark which are particularly fast). Note that *tag* is a tag identifier in the following.

First the navigation queries:

Root: Returns the tree root node (in *Par* this is simply 1).

SubtreeSize (x)**:** Returns the number of nodes (and attributes!) below node x.

SubtreeTags (x, tag)**:** Returns the number of occurrences of tag within the subtree of x. This involves a symbol rank in Tag, time $O(\log\log t)$ and usually $O(1)$.

IsLeaf (x)**:** Returns whether x is a leaf. Very fast.

IsAncestor (x, y)**:** Returns whether x is ancestor of y. Very fast.

IsChild (x, y)**:** Returns whether x is parent of y. $IsAncestor(x, y)$ and $Depth(x) = Depth(y) - 1$.

NumChildren (x)**:** Number of children of x.

ChildNumber (x)**:** Returns i if x is the i-th of its parent.

Depth (x)**:** Depth of node x, $excess(Par, x)$. A simple binary $rank$.

Preorder (x)**:** Preorder numbering of node x. This is just $rank_((Par, x)$.

Postorder (x)**:** Postorder numbering of node x. This is $rank_)(Par, findclose(Par, x))$.

Tag (x)**:** Gives the tag identifier of node x, just $Tag[x]$. Takes $O(\log\log t)$ time on Golynski et al.'s representation, usually $O(1)$.

DocIds (x)**:** Gives the range (i.e., a pair of natural numbers) of document identifiers that descend from x.

Now moving operations:

Parent (x)**:** Gives the parent of node x. Assumes x has a parent.

Child (x, i)**:** Gives the i-th child of x, assuming it exists.

FirstChild (x)**:** Gives the first child of x, assuming it exists. Very fast.

NextSibling (x)**:** Gives the next sibling of x, assuming it exists.

PrevSibling (x)**:** Gives the previous sibling of x, assuming it exists.

TaggedDesc (x, tag)**:** Gives the first node tagged tag with larger preorder than x and within the subtree of x (or returns -1 if there is none). Requires $rank/select$ in $Tags$, $O(\log\log t)$ time and usually constant.

TaggedPrec (x, tag)**:** Gives the last node tagged tag with smaller preorder than x and not an ancestor of x (or returns -1 if there is none).

TaggedFoll (x, tag)**:** Gives the first node tagged tag with larger preorder than x and not in the subtree of x (or returns -1 if there is none).

Note we can in general find fast the next occurrence of any tag after any position, which is important for fast XPath evaluation.

Finally, some book keeping functions:

PrevText (x)**:** Gives the document identifier of the text to the left of node x (and returns -1 if x is the root node or if the text is empty/non-existing). For instance, application of PrevText to the node identifier of the b-node in the XML instance `<a>text1<b>text2</b>text3</a>` returns the document identifier of "text1".

NextText (x)**:** Gives the document identifier of the text to the right of node x (and returns -1 if x is the root node or if the text is empty/non-existing). Application of NextText to the b-node in `<a>text1<b>text2</b>text3</a>` returns the document identifier of "text3".

MyText (x)**:** Gives the document identifier of the text below the node x (and returns -1 if x is not a leaf node if the text is empty/non-existing). Application of MyText to the b-node in In `<a>text1<b>text2</b>text3</a>` gives the document identifier of "text2".

TextXMLId (d)**:** Gives a unique number that is consistent with the preorder of the position of the document d in the tree consisting of all tree nodes and all text nodes. For instance, application of TextXMLId to the document identifier of "text1" in `<a>text1<b>text2</b>text3</a>` gives a number that is larger than NodeXMLId of the a-node, but is smaller than NodeXMLId applied to the b-node; e.g., it could give 2 for "text1" and 4 and 5 for "text2" and "text3", respectively.

NodeXMLId (x)**:** Gives a unique number that is consistent with the preorder of node x in the tree consisting of all tree nodes and all text nodes. For instance, application of NodeXMLId to the b-node in `<a>text1<b>text2</b>text3</a>` gives a number that is larger than TextXMLId of "text1" and is smaller than TextXMLId of "text2"; e.g., it could give 3 to be consistent with the numbers of the example given in TextXMLId.

2.5 Text Searching

These are operations to query the text contents. Except for the first, they are supposed to return all the document identifiers that match a string query, in document order. There are four versions of each query: existential (is there a match?), counting (how many matches), document reporting (give document numbers, with an iterator), full reporting (give pairs (doc, pos) of document numbers and byte offset within the document). The search is done with the backward search technique on the BWT of the collection; the whole structure is called an FM-index [FM05]. This includes a sampling for locating; we will put a sampled each l text positions (apart from *Doc*, which solves the problem for the text limits). At those samples we put the document number and offset, for $O(u \log(u)/l)$ extra bits. More precisely, we will store a bitmap aligned to the bwt marking the sampled positions, and an array similar do *Doc* giving document number and offset.

Time is $O(|s| \log \sigma)$ for the search, plus an extra to report we detail next, plus sorting the results in document order.

ParentNode (d)**:** Gives the parent node of document identifier d. For the rest, it is a matter of going to $Par[d]$, and if it is a `"("` the answer is d, otherwise it is $parent(d)$. Constant time.

Prefix (s)**:** Search for documents prefixed by string s. After the normal backward search (time $|s| \log \sigma$), there will be several $'s in the bwt range. Now we map to *Doc* using $rank_\$$ and can answer exist/count in $O(1)$ time, or report each occurrence, in *some* order (indeed, lexicographic, but not document order), in $O(1)$ time per occurrence.

Suffix (s)**:** Start the search with the $, and continue with backward search for s. Now have to check one by one in the final range, continuing the LF until reaching a $ or a position sampled for locating (this is part of a normal FM-index). Cost is $O(l \log \sigma)$ per answer. For existential and counting query, one can still use the range in L after the search.

Equal (s)**:** Start as suffix, then map at the end to the $'s. Cost is like $Prefix$.

Contains (s)**:** Normal backward search, completed like the suffix. Has the additional problem of, for all but full reporting queries, having to filter out the occurrences that fall within the same document, being the cost proportional to that number of occurrences. Improving this seems to require using much space.

LessThan (s)**:** Similar to a prefix search, but using only the ep of the search, sp is always 1. If at some point there are no occurrence of c within $[1, ep]$, have to 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.

Note that in case we do not store empty nodes in the bwt, *Doc* arrays store the original document numbers, and also we store a bitmap telling which documents are nonempty. This is sufficient for all purposes. A more general mechanism is to not index (yet store in the bwt) the contents of some texts (e.g. composed of separators). In this case we need another bitmap aligned to the $'s of the bwt, telling which are indexed.

2.6 Displaying Contents and Other Services

Given a node x, we want to display its text (XML) content, i.e. return the string. We traverse the structure starting from $Par[x]$, retrieving the tag names and the document contents, from the document identifiers. The time is $O(\log \sigma)$ per text symbol and $O(\log \log t)$ per tag.

GetText (d)**:** Gives a letter by letter iterator through the text with doc id d (in reverse order).

GetSubtree (x)**:** Generates the subtree at node x.

3 Parsing XML into our Data Structures

In order to support XPath queries, we need to represent XML documents faithfully with respect to the XPath data model. In this data model, an XML document is represented as an unranked, ordered tree, in which each node has one of the following *seven types*: root, element, attribute, text, comment, processing-instruction, or namespace.

Since our low-level data structures only have two different types of nodes, tagged and text, the type information about non-text nodes is stored by means of special tags not valid in XML documents, to avoid any collision. Consider for instance an element node that has an attribute with name "date" and value "11-11-2007". We store all attributes under a special node labeled `<<@>>`, which we insert as first child of the element. Each attribute is represented as a two nodes subtree under `<<@>>`, consisting of an element node which is labeled by the attribute name, and a text node that contains the attribute value. For instance, `<a date="11-11-2007" att2="foo"><b/><c/></a>` is represented by a tree of the following form

```
<a><<@>><date>11-11-2007</date><att2>foo</att2></<@>><b/><c/></a>
```

This encoding is the most suitable for our needs. Indeed, it allows us to skip all the attributes of an element quickly by using the fast primitives **FirstChild** and **NexSibling**. Then, it is fairly easy to rewrite XPath queries syntactically: `/a/@date` becomes `/a/<@>/date::text()` for instance. This internal encoding can be kept all the way through the query execution and only needs to be handled when serializing the results. Section 4.2 details how the automaton model can conveniently express tests such as "all nodes but attributes" within a single transition.

An XML document has one unique root node which is the parent of the document element node, and which carries no queriable label information. We can conveniently attach the XML document name to this node, in order to support the representation and querying of multiple XML documents.

A SAX parser is used to process the input XML document. Calls to the storage-level construction interface (cf. Section 2.2) are made in response to events reported by the parser, as follows.

Start Document: Invokes `New child(documentName)` to construct the root node.

> **Qu:** In Sec. 2.2, the `New child` operation has the tag id as its input parameter. It seems as though it should instead be the tag name, since the tag name-tag id mapping is stored inside the engine by the TagName table, and it's only the tag names that we otherwise refer to outside the engine.
>
> **Answ:** This is correct, the TagName table is constructed by the engine. Note, however, that after construction the engine will make the TagName table *public*; this is needed because the traversal functions need tag id's, not tag names.

End Document: Notifies the interface that parsing has completed, and that structures can be saved to disk.

> **Qu:** It's not clear how the storage engine knows that document parsing has completed. To handle it as described above, we'd need to add another method to the construction interface. But if tree depth is monitored inside the engine somehow, it should be able to automatically figure out when the document root node has been closed without an explicit call from us.

Start Element *(uri, localName, qName, attributes)*: The element node is constructed with a call to `New child(`*`qName`*`)`. If *uri* is a non-empty string, then invoke `New child(':' + `*`uri`*`)`. For each attribute entry in *attributes*, `New child(`*`attName`*`)` is invoked, followed by `New text(`*`attValue`*`)`, and `Node finished`, where *attName* and *attValue* represent the name and value of the current attribute entry.

End Element: Invokes `Node finished` to close off the current element node.

Characters(*string*): Invokes `New text(`*`string`*`)` to insert a text node as a child of the current element node.

Comment(*string*): Invokes `New child('!' + `*`string`*`)` followed by `Node finished`.

Processing Instruction(*target, data*): Invokes `New child('?' + `*`target`*`)`, followed by `New text(`*`data`*`)` and `Node finished`.

Note that under this strategy, a parent-child relationship exists between each attribute node and its enclosing element node. In the XPath data model, however, this relationship is one-way: the element node is the parent of its attribute nodes, but attributes are not considered to be children of the element node. The same holds for namespace nodes. Such inconsistencies will need to be managed by the XPath evaluator.

Additionally, occurrence counts for each distinct element tag name and attribute name are calculated during the parsing process. Such statistics are later used to perform XPath query optimization.

Qu: These calculations could either be done inside the storage engine, based on the number of calls to `New child` for each tag/attribute name, or we could do so ourselves outside the storage engine.

4 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. Since all tree navigation and text searching operations are supported within $O(\log n)$ time, we wish to support a subset of XPath which can be evaluated in $O(|Q| \log n)$ too, per result node being reported; by default, results are returned as iterators over node identifiers (in document order). Here $|Q|$ denotes the size of the query Q, that is, the number of nodes in Q's parse tree.

As a first shot we will support the "Core XPath" subset [GKP05] of XPath 1.0. It supports all 12 navigational axes, all node tests, and filters with navigational predicates and Boolean operations (and, or, not). A node test (nonterminal NodeTest below) is either the wildcard ('*'), a tagname, or a nodetype test, i.e., one of comment(), text(), processing-instruction(), or node(). 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 data values, i.e., its final axis is either the attribute- or the text-axis, 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.

$p = w$ (equality): tests if a string specified by p is equal to the string w.

contains (w, p): tests if the string w is contained in a string specified by p.

starts-with (p, w): tests if the string w is a prefix of the string specified by p

4.1 Top-Down and Bottom-Up Evaluations

The standard idea of top-down evaluation is to go through the expression parse tree and to compute for each path step the (intermediate) set of selected nodes, starting from the set obtained by the previous step. We rely on an efficient representation of sets of nodes. For instance, consider the query //a//b which selects all b-descendants of a-nodes. Our evaluator generates the call TaggedDesc(Root,t_a), where t_a is the tag identifier of a. According to Section 2.4, this takes time

$O(\log\log t)$. Note that the resulting iterator describes a set of nodes of size at most n. Next, we iterate through these nodes u and concatenate TaggedDesc(u, t_b) to the result iterator. This takes $O(n \log\log t)$ (usually $O(n)$ in practice) time. Altogether we obtain an algorithm which runs in time $O(|Q|n \log\log t)$. This is slower than (but in practice similar to) the standard core XPath algorithm by Gottlob, Koch, and Pichler, which runs in time $O(|Q|n)$ and is based on intersection of nodes sets for each path step in the query. On a real machine, our algorithm might still perform better though, because of cache behaviour and the small space required for our data structures.

The idea of bottom-up evaluation is to select certain leaves of the XML tree and then proceed upwards towards the root node while checking further constraints of the query. In common implementations of XPath, this method is rarely used, because it requires direct access to the leaf nodes. In our setting, we do have direct access to leaf text nodes: consider the query //text()[starts-with(.,"b")] which selects all text nodes that start with a "b". In top-down evaluation, we traverse through all nodes of the tree, and once we encounter a text-node check whether it starts with a "b". Thus, $O(n)$ time is needed, even if the number m of result nodes is small. If we use bottom-up evaluation, then we use **Prefix**("b") and obtain in $O(1)$ time the first result and in $O(m)$ all the m result nodes. Similarly, for a query such as Qtop = /a/b[starts-with(text(),"blah")] we iterate through text nodes that start with "blah", and for each one check whether its parent node is labeled "b" and the parent of that is labeled "a" (and is the document element). This takes time $O(m')$, where m' is the number of text nodes that start with "blah".

Which of the two is more efficient, top-down or bottom-up evaluation? This depends on the actual numbers of selected nodes. Consider Qtop. If the number n of nodes is very large, but the number α of children of the document element "a" is small, and the number β of text-children of /a/b-nodes is small too, then top-down evaluation of Qtop will be faster than bottom-up evaluation (because it runs in $O(\alpha + \beta)$ time, versus $O(m')$ required for bottom-up). In general, we propose the following thumb rules: 1) if the query consists of a top-down path of the form /a/b/... without any //, then we propose to do top-down evaluation. Otherwise, i.e., if it contains //-axes, then we propose bottom-up evaluation. 2) Finer grained decisions can be obtained by taking selectivities into account. For instance, we can pay $O(|"blah"| \log \sigma)$ to determine the number of text-nodes that start with "blah". If this number is small, then we would still prefer bottom-up evaluation over top-down, even if only /-axes are in the query (thus overriding the first thumb rule). Thus, in general, we will pay about the minimum of the times needed for top-down and bottom-up evaluation. In general, bottom-up versus top-down decisions can be done dynamically, during evaluation.

Note that everything said above concerning how to search text nodes, also applies to the search of attribute values, because they are stored in the same way as text nodes.

Qu: is it possible to apply TaggedDesc to a set of nodes of size m in time $O(\log m \log t \log n)$?

4.2 Tree Automata Representation

One important factor which must be taken into account when considering top-down versus bottom-up evaluation is the size of transient data-structures. To achieve both runtime and memory efficiency, we tailor the classical definition of tree automaton [CDG+07] to our needs. Let us consider an XPath query Q. Then a tree automaton is a tuple $(\Sigma, \mathcal{Q}, \mathcal{I}, \mathcal{A}ny, q_m, \Delta)$, where :

- Σ is the set of symbols
- $\mathcal{Q}$ is the set of states

- $\mathcal{I} \subseteq \mathcal{Q}$ is the set of initial states
- $\mathcal{A}ny \subseteq \mathcal{Q}$ is a set of *ignore* states
- $q_m \in \mathcal{A}ny$ is a special marking state
- $\Delta : \mathcal{Q} \times 2^{\Sigma} \rightarrow 2^{\mathcal{Q}} \times 2^{\mathcal{Q}}$ is the transition function.

Σ, $\mathcal{Q}$ and $\mathcal{I}$ correspond to the classical definition of a tree automaton. Of course, neither $\mathcal{Q}$ nor Σ need to be stored in the data structure, since they are "universal" sets, in the sense that every state ever created for the purpose of query evaluation will belong to $\mathcal{Q}$ and every tag belongs to Σ. $\mathcal{I}$ however is needed for both top-down and bottom-up runs. $\mathcal{A}ny$, the set of ignore states is kept for optimization purposes. Indeed, in the classical definition of automaton, only leaf states are final ones. For instance, the top down automaton, which checks whether the root node is an "`a`" tag would first take a successful transition labeled "`a`" from some initial state q_0 to a pair of states (q_1, q_1). It then needs to reach the leaves of the subtree to be in a final state, hence it must have "dummy" transitions $q_1 \times \{t\} \rightarrow (\{q_1\}, \{q_1\})$ for all $t \in \Sigma$, and where q_1 is a final state, the recognition stopping once the automaton reaches the leaves of the tree. Such "complete" automata are needed in theory to perform operations like determinization (for the bottom-up case), minimization, complementation, and so on. However, from our point of view, this dummy tree traversal is not needed, hence the special semantics given to ignore states: if the automaton is in an ignore state, then it accepts any input tree without checking it. One of these ignore states, the marking state q_m has a special role. If the automaton takes a transition for which one of the destination states is q_m, then the current subtree is pushed into the result set of the query.

Lastly, the transition function is a bit more involved than in a the classical model. Note that it is an intrinsic feature of XML, that the set of potential element tags that appear in a document is infinite and not known a priori; this is in contrast to conventional string automata implementations (e.g., in `grep`) where only legal characters need to be considered. Indeed, instead of being labeled by tags, they are labeled by a possibly infinite set of tags, meaning that the transition is taken if the input tag is in this set. However, this set of tags is also co-finite, meaning that it can be represented in a finite way. Indeed, either the set is finite in which case it is only a collection of tags or it is infinite but co-finite, in which case we store its complement (this idea is commonly used for set representations, see e.g., the CDuce code at `http://www.cduce.org/`) For instance the following XPath tests are represented as such:

- $\{$`'a'`$\} = \{$`'a'`$\}$, the set matching only `<a>`
- `node()` $= \overline{\{\texttt{<@>}\}}$, the set matching any node
- `*` $= \overline{\{\texttt{<@>}, \texttt{<\$>}\}}$, the set matching any XML elements but not text nor attributes

The interest of this extension is twofolds. It allows us to merge transitions easily either in the case of disjunction or conjunction. For instance consider : `child::*/self::a` such a path is equivalent to `child::a`. The automaton gives us this simplification for free since at construction time, the transition created is labeled : $\overline{\{\texttt{<@>}, \texttt{<\$>}\}} \cap \{\texttt{a}\}$ which is simply $\{\texttt{a}\}$.

Let us now briefly discuss how to make top-down and bottom-up recognition efficient. First off, the automaton we build is non-deterministic. Of course, determinizing it before running the query would not be a good idea, because this is potentially exponential in the size of the query and furthermore, it needs a complete automaton. It is also common knowledge that a top-down automaton cannot always be determinized. However, we use top-down recognitions for forward navigational paths, that is paths without predicates. Such paths can always be executed by a deterministic top-down automaton (which intuitively corresponds to the fact that they can be executed in streaming). Instead of completely determinizing the automaton once and for all, we keep a non-deterministic one which we determinize on the fly. Indeed, it is possible to interleave the well-known subset-construction with the recognition one. This works for top-down as well as bottom-up runs. The evaluation function now does not handle states but sets of states.

From a data-structure point of view, we need to manipulate sets of states and sets of tags (on transitions), for which we want fast set operations as well as fast membership, emptiness test, etc. Indeed, states of the automaton are merely identifiers and tags occurring in an XPath query can be hashed and mapped to a unique integer (exactly this is done in our low level interface by means of the $TagName$ table). An important remark is that the number of states as well as the number of tags is bounded by the size of the query Q In practice, this number is small enough so that sets of such integers can be efficiently represented as the bits of a machine-size integer (nowadays, the norm is 64 bits which allows queries of up to 64 steps, which seems unlikely for hand written XPath expressions). Thus in practice, all the aforementioned operations can be done in constant time by using binary arithmetic on integers. In our preliminary tests, using immutable balanced binary trees gives better results than using bitvectors of size larger than 64 bits. This is because immutable binary trees allow for much more data-sharing and are memory efficient when the stets are sparse, which often happens in our case.

From an algorithmic point of view there are many optimizations to perform, besides the on-the-fly determinization that we already mentioned. The first one, which is a logical extension is *lazy* determinization [GGM+04]. Suppose we have the two following transitions in the automaton:

$$\begin{array}{l} q_0, \{\mathtt{a}\} \to \{q_1\}, \{q_2\} \\ q_0, \mathtt{*} \to \{q_1\}, \{q_3\}. \end{array}$$

If the input tree has tag `'a'` then both can be taken, hence we compute a set of successors for $q_0, \{\mathtt{a}\}$ to be $\{q_1\}$ and $\{q_2, q_3\}$. This is the idea behind on-the-fly determinization. Now we further optimize this by using memoization, that is storing the result of this computation, since it will always be the same. In practice, it consists in replacing both transitions with:

$$\begin{array}{l} q_0, \{\mathtt{a}\} \to \{q_1\}, \{q_2, q_3\} \\ q_0, \mathtt{*} \setminus \{\mathtt{a}\} \to \{q_1\}, \{q_3\} \end{array}$$

Then, the next time the automaton is in state q_0, the choice is fully deterministic and computing the successors sets is not needed anymore. This situation happens very often in practice, since the above transitions are the same as the one encoding steps such as `//a` or `following-sibling::a`.

One last optimization, which is documented in [Hos] targets bottom-up runs. Indeed, bottom-up runs will usually create bigger sets of successors than top-down runs. The idea is then to perform bottom-up recognition with top-down filtering. When creating the successors sets in a bottom-up run, we filter out the states which cannot be the result of the equivalent top-down run. Such a technique always optimizes the bottom-up recognition and is fully detailed in [Hos]. More involved top-down and bottom-up interleaving based on heuristics or more complex decisions are sketched in Section 4.5.

4.3 Optimization through Pruning and Partitioning

Consider again the query //a//b of before. Of course, the evaluation can be optimized by applying TaggedDesc(u, t_b) only to top-most independent a-nodes in the document tree, instead of applying it to *all* a-nodes. Hence, after locating an a-node u and concatenating its b-descendants to the result set, we skip all following nodes of the iterator TaggedDesc(Root,t_a) that are descendants of u, using TaggedFoll(u, t_a).

This type of optimization can be carried out for each path-step that is applied to a set of nodes. For instance, to compute the following-nodes of a set of nodes S, it suffices to select the unique left-most, lowest node of S, or, more precisely, the nodes of S with the lowest post-value. Recall that the following-nodes of a node u are all nodes to the right of u in the tree, besides its ancestors and descendants. We can use Postorder(x) to determine the node in S with minimal post-value.

As a last example, consider computing the ancestors of a set of nodes S. We compute the ancestors only of the lowest independent nodes of S. However, this might still involve some duplicate work, because these nodes may have common ancestors. This can be avoided by starting with the right-most node in S and stopping the ancestor computation as soon a node is reached which has a descendant in S. In this way, the ancestors of S-nodes are partitioned. Using pruning and partitioning, no result node will be computed more than once [GvKT03].

4.4 Optimization using Selectivities

The selectivity of a query is the size of its result set, i.e., the selectivity of an XPath query Q is the number of nodes selected by Q. Through the SubtreeTags (x, tag) operation of Section 2.4 we obtain in time $O(\log\log t)$ the selectivity of the query $//tag$, applied to the node x. Consider again the query //a//b, and a tree that has many a-nodes but only few b-nodes. Instead of iterating through (top-most independent) a-nodes and selecting their b-descendants, it will be much more efficient to iterate through (bottom-most independent) b-nodes and select those which have an a-ancestor.

For queries involving data value comparisons (such as =, contains, and starts-with), it might in general be most efficient to *first* compute the occurrences of the corresponding text nodes using the operations of Section 2.5, because the number of matches is expected to be relatively small with respect to n. And then to check if those text nodes satisfy the rest of the query. It would be helpful if we had an operation similar to SubtreeTags, but for text search, which determines the *number of text nodes* that satisfy the search. This is already mentioned in Prefix(s): we can answer count-queries fast. The same probably holds for Equal-queries. Having these count-queries will allow us to make informed decision concerning the evaluation order (i.e., whether to start from the text and then check the tree, or the other way around).

Qu: Can we also support count-queries for Contains(s) in time $O(\log n)$? **Answ:** Only for the total number of occurrences, not total number of documents.

For typical XPath queries there are some thumb rules on how to use selectivities [MBB$^+$06], but a fully fledged query optimizer based on selectivity analysis is out of the scope of this project. The aim here is to keep the XPath fragment small (but practical) so that static (and selectivity based) optimizations can be kept simple. If more powerful fragments of XPath are to be supported, then more optimization techniques will be required, such as query rewriting prior to evaluation (see, e.g., [BKMH06]).

4.5 Automata-Based Mixed Evaluation

Our currently proposed evaluator supports both, top-down and bottom-up evaluation. This is done by compiling the query into an intermediate representation of tree automata. Such automata can be executed both bottom-up and top-down. During evaluation, we dynamically use count queries (which typically cost little time to determine whether to proceed top-down or bottom-up, in the current subtree.

Acknowledgement

We would like to thank Kim Nguyen who is implementing the XPath evaluator for writing the section about tree automata, and for many insightful discussions about the topic.

5 XQuery Support

The XQuery subset Q is characterized as follows. Let p and c be a "Core" and a "Core+" XPath expression, respectively. We first define a "Context Core" $\$x\ p$ XQuery expression and a "Context Core+" $\$x\ c$ XQuery expression as follows: let $\$x$ be a binding variable that provides the context nodes of the XPath expression, $\$x\ p$ consists of applying the path expression p to the context nodes given by $\$x$, and $\$x\ c$ consists of applying the path expression c to such context nodes. If no binding variable is given, the above expressions $\$x\ p$ and $\$x\ c$ will be evaluated on the current context (in XQuery the current context may be given by an external variable).

An XQuery expression $q \in Q$ is: (1) any **concatenation** of Context Core or Context Core+ expressions: $\$x\ p_1, \ldots, \$x\ p_n$ or $\$x\ c_1, \ldots, \$x\ c_n$; (2) given a tag t, an **element constructor** $\langle t\rangle\{q\}\langle/t\rangle$; (3) a **boolean expression** bq of the kind $p_1 = p_2$, with p_1, p_2 being a Context Core or a Context Core+ XPath expression, or any conjunction, disjunction or negation of such boolean expressions. bq can also be a boolean expression that returns the result of testing a value expression with another value expression or a string, as explained in Sect. 4 (thus, of the kind $p = s$ (**equality**), **contains** (p, s), **starts-with** (p, s) and **ends-with** (p, s)). (4) a **FLWR expression** xq of the kind:

$$\boxed{xq}\quad \begin{array}{l} \text{for } \$x \text{ in } p_1,\ \$x_1 \text{ in } \$x\ p_2,\ \ldots, \$x_k \text{ in } \$x_{k-1}\ p_{k+1} \\ \text{where } bq \\ \text{return } q \end{array}$$

where $p_1, \ldots, p_{k+1}$ are Core or Core+ expressions, that become Context Core or Context Core+ expressions when preceded by a binding variable. A return clause may contain other for-where-return queries, nested and/or concatenated and/or grouped inside constructed elements.

```
Q1:  FOR $m IN doc("movies.xml")//movie,
         $r IN $m/rating
    WHERE $r/text() = 'PG'
    RETURN { ⟨moviesummary⟩
                 ⟨movietitle⟩
                     $m/title
                 ⟨/movietitle⟩
```

```
            ⟨/moviesummary⟩ }
```

In order to evaluate the previous query, we need to build the corresponding parse tree expression. In particular, as we can see, the XQuery query consists of several XPath expressions, connected to each other by binding variables. We need to identify such path expressions and build a **generalized tree pattern** [CJLP03], which describes the parse tree based on its components, i.e. the Core and Core+ XPath expressions. Given a query q, we can build for such a query a generalized tree pattern GTP, which is a tree T with nodes labeled by variables, together with a boolean formula F specifying constraints on the nodes and their properties, including their tags, attributes, and contents. The tree consists of two kinds of edges parent-child (pc) and ancestor-descendant edges. The former are represented as single lines, and the latter are represented as double lines. Moreover, we can have compulsory edges (solid lines) or optional edges (dashed lines), according to which clause the connecting nodes are in. Optional edges are those connecting nodes in a RETURN clause: such nodes are optionally matched, whereas the nodes belonging to the other clauses are mandatory matched. Nodes belonging to the same FLWR expression are also labeled with a group number. We assume that this number is 0 for FOR/WHERE clauses, 1 for LET clauses and 2 for RETURN clauses. The group number is useful to represent nested subqueries and nested tree patterns. For instance, the generalized tree pattern for the above query is represented in Figure 1.

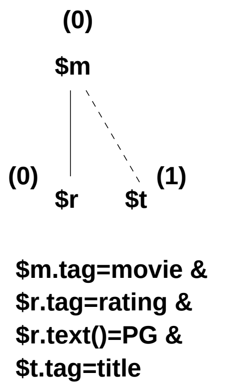

Figure 1: **Example with GTPs for Query Q1.**

If we take the previous query and expand it to a nested XQuery query, the resulting GTPs are represented in Figure 2.

```
Q2:  FOR $m IN doc("movies.xml")//movie,
         $r IN $m/rating
     WHERE $r/text() = 'PG'
     RETURN { ⟨moviesummary⟩
                 ⟨movietitle⟩
                     $m/title
                 ⟨/movietitle⟩
                 ⟨movieyear⟩
                     $m/year
```

```
    ⟨/movieyear⟩
    ⟨movieactors⟩
        FOR $a IN $m//a
        RETURN { ⟨actor⟩
        $a/firstname/text()
        $a/lastname/text()
        ⟨/actor⟩ }
    ⟨/movieactors⟩
⟨/moviesummary⟩ }
```

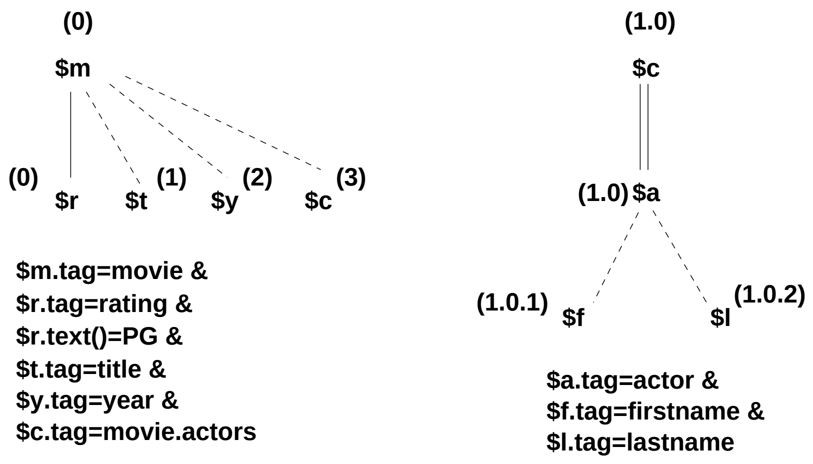

Figure 2: **Example with GTPs for Query Q2.**

6 XQuery Add-Ons

Given our fast primitives for text searching, that have logarithmic cost in practice, it is possible to evaluate string range queries and handle XQuery boolean functions of the kind **compare** (s_1, s_2). Compare would use the **LessThan** (s) and **Equal** (s) primitives. This would let us answer string range queries in the compressed domain, as in our previous work [ABMP07], but much faster.

Qu: Another suitable extension that we may think of is XQuery full-text, which would probably come for free thanks to our text searching primitives. Examples of such queries are those with boolean predicate **ft:contains** (s) with filters of the kind //movie/subtitle[. ftcontains "Toot*" ftand "Remake" ordered distance at most 2 words at start], meaning that we want to retrieve all movies with subtitle containing 'Tootsie' and 'Remake' within at most 2 words distance.

References

[ABMP07] A. Arion, A. Bonifati, I. Manolescu, and A. Pugliese. Xquec: A query-conscious compressed xml database. *ACM Trans. Internet Tech.*, 7(2):1–35, 2007.

[BKMH06] M. Brantner, C. Kanne, G. Moerkotte, and S. Helmer. Algebraic optimization of nested XPath expressions. In *ICDE*, page 128, 2006.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[CDG$^+$07] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, C. Löding, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007.

[CJLP03] Z. Chen, H.V. Jagadish, L.V.S. Lakshmanan, and S. Paparizos. From Tree Patterns to Generalized Tree Patterns: On Efficient Evaluation of XQuery. In *VLDB*, pages 237–248, 2003.

[FM05] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 54(4):552–581, 2005.

[GGM$^+$04] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29:752–788, 2004.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *Proc. SODA*, pages 841–850. SIAM, 2003.

[GKP05] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *ACM Trans. Database Syst.*, 30:444–491, 2005.

[GMR06] A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[GvKT03] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational dbms to watch its (axis) steps. In *VLDB*, pages 524–525, 2003.

[Hos] H. Hosoya. Foundations of XML processing. In preperation, see http://arbre.is.s.u-tokyo.ac.jp/ hahosoya/xmlbook/.

[Man01] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[MBB$^+$06] N. May, M. Brantner, A. Böhm, C. Kanne, and G. Moerkotte. Index vs. navigation in XPath evaluation. In *XSym*, pages 16–30, 2006.

[MR97] I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[RRR02] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. SODA*, pages 233–242, 2002.

[Sad08] K. Sadakane. The ultimate balanced parentheses. Technical report, Dept. of Computer Science and Communication Engineering, Kyushu University, Japan, 2008.