# Path Queries on Compressed XML[*]

Peter Buneman    Martin Grohe    Christoph Koch

peter@cis.upenn.edu    grohe@dcs.ed.ac.uk    koch@dbai.tuwien.ac.at

Laboratory for Foundations of Computer Science
University of Edinburgh, Edinburgh EH9 3JZ, UK

## Abstract

Central to any XML query language is a path language such as XPath which operates on the *tree structure* of the XML document. We demonstrate in this paper that the tree structure can be effectively compressed and manipulated using techniques derived from *symbolic model checking*. Specifically, we show first that succinct representations of document tree structures based on sharing subtrees are highly effective. Second, we show that compressed structures can be queried directly and efficiently through a process of manipulating selections of nodes and partial decompression. We study both the theoretical and experimental properties of this technique and provide algorithms for querying our compressed instances using node-selecting path query languages such as XPath.

We believe the ability to store and manipulate large portions of the structure of very large XML documents in main memory is crucial to the development of efficient, scalable native XML databases and query engines.

## 1 Introduction

That XML will serve as a universal medium for data exchange is not in doubt. Whether we shall store large XML documents as databases (as opposed to using conventional databases and employing XML just for data exchange) depends on our ability to find specific XML storage models that support efficient querying of XML. The main issue here is how we represent the document in secondary storage. One approach [9] is to index the document and to implement a cache policy

for bringing subtrees of the document tree into main memory on demand. Another approach [8, 11, 19] is to store information about each node in the document tree in one or more tuples in a relational database. In both cases the structure is fragmented and substantial I/O is required to evaluate a complex path expression; and this increases with the size of the source document.

An alternative approach is to extract the text (the character string data) from the document and to store it in separate containers, leaving the bare structure, a tree whose nodes are labeled with element and attribute names. We shall call this structure the *skeleton* of the document. This separation of the skeleton from string data is used in the XMILL compressor [15], which, as internal model of data representation in query engines, is reminiscent of earlier *vertical partitioning* techniques for relational data [3] which have recently been resurrected [2] for query optimization. Even though the decomposition in XMILL is used for compression only, it is natural to ask whether it could also be used for enabling efficient querying.

The promise of such an approach is clear: We only need access to the skeleton to handle the navigational aspect of (path) query evaluation, which usually takes a considerable share of the query processing time; the remaining features of such queries require only mostly localized access to the data. The skeleton of an XML document is usually relatively small, and the efficiency of query evaluation will depend on how much of the skeleton one can fit into main memory. However for large XML documents with millions of nodes, the tree skeletons are still large. Thus compressing the skeleton, provided we can query it directly, is an effective optimization technique.

In this paper, we develop a compression technique for skeletons, based on *sharing of common subtrees*, which allows us to represent the skeletons of large XML documents in main memory. Even though this method may not be as effective in the reduction of space as Lempel-Ziv compression [22] is on string data, it has the great advantage that queries may be efficiently evaluated directly on the compressed skeleton and that the result of a query is again a compressed skeleton. Moreover, the skeleton we use is a more general structure than that used in XMILL, where it is a tree containing just tag and attribute information of nodes. Our skeletons may be used to express other properties of nodes in the document tree, e.g. that they

contain a given string, or that they are the answer to a (sub)query.

Our work borrows from *symbolic model checking*, an extremely successful technique that has lead formal hardware verification to industrial strength [16, 6]. Model checking is an approach to the verification of finite state systems which is based on checking whether the state space of the system to be verified satisfies, or is a model of, certain properties specified in some temporal logic. It has already been observed that checking whether the state space of a system satisfies a property specified in some logic and evaluating a query against a database are very similar algorithmic problems, which opens possibilities for transferring techniques between the two areas. The main practical problem that researchers in verification are facing is the "state explosion" problem – state spaces get too large to be dealt with efficiently. The most successful way of handling this problem is *symbolic* model checking. Instead of representing the state space explicitly, it is compressed to an ordered binary decision diagram (OBDD) [4], and model-checking is done directly on the OBDD rather than the original uncompressed state space. The success of this method is due both to the compression rates achieved and to the fact that OBDDs have nice algorithmic properties which admit model-checking.

Our compression of XML skeletons by subtree sharing can be seen as a direct generalization of the compression of Boolean functions into OBDDs. This enables us to transfer the efficient algorithms for OBDDs to our setting and thus provides the basis for new algorithms for evaluating path queries directly on compressed skeletons.

**Example 1.1** Consider the following XML document of a simplified bibliographic database.

```
<bib>
  <book>
    <title>Foundations of Databases</title>
    <author>Abiteboul</author>
    <author>Hull</author>
    <author>Vianu</author>
  </book>
  <paper>
    <title>A Relational Model for
          Large Shared Data Banks</title>
    <author>Codd</author>
  </paper>
  <paper>
    <title>The Complexity of
          Relational Query Languages</title>
    <author>Vardi</author>
  </paper>
</bib>
```

The skeleton is shown in Figure 1 (a). Its compressed version, which is obtained by sharing common subtrees, is shown in Figure 1 (b). It is important to note that the order of the out-edges is significant.[1]

---

[1]Throughout this paper, we visualize order by making sure that a depth-first *left-to-right* pre-order traversal of the in-
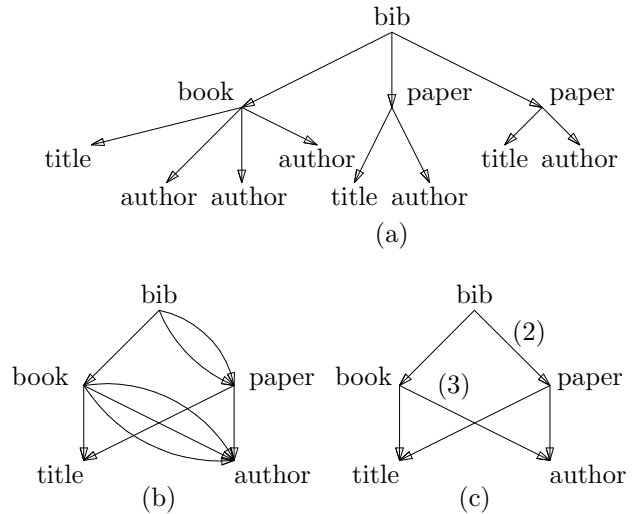


Figure 1: A tree skeleton (a) and its compressed versions (b) and (c).

Thus the original skeleton (a) can be recovered by the appropriate depth-first traversal of the compressed skeleton (b). Further compression can be achieved by replacing any consecutive sequence of out-edges to the same vertex by a single edge marked with the appropriate cardinality. Such a graph is shown in Figure 1 (c), where the unmarked edges have cardinality 1. □

This compressed representation is both natural and exhibits an interesting property of most practical XML documents: they possess a regular structure in that subtree structures in a large document are likely to be repeated many times. If one considers an example of extreme regularity, that of an XML-encoded relational table with $R$ rows and $C$ columns, the skeleton has size $O(C * R)$, and the compressed skeleton as in Figure 1 (b) has size $O(C + R)$. This is further reduced by merging multiple edges as in Figure 1 (c) to $O(C + \log R)$. Our experiments show that skeletons of less regular XML files, which contain several millions of tree nodes, also compress to fit comfortably into main memory. In several cases, the size of the compressed skeleton is less than 10% of the uncompressed tree. We should also remark that an interesting property of this technique is that it does not rely on a DTD or any other form of schema or static type system. Indeed we have evidence that common subtree compression can exploit regularity that is not present in DTDs.

Compressed skeletons are easy to compute and allow us to query the data in a natural way. Compressed and uncompressed instances are naturally related via *bisimulation*. Each node in the compressed skeleton represents a set of nodes in the uncompressed tree. The purpose of a path query is to select a set of nodes in the uncompressed tree. However this set can be represented by a subset of the nodes in a partially decompressed skeleton, and – as our experiments show – the amount of decompression is quite small in practice. Moreover there are efficient algorithms for producing

---

stances as depicted in our figures corresponds to the intended order of the nodes.

these partially decompressed skeletons, which we outline in this paper.

We use a large fragment of XPath as the query language of choice, and we use a simplified model of XML without attributes. These simplifications are not critical: the same succinct representation extends to full XML and can be used for more general query languages (including languages that support joins such as XQuery). We plan to generalize from the current work in the future.

In general, if the evaluation of a query can be performed in main memory, we expect it to outperform any query processor that needs to page in fragments of data from secondary storage. Compression that places large amounts of the most heavily accessed data into main memory in an easily manageable form is clearly valuable. Moreover, path query evaluation using our techniques is very efficient, and runs in linear time on uncompressed (i.e., tree) instances, with a small constant. On compressed instances, it can also beat existing main memory query processors that do not employ compression, for some computations need to be performed only once on a shared subtree.

The authors are aware of only two published works on querying compressed XML, XGRIND [20] and DDOM [18], which both simply compress character data in a query-friendly fashion. No attempt is made to compress the skeleton or to do any form of "global" compression. Our approach to representing data using bisimulation may appear reminiscent of graph schemata [5, 10] and data guides [13, 1] (these approaches use *simulation* rather than bisimulation) and index structures such as T-indices [17]. However, it must be stressed that these formalisms preserve neither structure nor order of a database instance; they are intended as auxiliary data structures for query optimization. In our model, we preserve a compact representation of the skeleton and of the results of queries.

**Technical Contributions**

The main contributions of this paper are as follows.

- We propose a novel approach to querying XML documents compressed by sharing subtrees of the skeleton tree, using path query languages such as XPath.

- We provide a formal description of the compression technique and the relationship – that of bisimulation – between compressed and uncompressed skeletons.

- We study algorithmic aspects of compression and primitive operations needed for evaluating path queries on compressed instances. We adapt techniques previously used in symbolic model checking – such as *OBDD reduction* [4] – to the XML setting of ordered, unranked, node-labeled trees and edges with multiplicities.

- From this we obtain an algebra of efficient query operators for compressed XML trees, which we use to evaluate queries of our XPath fragment.

- We demonstrate the feasibility and practical relevance of our approach by a prototype query engine implementation and a number of experiments:

We provide experimental evidence that subtree sharing is effective by testing it on a number of XML corpora.

We also provide evidence that queries over compressed skeletons are efficient and that their results, represented as partially decompressed skeletons, are typically not much larger than the compressed input skeletons.

By and large, the structure of the paper follows this order. Some proofs and illustrations had to be omitted here because of space limitations but will be presented in the long version of this paper.

## 2 Compressed Instances

In this section, we give a formal framework for our compression technique. We introduce a general notion of *instance*, which we use to model both the original XML documents and the compressed versions of these documents. We then define an appropriate notion of *equivalence* on these instances; equivalent instances are representations of the same XML document in possibly different states of compression. Using *bisimilarity relations*, we define a lattice structure on each class of equivalent instances. The maximum element of this lattice is the original XML document, whereas the minimum element is the fully compressed version of the document. We sketch a linear time compression algorithm. Finally, we show how to find common extensions of partially compressed instances that carry different information.

### 2.1 Instances

We construct instances from a set $V$ of vertices and a function $\gamma : V \to V^*$ which assigns to each vertex the sequence of its child vertices. Such a function immediately defines the edges of a directed graph: $(v, w)$ is an edge if $w \in \gamma(v)$. We shall require such graphs to be *acyclic* – there are no directed cycles, and *rooted* – there is just one vertex that has no incoming edge. Figure 2 (a) shows the same graph as in Figure 1 (b) with explicit names for the vertices. For this graph the set of vertices is $V = \{v_1, v_2, v_3, v_4, v_5\}$, and the function $\gamma$ is defined by

$$\begin{array}{rcl}
\gamma(v_1) & = & v_2 v_4 v_4 \\
\gamma(v_2) & = & v_3 v_5 v_5 v_5 \\
\gamma(v_4) & = & v_3 v_5 \\
\gamma(v_3) & = & \gamma(v_5) = \emptyset
\end{array}$$

If vertex $w$ occurs in the $i^{th}$ position of $\gamma(v)$ we write $v \xrightarrow{i} w$. Figure 2 (a) also shows these positions.

In addition to the graph we shall want to distinguish certain sets of nodes. For instance, we might want to indicate that all vertices of a certain set describe nodes in the source document with a given tag, or we might wish to indicate that a certain subset of nodes describes the answer to a query. In Figure 2, for example, we have used names such as $S_{\mathrm{bib}}$ to indicate that the vertices all correspond to vertices in the source

$S_{bib}$
$v_1$
1 2 3
$S_{book}$ $v_2$ 4 $v_4$ $S_{paper}$
1 2 3 1 2
$S_{title}$ $v_3$ $v_5$ $S_{author}$

(a)

$S_{bib}$
$w_1$
1 3
2
$S_{book}$ $w_2$ 4 $w_4$ $w_6$ $S_{paper}$
1 1
1 2 3 2 2
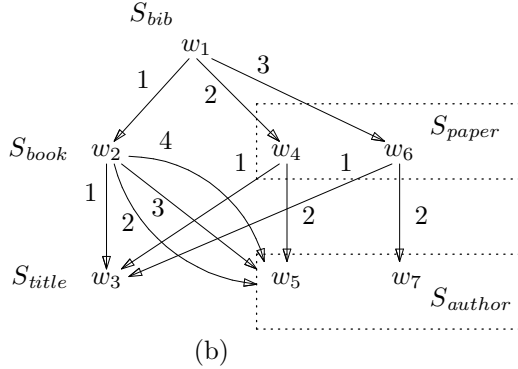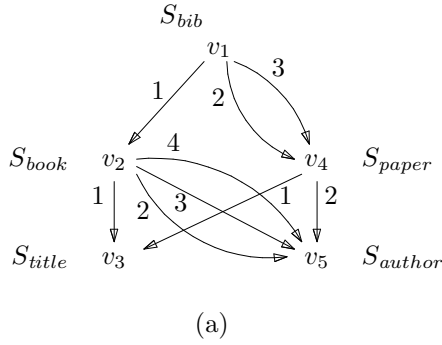$S_{title}$ $w_3$ $w_5$ $w_7$ $S_{author}$

(b)

Figure 2: Skeleton instances

tree with the same tag. However, Figure 2 (b) – a "partially decompressed" instance – has enough structure to distinguish papers written by "Vardi" (the vertex $w_6$) from the other papers, and this is a property that we may also want to be part of the schema.

Formally, a *schema* is a finite set of unary relation names. Let $\sigma = \{S_1, \ldots, S_n\}$ be a schema. A $\sigma$-*instance* is a tuple $\boldsymbol{I} = \left(V^{\boldsymbol{I}}, \gamma^{\boldsymbol{I}}, root^{\boldsymbol{I}}, S_1^{\boldsymbol{I}}, \ldots, S_n^{\boldsymbol{I}}\right)$, where

- $V^{\boldsymbol{I}}$ is a set of vertices.
- $\gamma^{\boldsymbol{I}} : V^{\boldsymbol{I}} \to (V^{\boldsymbol{I}})^*$ is a function whose graph is acyclic and has root $root^{\boldsymbol{I}}$.
- $S_1^{\boldsymbol{I}}, \ldots, S_n^{\boldsymbol{I}}$ are subsets of $V^{\boldsymbol{I}}$.

If the instance $\boldsymbol{I}$ is clear from the context, we often omit the superscript $^{\boldsymbol{I}}$. We say that $(V^{\boldsymbol{I}}, \gamma^{\boldsymbol{I}}, root^{\boldsymbol{I}})$ is the *DAG* of $\boldsymbol{I}$.

If $v_0$ and $v_n$ are vertices in an instance, and there are intermediate vertices $v_1, \ldots, v_{n-1}$ such that $v_0 \xrightarrow{i_1} v_1 \ldots v_{n-1} \xrightarrow{i_n} v_n$, we say that the integer sequence $i_1 \ldots i_n$ is an *edge-path* between $v_0$ and $v_n$. For each vertex $v \in V$ we define

$$\Pi(v) = \{P \mid P \text{ is an edge-path from } root \text{ to } v\},$$

and for a set $S \subseteq V$ we let $\Pi(S) = \bigcup_{v \in S} \Pi(v)$.

The two examples in Figure 2 have exactly the same set of edge paths from the root, and the paths that end in any set in the schema are also the same. Thus the

paths $\xrightarrow{2}\xrightarrow{2}$ and $\xrightarrow{3}\xrightarrow{2}$ give the vertex set for $S_{author}$ in both instances. This prompts our definition of equivalence of instances:

**Definition 2.1** *Two $\sigma$-instances $\boldsymbol{I}$ and $\boldsymbol{J}$ are* equivalent *if $\Pi(V^{\boldsymbol{I}}) = \Pi(V^{\boldsymbol{J}})$ and $\Pi(S^{\boldsymbol{I}}) = \Pi(S^{\boldsymbol{J}})$ for all $S \in \sigma$.*

In the original (fully uncompressed) skeleton, there is a unique edge-path from the root to each vertex (the edge-paths are addresses of nodes in the tree), and it is this observation that allows us to construct the tree from a compressed instance, by taking the node set of all edge paths from the root, a prefix closed set, as the vertices of the tree.

**Proposition 2.2** *For each instance $\boldsymbol{I}$ there is exactly one (up to isomorphism) tree-instance $\boldsymbol{T}(\boldsymbol{I})$ that is equivalent to $\boldsymbol{I}$.*

*Proof.* The vertices of $\boldsymbol{T}(\boldsymbol{I})$ are edge-paths in $\Pi(V^{\boldsymbol{I}})$. Uniqueness follows from the fact that for each vertex of a tree there is exactly one path from the root to this vertex. □

### 2.2 Bisimilarity Relations and Compression

A *bisimilarity relation* on a $\sigma$-instance $\boldsymbol{I}$ is an equivalence relation[2] $\sim$ on $V$ s.t. for all $v, w \in V$ with $v \sim w$ we have

- for all $i$, if $v \sim w$ and $v \xrightarrow{i} v'$ then there exists $w' \in V$ s.t. $w \xrightarrow{i} w'$ and $v' \sim w'$, and
- for all $S \in \sigma$: $(v \in S \iff w \in S)$.

Let us remark that this notion of bisimilarity relation coincides with the standard notion on transition systems with labeled transitions and states. In our setting the vertex labels are given by the unary relations in the schema.

If $\boldsymbol{I}$ is an instance and $\sim$ a bisimilarity relation on $\boldsymbol{I}$ then $\boldsymbol{I}/_\sim$ is the instance obtained from $\boldsymbol{I}$ by identifying all vertices $v, w$ with $v \sim w$ (that is, mapping them to their equivalence class w.r.t. $\sim$).

**Proposition 2.3** *For all instances $\boldsymbol{I}$ and bisimilarity relations $\sim$ on $\boldsymbol{I}$, $\boldsymbol{I}$ is equivalent to $\boldsymbol{I}/_\sim$.*

**Proposition 2.4** *For every instance $\boldsymbol{I}$ there is a bisimilarity relation $\sim$ on $\boldsymbol{T}(\boldsymbol{I})$ such that $\boldsymbol{I}$ is isomorphic to $\boldsymbol{T}(\boldsymbol{I})/_\sim$.*

*Proof.* The vertices of $\boldsymbol{T}(\boldsymbol{I})$ can be identified with the elements of $\Pi(V)$, which in turn can be identified with paths in the graph of $\gamma^{\boldsymbol{I}}$. We define two vertices to be in relation $\sim$ if the corresponding paths have the same endpoint. □

The bisimilarity relations $\sim$ on an instance $\boldsymbol{I}$ (as well as the instances $\boldsymbol{I}/_\sim$) form a lattice: The greatest lower bound of two bisimilarity relations $\sim$ and $\approx$

---

[2]Thus, it is also the case that for all $i$, if $v \sim w$ and $w \xrightarrow{i} w'$ then there exists $v' \in V$ s.t. $v \xrightarrow{i} v'$ and $v' \sim w'$.

is their intersection, and the least upper bound is the transitive closure of the union of $\sim$ and $\approx$. The equality relation is the minimum element of this lattice.

An instance $I$ is *minimal* if equality on $V$ is the only bisimilarity relation on $I$. Note that an instance is minimal if, and only if, the bisimilarity relation $\sim$ on $T(I)$ with $I = T(I)/\sim$ is the (unique) maximum element of the lattice of bisimilarity relations on $T(I)$. This immediately implies the following:

**Proposition 2.5** *For each instance $I$ there is exactly one (up to isomorphism) minimal instance $M(I)$ that is equivalent to $I$. There is no instance equivalent to $I$ with fewer vertices than $M(I)$.*

If a tree-instance $T$ represents an XML document, then $M(T)$ represents its compression. Other equivalent instances $I$ are "partially compressed".

**Proposition 2.6** *There is an algorithm that, given an instance $I$, computes $M(I)$ in linear time.*

In our implementation, we use an algorithm that traverses the original XML-tree only once in document order (post-order, corresponding to a bottom-up traversal, which however can be easily effected in a SAX parsing run using a stack holding lists of siblings for the path from the root to the current node) and maintains a hash table of nodes previously inserted into the compressed instance. Whenever a new node is to be inserted, its children have already been inserted and the redundancy check using the hash table can be made in (amortized) constant time.[3]

Compressed or partially compressed instances usually have *multiple edges* (see Figures 1 and 2) between pairs of vertices. Instead of representing these explicitly, in our implementation (discussed later on in this paper) for the compressed instances, we represent successive multiple edges by just one edge labeled by the number of edges it represents (see Figure 1 (c)). This implicit representation improves the compression rate quite significantly, because XML-trees tend to be very wide[4], which means that there are usually many parallel edges in the compressed instances. Indeed, all of the results in Sections 2 and 3 can be extended straightforwardly to DAGs in which each edge is labeled with a multiplicity.

## 2.3 Reducts and Common Extensions

In the data model discussed so far, instances are (possibly compressed versions of) ordered trees with node labels from a fixed alphabet, the schema $\sigma$. String data may be represented in this model in two ways.

The first is to make each string character a node in the tree, which has the disadvantages of considerably increasing its size and of worsening the degree of compression obtained. Moreover, this makes it difficult to use indices or specific algorithms for string search to deal with conditions on strings in queries.

---

[3]A strictly linear-time algorithm (which, however, needs more memory) is discussed in the long version of this paper.
[4]OBDDs, in contrast, are compressed binary trees.

The alternative is to have the schema only represent "matches" of tags and strings relevant to a given query. This requires to be able, given a compressed instance $I_1$ representing the result of a subquery and a *compatible instance* $I_2$ (obtained from the same tree but containing different labelings) representing e.g. the set of nodes matching a given string condition, to efficiently merge $I_1$ and $I_2$ to obtain a new instance $I_3$ containing the labelings of both $I_1$ and $I_2$. Together with the results of the next section, this will provide us with a method of combining fast tree-structure-based query evaluation with efficient string value-based search (using indexes on strings if they are available).

Let $\sigma' \subseteq \sigma$ be schemas. The $\sigma'$-*reduct* of a $\sigma$-instance $I$ is the $\sigma'$-instance $I'$ with the same DAG as $I$ and $S^{I'} = S^I$ for all $S \in \sigma'$. In the following, the $\sigma'$-reduct of a $\sigma$-instance $I$ will be denoted by $I|_{\sigma'}$.

Let $\sigma$ and $\tau$ be schemas. A $\sigma$-instance $I$ and a $\tau$-instance $J$ are *compatible* if the reducts $I|_{\sigma\cap\tau}$ and $J|_{\sigma\cap\tau}$ are equivalent. A *common extension* of $I$ and $J$ is a $\sigma \cup \tau$-instance $K$ such that $K|_\sigma$ is equivalent to $I$ and $K|_\tau$ is equivalent to $J$. Note that a common extension of $I$ and $J$ can only exist if $I$ and $J$ are compatible. Furthermore, if $I$ and $J$ are compatible then the $\sigma \cup \tau$-tree-instance $T$ with $T|_\sigma = T(I)$ and $T|_\tau = T(J)$ is a common extension of $I$ and $J$. This is the case because if $I$ and $J$ are compatible then the tree-instances $T(I)$ and $T(J)$ have the same $\sigma \cap \tau$-reduct.

**Lemma 2.7** *There is an algorithm that, given instances $I$ and $J$, computes a common extension of $I$ and $J$ in quadratic time.*

The construction in the proof of the lemma is the product construction for finite automata. It should be implemented as it is for automata, that is, only the states that have actually been reached should be constructed. This reduces the running time to being *linear in the size of the output* (and of course, the size of the output is at most as large as the size of the uncompressed instance). It can be shown that this construction always produces the least upper bound of the input instances in the lattice of bisimilarity relations of their (common) tree version. The construction can be easily extended to support edge multiplicities.

It is important to emphasize that the running time is linear in the size of the output, because of which it is also at worst linear in the size of the uncompressed tree-instance. Of course, in the worst case, this can still be quadratic in the size of the compressed input instance. Quadratic running time is only required if the input instances $I$ and $J$ compressed very well initially, and the common extension needed to accommodate the labeling information is much larger (which we assume will be rare).

## 3 Query Evaluation

Next, we study the problem of evaluating a Core XPath query on a compressed instance. Core XPath [14] constitutes a large, practical fragment of XPath, and subsumes a large variety of tree pattern languages (e.g. [7]).
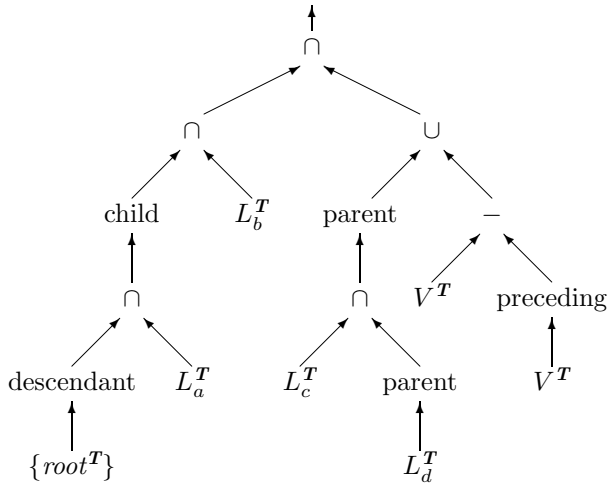
Figure 3: A query tree.

## 3.1 The Core XPath Language

We assume the notion of an *XPath axis* known and refer to [21] for reference. Let $T$ be a tree-instance. We define each axis $\chi$ that maps between ordinary tree nodes, i.e., each of the axes self, child, parent, descendant, descendant-or-self, ancestor, ancestor-or-self, following-sibling, preceding-sibling, following, and preceding, as a function $\chi : 2^{V^T} \rightarrow 2^{V^T}$ encoding its semantics (e.g., $n \in \text{child}(S)$ iff the parent of $n$ is in $S$; see Section 4 of [14] for a precise definition and efficient algorithms for computing these functions). As shown in [14], each Core XPath query can be mapped to an algebraic expression over

- node sets $S^T$ from the instance $T$,
- binary operations $\cup, \cap, -: 2^{V^T} \times 2^{V^T} \rightarrow 2^{V^T}$,
- axis applications $\chi$, and
- an operation $V|root(S) = \{V^T \mid root^T \in S\}$.[5]

A node set at a leaf of a query expression is either the singleton set $\{root^T\}$, the set of all nodes labeled with a certain tag, the set of all nodes containing a certain string in their *string value*, or the so-called *context* of the query (cf. [21]), a user-defined initial selection of nodes.

The intuition in Core XPath, which reduces the query evaluation problem to manipulating sets rather than binary relations, is to *reverse* paths in conditions to direct the computation of node sets in the query towards the root of the query tree.

**Example 3.1** Let $T$ be a tree-instance over the schema $\sigma = (\{root^T\}, L_a^T, L_b^T, L_c^T, L_d^T)$, where $L_l^T \subseteq V^T$ denotes the set of all nodes in $T$ labeled $l$. The Core XPath query

/descendant::a/child::b[child::c/child::d or

not(following::*)]

---

[5]This operation is needed for technical reasons, to support paths relative to the root node in conditions, as in /descendant::a[/descendant::b].

**procedure** downward_axis(vertex $v$, bool $s_v$)
```
{
1:     mark v as visited;
2:     v.S_j := s_v;

3:     for each w ∈ γ(v) do {
4:         bool s_w := (v.S_i ∨
               (s_v ∧ (axis is descendant or
                              descendant-or-self)) ∨
               (axis is descendant-or-self ∧ w.S_i));
5:         if(w has not been visited yet)
               downward_axis(w, s_w);
6:         else if(w.S_j != s_w) {
7:             if(w.aux_ptr = 0) {
8:                 create new node w' as copy of w;
9:                 w'.S_j := s_w;

10:                if(axis is descendant or
                           descendant-or-self) {
11:                    mark w' as not visited yet;
12:                    downward_axis(w', s_w);
                   }
13:                w.aux_ptr := w';
               }
14:            w := w.aux_ptr;
           }
       }
}
```

Figure 4: Pseudocode for downward axes.

is evaluated as specified by the query tree shown in Figure 3. (There are alternative but equivalent query trees due to the associativity and commutativity of $\cap$ and $\cup$.) □

It is known [14] that Core XPath queries $Q$ can be evaluated on a tree-instance $T$ in time $O(|Q| * |T|)$.

## 3.2 Operations on Compressed Instances

Next, we discuss query operations analogous to the Core XPath operations discussed earlier, but which work on compressed instances. As mentioned earlier, in certain cases, decompression may be necessary to be able to represent the resulting selection. Our goal is to avoid full de-compression when it is not necessary.

**Proposition 3.2** *There are linear time algorithms implementing the downward axes child, descendant-or-self, and descendant on compressed instances. Moreover, each such axis application at most doubles the number of nodes in the instance.*

*Proof.* The idea of the algorithm(s) is to traverse the DAG of the input instance starting from the root, visiting each node $v$ only once. We choose a new selection of $v$ on the basis of the selection of its ancestors, and split $v$ if different predecessors of $v$ require different selections. We remember which node we have copied to avoid doing it repeatedly.

Let $S_i$ be a node set of the input instance to which the axis $\chi$ is to be applied, creating a new selection $S_j := \chi(S_i)$. The schema of the output instance is

the schema of the input instance plus $S_j$. A (recursive) pseudocode procedure is given in Figure 4, which we initially invoke as downward_axis($root, root.S_i$) if $\chi$ is descendant-or-self and downward_axis($root$, false) otherwise. The first argument is the node to be processed – each node is only visited once – and the second argument, $s_v$, always passes a new selection down to a child. We assume the following data structures: each node has an associated bit "visited" and a handle "aux_ptr" for possibly linking to a copy. We represent selections $S_i, S_j$ of the instance as bits $v.S_i, v.S_j$ stored with the node. Initially, for all nodes, "visited" is false and the aux_ptr handles are "null".

The algorithm proceeds as follows. Starting with the root node, we assign a new selection and then traverse the list of children. We compute a new selection for each child using the formula of line 4. If a child has not been visited yet, we do that by a recursive invocation of downward_axis. Otherwise, we check whether the child has the desired selection (line 6). If this is not the case, we have to create a copy of the child, assign the desired (opposite) truth value as selection, and for the descendant and descendant-or-self axes recursively assure that also the reachable descendants will be selected. We also store a handle to the copy in the aux_ptr of the node in order to avoid to make multiple redundant copies. By copying a node $w$ and creating a new node $w'$, we mean that we also copy all the selections of $w$ w.r.t. the node sets of the schema and the links to children (but not the children themselves), $\gamma(w') := \gamma(w)$, and insert $w'$ into the modified instance.

It is easy to see that this algorithm leads to the desired result and that each node is copied at most once. If instances also contain edge multiplicities, the algorithm remains unchanged; these are completely orthogonal to downward axis computations. $\square$

Upward axes and the set operations never require us to split nodes and thus (partially) decompress the instance.

**Proposition 3.3** *The union, intersection, and set-theoretic difference operators, and the upward axes*

$$\chi \in \{self, parent, ancestor, ancestor\text{-}or\text{-}self\}$$

*do not change the instance. Moreover, there are linear-time algorithms implementing these operators.*

**Proposition 3.4** *There are linear-time algorithms implementing the following-sibling and preceding-sibling axes.*

The algorithm for upward axes is simple, as they do not change the instance DAG. Basically, all we need to do is to recursively traverse the DAG without visiting nodes twice, and to compute the selection when returning from the recursion. Algorithms for following-sibling and preceding-sibling are slightly more involved when edge multiplicities are taken into account, but not difficult. The semantics of the following and preceding axes can be obtained by composition

of the other axes (cf. [14]). Namely,

$$following(S) = descendant\text{-}or\text{-}self($$
$$following\text{-}sibling(ancestor\text{-}or\text{-}self(S)))$$

and

$$preceding(S) = descendant\text{-}or\text{-}self($$
$$preceding\text{-}sibling(ancestor\text{-}or\text{-}self(S))).$$

### 3.3 Evaluation of Core XPath

Given the axis operators of the previous section, query evaluation is easy: Each expression of the Core XPath algebra of Section 3.1 can be immediately evaluated on a compressed instance $\boldsymbol{I} = (V, \gamma, root, S_1, \ldots, S_n)$. Given that the node sets at the leaves of the expression are present in $\boldsymbol{I}$,

- an expression $S_k \circ S_m$ (where the binary operation $\circ$ is either $\cup$, $\cap$, or $-$ and $1 \leq k, m \leq n$) evaluates to instance $\boldsymbol{J} = (V, \gamma, root, S_1, \ldots, S_n, (S_k \circ S_m))$,
- an expression $\chi(S_k^I)$ evaluates to an instance $\boldsymbol{J}$ to which the new selection has been added and which possibly has been partially de-compressed to accommodate this selection, and
- $V|root(S_k^I)$ evaluates to an instance $\boldsymbol{J}$ to which the new selection $\{v \in V^I \mid root \in S_k^I\}$ has been added.

**Example 3.5** Consider the instance

$$\boldsymbol{I} = (V^I, \gamma^I, root^I, \{root^I\}, L_a^I, L_b^I)$$

of Figure 5 (a) and the query //a/b. In our query algebra, this reads as the expression

$$child(descendant(\{root^I\}) \cap L_a^I) \cap L_b^I.$$

We start by putting the operator applications of the associated expression tree into any total order; in this example, this order is unique. Then we process one expression after the other, always adding the resulting selection to the resulting instance for future use (and possibly partial decompression). We start with $D^{I_1} := descendant(\{root^I\})$. $\boldsymbol{I}_1$ is obtained from $\boldsymbol{I}$ by applying the descendant axis to $\{root^I\}$ (in this case, there is no decompression) and adding the new set $D^{I_1}$ to the instance. Then, we compute the intersection $A^{I_2} := D^{I_1} \cap L_a^I$ and add it to $\boldsymbol{I}_1$ to obtain instance $\boldsymbol{I}_2 = (V^I, \gamma^I, root^I, \{root^I\}, L_a^I, L_b^I, D^{I_1}, A^{I_2})$. This instance is also shown in Figure 5 (b). Next we compute $C^{I_3} := child(A^{I_2})$ which leads to some decompression of instance $\boldsymbol{I}_3$ relative to $\boldsymbol{I}_2$. Finally, we add the selection $C^{I_3} \cap L_b^{I_3}$ to new instance $\boldsymbol{I}_4$, which is the result of our query. $\boldsymbol{I}_4$ is also shown in Figure 5 (c).

Further examples of query evaluation are depicted in Figure 5, (d − i). $\square$

Of course, selections that have been computed as intermediate results and are not needed anymore can be removed from an instance.

In general, an instance resulting from a query is not necessarily in optimal state of compression (even if the input instance was). It is easy to re-compress, but we suspect that this will rarely pay off in practice.
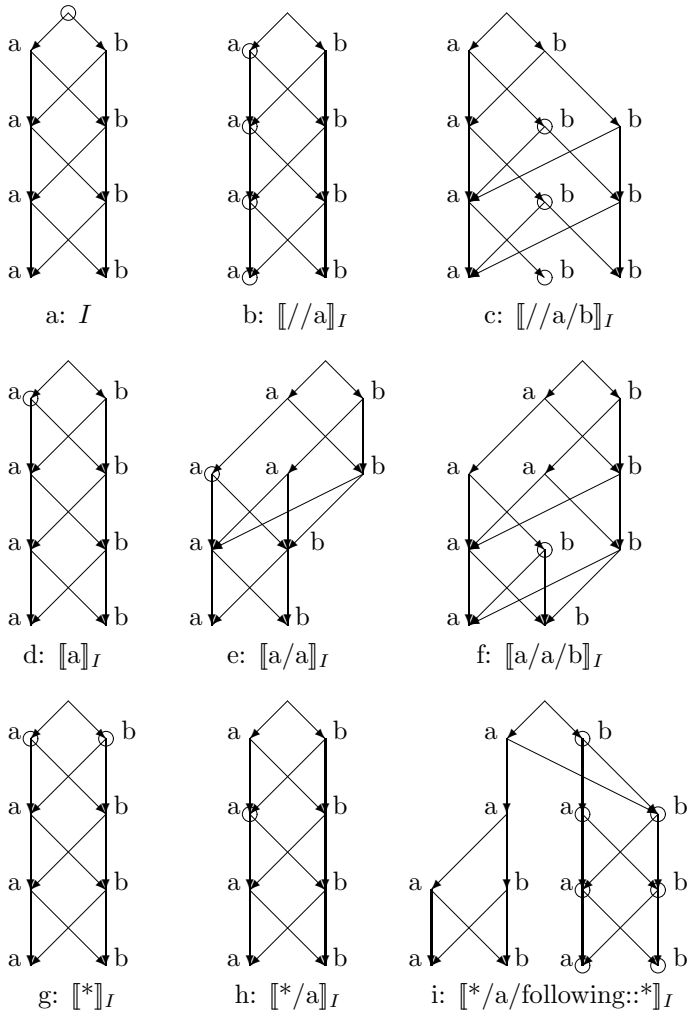
Figure 5: (a) Optimally compressed version of complete binary tree of depth 5, with the root node being selected as context for queries, and eight XPath queries on it (b–i). Nodes selected by the queries are highlighted by circles.

## 3.4 Complexity and Decompression

In theory, our method of compression by sharing subtrees can lead to an exponential reduction in instance size, and even to doubly exponential compression using edge multiplicities. Unfortunately, compressed trees may decompress exponentially in the worst case even on very simple queries (although of course not beyond the size of the completely decompressed instance) [6].

This is a folklore phenomenon in the similar setting of symbolic model checking with OBDDs [4] and a consequence of the great potential for compression offered by bisimulation on trees (respectively, OBDDs).

It is unlikely that better algorithms can improve on the worst-case exponential degree of de-compression [7].

---

[6] In real-life XML documents, we expect to see neither such extreme compression nor decompression through queries.

[7] It is known that exponential degrees of de-compression can always occur unless P = PSPACE. To be precise, while all Core

However, surprisingly, the decompression is only exponential in the size of the *queries* (but not the data), which tend to be small. Thus, query evaluation is *fixed-parameter tractable* in the following sense:

**Theorem 3.6** *Let $Q$ be a Core XPath query and $I$ a compressed instance. Then, $Q$ can be evaluated on $I$ in time $O(2^{|Q|} * |I|)$.*

*Proof.* As can be easily verified, each of the operations of our algebra at most doubles the number of edges and vertices in the instance (cf. Proposition 3.3). Our result follows. □

It must be emphasized that the exponential factor for query evaluation times in Theorem 3.6 strictly depends on decompression; if no decompression occurs (as is the case when our algorithms are applied to an uncompressed tree-instance $I$), our techniques only take time $O(|Q| * |I|)$. Moreover, our $O(2^{|Q|} * |I|)$ algorithm never takes more than $O(|Q| * |T(I)|)$ time.

As a corollary of Proposition 3.3, we have

**Corollary 3.7** *Let $Q$ be a query in our Core XPath algebra where only upward axes are used, and let $I$ be a compressed instance. Then, $Q$ can be computed in time $O(|Q| * |I|)$.*

*Tree pattern queries* (i.e., which are basically boolean queries "selecting" the root node if successful; see e.g. [7]) can be easily (in linear time in the size of the query) transformed into the Core XPath algebra and will only use upward axes. Thus, tree pattern queries inherit this most favorable result.

## 4 Implementation

The main goals of our implementation and experiments were to demonstrate that practical XML data compress well using our techniques and that, as a consequence, even large document structures can be put into main memory. There, queries – even on compressed instances – can be evaluated extremely efficiently. We exploit the fact that at the time of writing this, compiled code running on a commodity computer can effect, say, a depth-first traversal[8] of a tree data structure of several millions of nodes in main memory in a fraction of a second. In our implementation, DAG instances were represented by a straightforward tree data structure in main memory, allowing several child pointers to point to one and the same shared node (without introducing cycles) to be able to represent DAGs.

In our model, an instance $I$ over schema $\sigma$ contains a set of nodes $S^I$ for each $S \in \sigma$. These sets may represent XML tags, but also derived properties of nodes

---

XPath queries on XML trees $T$ can be evaluated in time $O(|Q| * |T|)$, the same Core XPath evaluation problem on compressed trees is PSPACE-complete [12].

[8] All of the Core XPath operations discussed – except for the computation of common extensions – can be carried out with only a single depth-first (left-to-right or right-to-left) traversal of the data structure "modulo decompressions". (Consider Figure 4 to verify this for downward axes.)

such as the matching of a string condition or the membership in a (sub-)query result. Queries in Core XPath build on this idea and consist of expressions created from the names from the schema and the operations introduced in Section 3.

A practical query evaluation algorithm may thus proceed by processing the subexpressions of a query (cf. the query tree in Figure 3) bottom-up, starting with a compressed instance $I$ that holds at least one property $S^I$ (i.e. $I$ is an instance over schema $\sigma$ with $S \in \sigma$) s.t. $S$ appears at a leaf in the query tree. Iteratively, an operation is applied to the instance, which computes a new set of nodes $R$ and possibly partially decompresses $I$, creating an instance $I'$ over schema $\sigma \cup \{R\}$. Whenever a property $P$ is required that is not yet represented in the instance, we can search the (uncompressed) representation of the XML document on disk, distill a compressed instance over schema $\{P\}$, and merge it with the instance that holds our current intermediate result using the common extensions algorithm of Section 2.3.

Our implementation basically follows this mode of evaluation, and all operators discussed in this paper were implemented and their algorithms tested for their practical efficiency and robustness.

Our implementation was written in C++ and includes a new very fast SAX(-like) parser that creates our compressed instances. Given a set of tags and string conditions, our SAX parser builds the compressed instance in one scan of the document and linear time in total. It uses a stack for DAG nodes under construction and a hash table of existing nodes already in the compressed instance that is being created. The hash function on a node $v$ combines the membership of $v$ in the various node sets with the identities (pointer values) of a bounded number of children. String constraints are matched to nodes on the stack on the fly during parsing using automata-based techniques.

As an optimization, in our experiments, we create a compressed instance from the document on disk *once* every time a new query is issued and do not make use of the common extensions algorithm. Given a set of tags $\{L_1, \ldots, L_m\}$ and a set of string conditions $\{L_{m+1}, \ldots, L_n\}$, our parser can create a compressed instance over schema $\{L_1, \ldots, L_n\}$, containing all the relevant information, in one linear scan. Subsequently, we evaluate the query purely in main memory.

Although this is clearly a future goal, our current system does not use a database to store XML data on disk; instead, in the current prototype, we re-parse the XML document every time we have to access it. Currently, we do not make use of index structures to optimize the matching of string data. However, it seems interesting but not difficult to modify the creation of compressed instances to exploit string indexes. [9]

| | $|V^T|$ | $|V^{M(T)}|$ | $|E^{M(T)}|$ | $\frac{|E^{M(T)}|}{|E^T|}$ | |
|---|---|---|---|---|---|
| SwissProt | 10,903,569 | 83,427 | 792,620 | 7.3 % | − |
| (457.4 MB) | | 85,712 | 1,100,648 | 10.1 % | + |
| DBLP | 2,611,932 | 321 | 171,820 | 6.6 % | − |
| (103.6 MB) | | 4481 | 222,755 | 8.5 % | + |
| TreeBank | 2,447,728 | 323,256 | 853,242 | 34.9 % | − |
| (55.8 MB) | | 475,366 | 1,301,690 | 53.2 % | + |
| OMIM | 206,454 | 962 | 11,921 | 5.8 % | − |
| (28.3 MB) | | 975 | 14,416 | 7.0 % | + |
| XMark | 190,488 | 3,642 | 11,837 | 6.2 % | − |
| (9.6 MB) | | 6,692 | 27,438 | 14.4 % | + |
| Shakespeare | 179,691 | 1,121 | 29,006 | 16.1 % | − |
| (7.9 MB) | | 1,534 | 31,910 | 17.8 % | + |
| Baseball | 28,307 | 26 | 76 | 0.3 % | − |
| (671.9 KB) | | 83 | 727 | 2.6 % | + |
| TPC-D | 11,765 | 15 | 161 | 1.4 % | − |
| (287.9 KB) | | 53 | 261 | 2.2 % | + |

Figure 6: Degree of compression of benchmarked corpora (tags ignored: "−"; all tags included: "+").

## 5 Experiments

We carried out a number of experiments to assess the effectiveness of our query evaluation techniques. All experiments were run on a Dell Inspiron 8100 laptop with 256 MB of RAM and a 1 GHz Pentium III processor running Linux. As benchmark data, we chose[10] SwissProt (a protein database), DBLP, Penn TreeBank (a linguistic database containing text from the Wall Street Journal that has been manually annotated with its phrase structure), OMIM (*Online Mendelian Inheritance in Man*, a database of human genes and genetic disorders), XMark (generated auction data), Shakespeare's collected works, and the 1998 Major League Baseball statistics, all in XML and of sizes indicated in the figures.

For each data set, we created five representative queries, $Q_1$ through $Q_5$, which are all listed in Appendix A. In each case, $Q_1$ was a tree pattern query selecting the root node if a given path can be matched in the document. In their algebraic representations, these queries use "parent" as the only axis, thus no decompression is required. Each $Q_2$ was the same query reversed, now selecting the nodes matched by the given path. $Q_3$ also incorporated the descendant axis, conditions, and string constraints. $Q_4$ added branching query trees, and $Q_5$ extended $Q_4$ in that all the remaining axes were allowed. By this choice, we believe to have covered a wide range of practical path queries, and can study the costs of their features individually. All queries were designed to select at least one node.

The experimental results are shown in Figures 6 and 7. They read as follows. In Figure 6, we study the degree of compression obtained using bisimulation. As

---

[9] We intend to extend our system to use indexes, and to combine it with vertical partitioning for string data mentioned in the introduction. This combines efficient search for (string) values in the database with fast navigation of the document structure using our compressed instances in main memory.

[10] Figure 6 also shows the compression on some TPC-D data, which we excluded from the query evaluation experiments because as purely XML-ized relational data, querying it with XPath is not very interesting.

| | | (1) parse time | (2) bef. $|V^{M(T)}|$ | (3) bef. $|E^{M(T)}|$ | (4) query time | (5) after $|V^{Q(M(T))}|$ | (6) after $|E^{Q(M(T))}|$ | (7) #nodes sel. (dag) | (8) #nodes sel. (tree) |
|---|---|---|---|---|---|---|---|---|---|
| SwissProt | $Q_1$ | 56.921s | 84,314 | 796,059 | 1.748s | 84,314 | 796,059 | 1 | 1 |
| (457.4 MB) | $Q_2$ | 56.661s | 84,314 | 796,059 | 1.783s | 84,344 | 796,087 | 1 | 249,978 |
| | $Q_3$ | 64.971s | 84,166 | 798,354 | 1.664s | 84,184 | 798,371 | 106 | 46,679 |
| | $Q_4$ | 79.279s | 84,071 | 808,771 | 2.627s | 84,071 | 808,771 | 1 | 1 |
| | $Q_5$ | 60.036s | 84,480 | 814,307 | 2.825s | 84,999 | 815,281 | 3 | 991 |
| DBLP | $Q_1$ | 8.805s | 1,246 | 176,280 | 0.137s | 1,246 | 176,280 | 1 | 1 |
| (103.6 MB) | $Q_2$ | 8.795s | 1,246 | 176,280 | 0.136s | 1,265 | 176,302 | 1 | 100,313 |
| | $Q_3$ | 10.954s | 2,469 | 187,761 | 0.146s | 2,469 | 187,761 | 18 | 32 |
| | $Q_4$ | 14.056s | 2,191 | 188,368 | 0.313s | 2,196 | 188,368 | 1 | 3 |
| | $Q_5$ | 13.866s | 2,191 | 188,368 | 0.325s | 2,200 | 188,368 | 1 | 3 |
| TreeBank | $Q_1$ | 8.942s | 349,229 | 913,743 | 8.884s | 349,229 | 913,743 | 1 | 1 |
| (55.8 MB) | $Q_2$ | 8.961s | 349,229 | 913,743 | 9.048s | 362,662 | 945,576 | 740 | 1,778 |
| | $Q_3$ | 9.647s | 357,254 | 938,785 | 4.659s | 361,222 | 948,205 | 202 | 203 |
| | $Q_4$ | 11.370s | 348,582 | 912,549 | 4.234s | 348,582 | 912,549 | 9 | 9 |
| | $Q_5$ | 7.883s | 350,671 | 917,197 | 9.910s | 364,141 | 948,170 | 249 | 624 |
| OMIM | $Q_1$ | 1.363s | 963 | 13,819 | 0.011s | 963 | 13,819 | 1 | 1 |
| (28.3 MB) | $Q_2$ | 1.380s | 963 | 13,819 | 0.011s | 964 | 13,819 | 1 | 8,650 |
| | $Q_3$ | 1.669s | 977 | 13,893 | 0.008s | 977 | 13,893 | 1 | 26 |
| | $Q_4$ | 2.085s | 1,030 | 14,766 | 0.016s | 1,042 | 14,781 | 1 | 3 |
| | $Q_5$ | 2.098s | 1,023 | 12,243 | 0.017s | 1,024 | 12,243 | 4 | 4 |
| XMark | $Q_1$ | 1.160s | 3,780 | 11,993 | 0.074s | 3,780 | 11,993 | 1 | 1 |
| (9.6 MB) | $Q_2$ | 0.810s | 3,780 | 11,993 | 0.439s | 3,877 | 12,168 | 13 | 39 |
| | $Q_3$ | 0.839s | 3,755 | 13,578 | 0.033s | 3,755 | 13,578 | 661 | 1,083 |
| | $Q_4$ | 0.844s | 3,733 | 14,747 | 0.042s | 3,750 | 14,841 | 38 | 47 |
| | $Q_5$ | 1.053s | 4,101 | 12,639 | 0.061s | 4,410 | 13,171 | 5 | 5 |
| Shakespeare | $Q_1$ | 1.457s | 1,520 | 31,048 | 0.054s | 1,520 | 31,048 | 1 | 1 |
| (7.9 MB) | $Q_2$ | 0.792s | 1,520 | 31,048 | 0.055s | 1,551 | 31,105 | 2 | 106,882 |
| | $Q_3$ | 0.894s | 1,560 | 31,253 | 0.038s | 1,564 | 31,254 | 2 | 851 |
| | $Q_4$ | 1.050s | 1,586 | 31,364 | 0.046s | 1,586 | 31,364 | 57 | 235 |
| | $Q_5$ | 0.958s | 1,194 | 29,418 | 0.045s | 1,235 | 29,497 | 14 | 67 |
| Baseball | $Q_1$ | 0.082s | 26 | 76 | 0.001s | 26 | 76 | 1 | 1 |
| (671.9 KB) | $Q_2$ | 0.082s | 26 | 76 | 0.001s | 30 | 76 | 1 | 1,226 |
| | $Q_3$ | 0.083s | 46 | 805 | 0.001s | 46 | 805 | 1 | 276 |
| | $Q_4$ | 0.116s | 1,215 | 14,413 | 0.023s | 1,226 | 14,413 | 47 | 47 |
| | $Q_5$ | 0.090s | 48 | 870 | 0.003s | 53 | 892 | 1 | 58 |

Figure 7: Parsing and query evaluation performance.

pointed out before, compression depends on which labeling information the trees to be compressed have to carry, so we chose two settings for this initial experiment (represented by the two rows in the table for each corpus). In the upper rows (marked $-$), we show the degree of compression for the simplest case, that in which the schema is empty (tags have been erased) and the bare tree structure is compressed. We believe that this is a valuable indicator of the intrinsic complexity of the structure of a document. In the lower rows (those marked $+$), we include all tags of the document into the compressed instance ($+$), but no other labelings.

The degree of compression is measured as the ratio $|E^{M(T)}|/|E^T|$ of the number of edges $|E^{M(T)}|$ in the compressed instance $M(T)$ to the number of edges $|E^T|$ in the tree skeleton $T$ (as edges dominate the vertices in the compressed instances). Of course, $|E^T| = |V^T| - 1$.

Figure 7 reports on our experiments with queries.

As discussed in Section 4, the evaluation of each query consists of two parts, first the extraction of the relevant information from the document into a compressed instance (The "parse time" in column (1) of Figure 7 thus includes the time taken for compression.) and second the actual query evaluation in main memory. The sizes of these compressed instances $M(T)$ (initially, before query evaluation) are shown in columns (2) and (3) of Figure 7. The query evaluation times are presented in column (4), and the sizes of instances after query evaluation (indicating how much decompression occurred during query evaluation) are shown in columns (5) and (6). We counted how many nodes in the compressed instance were selected, as shown in column (7). Finally, we also calculated to how many nodes in the uncompressed tree-version of the result these nodes corresponded (column (8)). The depth-first traversal required to compute the latter is the same as the one required to "decode" the query result in order to compute "translate" or "apply" it to the

uncompressed tree-version of the instance.

Columns (2) and (3) of Figure 7 add to our discussion of the degree of compression obtained. Figure 6 reported on the case where either just the bare tree structure $(-)$ or also all of the node tags in the document $(+)$ were included in the compressed instance. Here, the information included into the compressed instance was one node set for each of the tags and one for each of the string constraints appearing in the queries; all other tags were omitted. (Thus, whenever a query does not contain string constraints, the initial degree of compression obtained is certain to be between the two numbers given for each data set in Figure 6.)

The experiments suggest that, given a new set of data, we can expect compression to about one-tenth to one-fifteenth of the original size of the skeleton. For highly structured documents, and particularly databases that are (close to) XML-ized relational data, we can expect a substantially better degree of compression.

The notable outlier is Penn TreeBank (in which trees often are very deep), which we suspect does not compress substantially better than randomly generated trees of similar shape. This is further evidence that linguistic data sets are among the most complicated around, and deserve further study.

Regarding query performance, our results are extremely competitive. Indeed, we believe that the fact that compression leads to reduced amounts of data to be processed during query evaluation adds another strong point to our approach, besides reduced main memory consumption. It is worth mentioning that while memory consumption is dominated by instance sizes in terms of edges, query evaluation times are dominated by node counts. This is not a surprise, as our algorithms are very much centered around nodes and their selections. While the compressed instances of SwissProt and TreeBank are of similar size, Tree-Bank's query evaluation times are considerably higher, as is the node count (see Figure 7).

## 6 Discussion and Conclusions

In this paper, we have presented a novel approach to querying XML by keeping compressed representations of the tree structure of documents in main memory. As we have argued, our approach has a strong motivation from symbolic model checking. Beyond algorithms, we are able to borrow some of its theory and elegant framework.

Most interesting, though, is that this approach supports very efficient XML query processing, a claim for which we have provided experimental evidence. Three reasons deserve to be given.

- Our notion of compression is based upon bisimulation, allowing for natural evaluation techniques with virtually no overhead compared with traditional main-memory techniques. Thus, our algorithms are competitive even when applied to uncompressed data.

- The separation of skeleton tree structure from string data and the subsequent compression ensure

that very large parts of XML data (w.r.t. query evaluation efficiency) – those that queries have to access globally and navigate in – fit into main memory. Minimizing the degree of fragmentation ("shredding") of such data is essential.

- Our compression technique uses sharing of common substructures. Thus, even for moderately-sized documents that traditional main-memory engines can process without difficulty, we may be more efficient because such engines have to repetitively re-compute the same results on subtrees that are shared in our compressed instances.

We have observed that for moderately regular documents, the growth of the size of compressed instances as a function of document sizes slows down when documents get very large, and we may indeed be able to deal with extremely large instances of this kind in main memory alone. However, in general, we want to be able to apply some shredding and cache chunks of compressed instances in secondary storage to be truly scalable. Of course these chunks should be as large as they can be to fit into main memory. Although it seems not to be difficult, this is future work.

In this paper, we have focussed on path queries. For the future, we plan to extend our work to evaluating XQuery on compressed instances.

## Acknowledgements

## References

[1] S. Abiteboul. "Querying Semistructured Data". In *Proceedings of the 6th International Conference on Database Theory (ICDT)*, Delphi, Greece, 1997.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. "Weaving Relations for Cache Performance". In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, 2001.

[3] D. S. Batory. "On Searching Transposed Files". *ACM Transactions on Database Systems*, **4**(4):531–544, 1979.

[4] R. E. Bryant. "Graph-Based Algorithms for Boolean Function Manipulation". *IEEE Transactions on Computers*, **C-35**(8):677–691, Aug. 1986.

[5] P. Buneman, S. B. Davidson, M. F. Fernandez, and D. Suciu. "Adding Structure to Unstructured Data". In *Proceedings of the 6th International Conference on Database Theory (ICDT)*, pages 336–350, Delphi, Greece, 1997.

[6] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. "Symbolic Model Checking: $10^{20}$ States and Beyond". *Information and Computation*, **98**(2):142–170, 1992.

[7] C. Y. Chan, W. Fan, P. Felber, M. N. Garofalakis, and R. Rastogi. "Tree Pattern Aggregation for Scalable XML Data Dissemination". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, 2002.

[8] A. Deutsch, M. Fernandez, and D. Suciu. "Storing Semistructured Data with STORED". In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1999.

[9] P. Fankhauser, G. Huck, and I. Macherius. "Components for Data Intensive XML Applications". See also: http://www.ercim.org/publication/Ercim_News/enw41/fankhauser.html.

[10] M. F. Fernandez and D. Suciu. "Optimizing Regular Path Expressions Using Graph Schemas". In *Proceedings of the 14th IEEE International Conference on Data Engineering (ICDE)*, pages 14–23, Orlando, Florida, USA, Feb. 1998.

[11] D. Florescu and D. Kossmann. "Storing and Querying XML Data using an RDMBS". *IEEE Data Engineering Bulletin*, **22**(3):27–34, 1999.

[12] M. Frick, M. Grohe, and C. Koch. "Query Evaluation on Compressed Trees". In *Proceedings of the 18th Annual IEEE Symposium on Logic in Computer Science (LICS)*, Ottawa, Canada, 2003.

[13] R. Goldman and J. Widom. "DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases". In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, pages 436–445. Morgan Kaufmann, 1997.

[14] G. Gottlob, C. Koch, and R. Pichler. "Efficient Algorithms for Processing XPath Queries". In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, 2002.

[15] H. Liefke and D. Suciu. "XMill: An Efficient Compressor for XML Data". In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 2000.

[16] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[17] T. Milo and D. Suciu. "Index Structures for Path Expressions". In *Proceedings of the 7th International Conference on Database Theory (ICDT)*, 1999.

[18] M. Neumüller and J. N. Wilson. "Improving XML Processing Using Adapted Data Structures". In *Proc. Web, Web-Services, and Database Systems Workshop*, pages 206–220. Springer LNCS 2593, 2002.

[19] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. "Relational Databases for Querying XML Documents: Limitations and Opportunities". In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB)*, pages 302–314, 1999.

[20] P. Tolani and J. R. Haritsa. "XGRIND: A Query-friendly XML Compressor". In *Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE)*, 2002.

[21] World Wide Web Consortium. XML Path Language (XPath) Recommendation. http://www.w3c.org/TR/xpath/, Nov. 1999.

[22] J. Ziv and A. Lempel. "A Universal Algorithm for Sequential Data Compression". *IEEE Transactions of Information Theory*, **23**(3):337–349, May 1977.

## A   Benchmark Queries

This appendix lists the queries used in the experiments of Section 5. We write string constraints as "abc", meaning that a node matches the constraint if the string "abc" is *contained* in its string value.

**SwissProt**
```
Q1: /self::*[ROOT/Record/comment/topic]
Q2: /ROOT/Record/comment/topic
Q3: //Record/protein[taxo["Eukaryota"]]
Q4: //Record[sequence/seq["MMSARGDFLN"] and
            protein/from["Rattus norvegicus"]]
Q5: //Record/comment[topic["TISSUE SPECIFICITY"]
      and following-sibling::comment/topic[
          "DEVELOPMENTAL STAGE"]]
```

**DBLP**
```
Q1: /self::*[dblp/article/url]
Q2: /dblp/article/url
Q3: //article[author["Codd"]]
Q4: /dblp/article[author["Chandra"] and
                author["Harel"]]/title
Q5: /dblp/article[author["Chandra" and
      following-sibling::author["Harel"]]]/title
```

**Penn TreeBank**
```
Q1: /self::*[alltreebank/FILE/EMPTY/S/VP/S/VP/NP]
Q2: /alltreebank/FILE/EMPTY/S/VP/S/VP/NP
Q3: //S//S[descendant::NNS["children"]]
Q4: //VP["granting" and descendant::NP["access"]]
Q5: //VP/NP/VP/NP[following::NP/VP/NP/PP]
```

**OMIM**
```
Q1: /self::*[ROOT/Record/Title]
Q2: /ROOT/Record/Title
Q3: //Title["LETHAL"]
Q4: //Record[Text["consanguineous parents"]
          ]/Title["LETHAL"]
Q5: //Record[Clinical_Synop/Part["Metabolic"
      ]/following-sibling::Synop[
          "Lactic acidosis"]]
```

**XMark**
```
Q1: /self::*[site/regions/africa/item/
      description/parlist/listitem/text]
Q2: /site/regions/africa/item/
      description/parlist/listitem/text
Q3: //item[payment["Creditcard"]]
Q4: //item[location["United States"] and
      parent::africa]
Q5: //item/description/parlist/listitem[
      "cassio" and
      following-sibling::*["portia"]]
```

**Shakespeare's Collected Works**
```
Q1: /self::*[all/PLAY/ACT/SCENE/SPEECH/LINE]
Q2: /all/PLAY/ACT/SCENE/SPEECH/LINE
Q3: //SPEECH[SPEAKER["MARK ANTONY"]]/LINE
Q4: //SPEECH[SPEAKER["CLEOPATRA"] or
          LINE["Cleopatra"]]
Q5: //SPEECH[SPEAKER["CLEOPATRA"] and
      preceding-sibling::SPEECH[
          SPEAKER["MARK ANTONY"]]]
```

**1998Baseball**
```
Q1: /self::*[SEASON/LEAGUE/DIVISION/TEAM/PLAYER]
Q2: /SEASON/LEAGUE/DIVISION/TEAM/PLAYER
Q3: //PLAYER[THROWS["Right"]]
Q4: //PLAYER[ancestor::TEAM[TEAM_CITY["Atlanta"]]
        or (HOME_RUNS["5"] and STEALS["1"])]
Q5: //PLAYER[POSITION["First Base"] and
          following-sibling::PLAYER[
              POSITION["Starting Pitcher"]]]
```