Inflatable XML Processing

Rohit Fernandes[1] (rohitf@cs.cornell.edu)
Mukund Raghavachari[2] (raghavac@us.ibm.com)

[1] Department of Computer Science, Cornell University
[2] IBM T.J. Watson Research Center

Abstract. The past few years have seen the widespread adoption of XML as a data representation format in various middleware: databases, Web Services, messaging systems, etc. One drawback of XML has been the high cost of XML processing. We present in this paper InflateX, a system that supports efficient XML processing. InflateX advances the state of the art in two ways. First, it uses a novel representation of XML, called *inflatable trees*, that supports lazy construction of an XML document in-memory in response to client requests, as well as, more efficient serialization of results. Second, it incorporates a novel algorithm, based on the idea of *projection* [8], for efficiently constructing an inflatable tree given a set of XPath expressions. The projection algorithm presented in this paper, unlike previous work, can handle all axes in XPath, including complex axes such as `ancestor`. While we describe the algorithm in terms of our inflatable tree representation, it is portable to other representations of XML. We provide experiments that validate the utility of our inflatable tree representation and our projection algorithm.

Keywords: XML, XPath, Performance, Projection

1 Introduction

The past few years have seen the widespread adoption of XML as a data interchange format in various middleware: databases, Web Services, messaging systems, etc. The popularity of XML has been accompanied by its main drawback — the high cost of XML processing. One of the factors affecting XML processing is the memory footprint of XML documents — when documents are large or many documents are processed simultaneously, XML processors may operate inefficiently or not execute at all.

Consider the following (common) situation — a web service receives an XML document over the network. In processing the document, the web service accesses certain portions of the document (possibly by executing queries in a language such as XQuery [14] or XPath [12] on the document). Based on the result of processing, the web service constructs a new XML document and publishes it over the network. In such a situation, the cost of loading an instance of the XML document into main memory and serializing the constructed output can dwarf the cost of query evaluation during the execution of the web service.

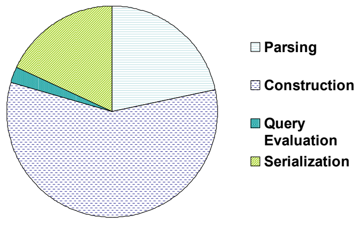

Fig. 1. Breakdown of query processing time in terms of parsing time, construction time, query evaluation time, and result serialization time.

Figure 1 presents a breakdown of the cost of executing a query on a DOM [13] representation of an XML document.[3]

In this paper, we describe a system, InflateX, that addresses the high cost of XML processing. At the heart of the InflateX system is a novel representation of XML, called *inflatable tree*, that builds portions of an XML document lazily in memory in response to traversals of the document initiated by clients. The remaining portion of the XML document is stored in binary form, which can be up to five times more concise than the DOM representation of XML [8]. To a client, InflateX provides a DOM view of the XML document — the client may manipulate this view as one would any DOM representation. We will show that the inflatable tree representation is more efficient (in general) than full DOM materialization of a document in all aspects of XML processing : construction of an instance of a document in memory, query evaluation, and serialization of output.

To optimize the lazy construction of inflatable trees, InflateX allows clients to specify a set of XPath expressions with respect to which the document should be *projected* [8]. In one pass over the document, the InflateX system materializes those portions of the document that are relevant to the provided set of XPath expressions and retains the remaining portions in binary form. Traversals of the inflatable tree that are contained in the set of XPath expressions can be processed efficiently (since those nodes are already materialized in memory). Traversals that access portions that are not materialized will cause the InflateX system to materialize those portions on-demand. We will provide a novel projection algorithm that can handle all XPath axes — previous work could handle only XPath expressions with `child` and `descendant` axes.

1.1 Contributions

The contributions of the paper are the following:

[3] The figure reports the execution of the Java equivalent of the XQuery `for $i in /site/regions/namerica/item return $i` on a 10MB XMark [11] document.

- A novel representation of XML, called *inflatable tree*, that supports lazy construction of an XML document in memory. The representation allows for more efficient construction, query evaluation and serialization of XML data.
- A novel projection algorithm that can handle all XPath axes. We will show that the definition of projection of Marian and Siméon is not sufficient when axes other than `child` and `descendant` are used, and provide a general definition of projection that is valid for all XPath axes.
- Experiments that demonstrate that the *inflatable tree* representation substantially reduces the construction and serialization time in XML processing. Furthermore, the inflatable tree representation allows an XML processor to handle larger documents than it might otherwise (approximately, 2-5 times the corresponding DOM representation).

1.2 Related Work

Bohannon *et al.* [4] describe a virtual DOM interface that delivers navigable XML views of relational data. Like inflatable trees, their interface supports lazy materialization of an XML document. Their system, however, relies on the existence of an underlying database that acts as a persistent store for the XML data. The system also relies on the database for query execution. In many situations, for example, for some web services, such a store may not be available. Our inflatable tree representation provides a mechanism for efficient XML processing in memory, without any requirements of an underlying database.

Marian and Siméon have introduced the idea of *projection* which constructs a DOM representation of a document based on a set of XPath expressions [8]. One drawback to projection as defined by Marian and Siméon is that it assumes that all queries that will be executed on the document are known in advance. The inflatable tree representation is robust in that it can be used even when the full set of XPath expressions that will be evaluated on the document is not known in advance. Second, their projection algorithm cannot handle XPath expressions involving axes such as `parent` and `ancestor`. Finally, their approach does not reduce the cost of serialization of results which, as observed in Figure 1, can be high.

Compressed XML [5] is a concise representation of an XML document. The tree skeleton of an XML document — the portion of an XML document obtained by ignoring all string information — is compressed. String information is not stored directly, but if the queries are known in advance, compressed XML encodes information about the strings that may be required to evaluate the queries on the document. Unlike compressed XML, our representation retains all information relevant to an XML document.

Streaming algorithms [3, 6, 7] reduce the memory overhead of XML processing by not constructing the document in memory, but processing it as it is parsed. They can be applied in constrained circumstances where all queries evaluated in the document are known in advance and are independent of each other. As with projection, streaming algorithms support only limited subsets of query languages; for complex queries involving joins or nested queries, it is necessary to manifest portions of the document in memory [8].

1.3 Structure of Paper

The paper is structured as follows. In Section 2, we describe our system architecture and the *inflatable* tree representation. In Section 3, we present a new definition of projection that is valid when all XPath axes are allowed. In Section 4 we present our algorithm for document projection. In Section 5, we give an overview of our implementation. In Section 6, we provide experimental results. Finally, in Section 7, we conclude and describe future work.

2 System Architecture

The architecture of our system is depicted in Figure 2. A client passes a reference to a data stream, and optionally, a set of XPath expressions called the *projection set* to the InflateX system. The projection set is an approximation of the traversals that will be executed over the XML document; it is used as a hint to optimize the construction of the inflatable tree representation of the document. The projection set need not be complete — the client may execute XPath expressions over the document that are not covered by the projection set. The InflateX system uses the projection set and the XML data stream to construct an initial inflatable tree representation of the XML document. The client may determine the initial projection set using various mechanisms, for example, static analysis of the client application, profiling information of the most common XPath expressions or traversals used, etc. In this paper, we will focus on mechanisms for building the inflatable tree efficiently given a projection set.

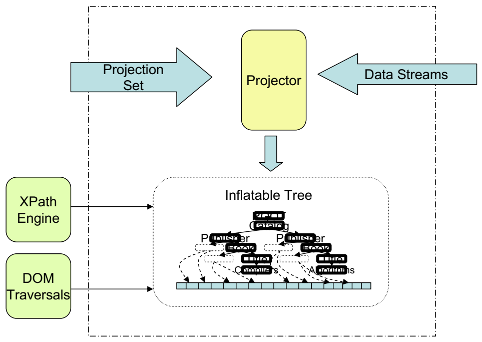

Fig. 2. System architecture.

We now describe our inflatable tree representation and how a client interacts with it in greater detail. For simplicity, we will focus on elements, though our implementation can handle the other XML nodes, such as attribute nodes.

2.1 Inflatable Trees

Our representation of XML documents, *inflatable tree*, is based on the observation that the binary representation of an XML document (as a sequence of bytes) can be 4-5 times more concise than constructing a DOM model instance of the document. Given a reference to an XML document, we store the sequence of bytes corresponding to the XML document in an array of bytes in memory. Our representation of the XML document in memory consists of two sorts of nodes: *materialized* nodes and *inflatable* nodes. A *materialized* node corresponds to an element in the document and contains all information relevant to the element, such as its tag. An *inflatable* node represents an unexpanded portion of the XML document; it contains a pair of offsets into the byte array representation of the document corresponding to the start and end of the unexpanded portion. For example, Figure 3a depicts the inflatable tree representation of an XML document tree. The materialized nodes are shown with a label, and the nodes that have a dashed border are inflatable nodes. They contain offsets into the binary array of bytes.

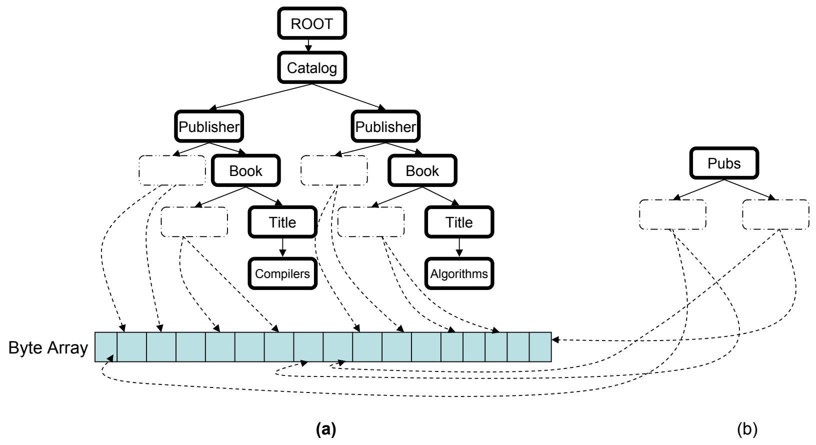

Fig. 3. (a) Inflatable tree epresentation of an XML document. Boxes with solid borders represent materialized nodes. Boxes with dashed borders represent inflatable nodes. (b) Representation of a constructed XML document.

2.2 Operations on Inflatable Trees

We now describe how a client may operate on an inflatable tree.

Inflatable Tree Refinement Once an inflatable tree is constructed, a client may operate on the tree as with any other DOM representation of an XML document, for example, by executing an XPath expression with respect to a node of the inflatable tree. If the client accesses a portion of the tree that has not yet been materialized, the runtime system inflates that portion of the tree automatically in response to the client's request. If desired, the client may pass a new projection set to the InflateX system, which will be used by the system to inflate portions of the tree corresponding to the new provided set of XPaths.

Construction of XML A client may construct new nodes and trees, which are always constructed in materialized form. When construction refers to subtrees from existing documents, InflateX constructs an inflatable node with the appropriate offsets. For example, Figure 3b shows the result of constructing a tree based on the input XML document of Figure 3a. The children on the `Pubs` element in Figure 3b are the two `Publisher` subtrees in Figure 3a.

Serialization of Results Since the byte array representation of the input XML documents is retained in memory, portions of the results that are derived from the input document can be serialized directly from the byte array. As we will show in Section 6, this direct serialization can be substantially more efficient than explicit traversal of a tree to perform serialization. For example, in Figure 3b, the inflatable nodes corresponding to the `Publisher` elements can be serialized directly from the input document byte array.

3 Preliminaries

We define the abstractions of XML documents and XPath expressions that will be used in this paper. We will then provide a definition of projection that is valid when all XPath axes are supported.

3.1 Tree Model of XML Documents

An XML document can be represented as a tree whose nodes represent the structural components of the document — elements, text, attributes, etc. Parent-child edges in the tree represent the inclusion of the child component in its parent element, where the scope of an element is bounded by its start and end tags. The tree corresponding to an XML document is rooted at a virtual element, ROOT, which contains the document element. We will discuss XML documents in terms of their tree representation; D represents an XML document, and N_D and E_D denote its nodes and edges respectively.

For simplicity of exposition, we focus on elements in this paper, and ignore attributes, text nodes, etc. The tree, therefore, consists of the virtual root and the elements of the document. We refer to the nodes of the document tree as *elements* to avoid confusion with vertices of the tree representation of an XPath which we will discuss shortly. We assume that the following functions are defined on the elements of an XML document:

- $\text{ID}_D : N_D \rightarrow \mathit{Integer}$: Returns a unique identifier for each element in a document. We will assume that ID_D is a total order on the elements in D, such that the assignment of identifiers to elements corresponds to a depth-first preorder traversal of the tree (that is, *document order* in XML).

– $\textsc{tag}_D : N_D \to \mathit{String}$: Returns the tag name of the element.

We also assume functions, $\textsc{child}_D$, $\textsc{desc}_D$, $\textsc{self}_D$, $\textsc{fs}_D$, and $\textsc{following}_D$, each with the signature $N_D \times N_D \to \{\mathit{true}, \mathit{false}\}$. The semantics of these functions is straightforward, $\textsc{child}_D(v_1, v_2)$ returns *true* if v_2 is a child of v_1 in D, and $\textsc{fs}_D(v_1, v_2)$ returns *true* if v_1 and v_2 share a common parent, and moreover, $\textsc{id}_D(v_2) > \textsc{id}_D(v_1)$. $\textsc{following}_D(v_1, v2)$ returns true if $\textsc{id}_D(v_2) > \textsc{id}_D(v_1)$ and v_2 is not a descendant of v_1. Finally, $\textsc{self}_D(v_1, v_2)$ returns true if $v_1 = v_2$.

3.2 XPath Subset

The grammar of XPath expressions accepted by our projection algorithm is provided below. In the grammar, the non-terminal *Axis* includes all axes defined in the XPath specification [12]. For simplicity, we will only consider elements and not consider the `namespace` and `attribute` axes.

$$\begin{array}{ll}
\mathit{AbsLocPath} & := \text{'/'}\ \mathit{RelLocPath} \\
\mathit{RelLocPath} & := \mathit{Step}\ \text{'/'}\ \mathit{RelLocPath} \ \mid\ \mathit{Step} \\
\mathit{Step} & := \mathit{Axis} :: \mathit{NodeTest}\ \mid \mathit{Step}\ \text{'['}\ \mathit{PredExpr}\ \text{']'} \\
\mathit{PredExpr} & := \mathit{RelLocPath\ and\ PredExpr}\ \mid \mathit{AbsLocPath\ and\ PredExpr}\ \mid \\
 & \quad\ \mathit{RelLocPath}\ \mid\ \mathit{AbsLocPath} \\
\mathit{NodeTest} & := \mathit{String} \mid *
\end{array}$$

An *absolute* path expression corresponds to one that satisfies *AbsLocPath* and is evaluated with respect to the root node of the tree. A relative XPath expression corresponds to *RelLocPath* and is evaluated with respect to a provided set of elements in the tree.

3.3 XPath Expression Trees

An XPath expression can be represented as a rooted tree $T = (V_T, E_T)$ with labeled vertices and edges. The root of the tree is labeled ROOT. For every *NodeTest* in the expression, there is a vertex labeled with the *NodeTest*. Each vertex other than ROOT has a unique incoming edge labeled with the *Axis* specified before the *NodeTest*. The vertex corresponding to the rightmost *NodeTest* which is not contained in a *PredExpr* is designated to be the output vertex. There are functions, $\textsc{label}_T : V_T \to \mathit{String}$, and $\textsc{axis}_T : E_T \to \mathit{Axis}$ that return the labels associated with the vertices and edges respectively. Figure 4 provides an example of the tree representation of the XPath expression `//book[title and author]/ancestor::publisher`.[4]

The semantics of an absolute XPath expression is defined in terms of *embeddings* [9].

Definition 1. *A pair of elements* (n_1, n_2) *in a document,* D, $n_1, n_2 \in N_D$ satisfies *an edge constraint* (v_1, v_2) *in the tree representation* T *of an XPath expression if the relation between* n_1 *and* n_2 *in the document matches* $\textsc{axis}_T(v_1, v_2)$. *For*

[4] We will use the abbreviated XPath syntax in the paper for conciseness.

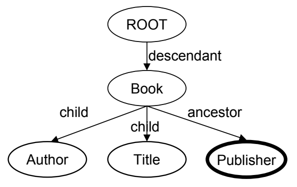

Fig. 4. Tree representation of the XPath expression `//Book[Title and Author]/ancestor::Publisher`. The output vertex has a thick border.

example, n_1, n_2 satisfies (v_1, v_2) if $\textsc{axis}_T(v_1, v_2) =$ ***child*** *and* $\textsc{child}_D(n_1, n_2) =$ *true, or, if* $\textsc{axis}_T(v_1, v_2) =$ ***ancestor*** *and* $\textsc{desc}_D(n_2, n_1) =$ *true.*

Definition 2. *An embedding of an absolute XPath expression T into a document D is a function $\mathcal{E} : V_T \rightarrow N_D$ such that:*

1. *$\mathcal{E}$ maps the* ROOT *vertex of the XPath expression to the* ROOT *element of the document.*
2. *Labels are* matched*, that is, for each $v \in V_T$,* $\textsc{label}_T(v) = *$ *or* $\textsc{label}_T(v) = \textsc{tag}_D(\mathcal{E}(v))$.
3. *Edges are* satisfied*, that is, if $(v_1, v_2) \in E_T$, then $(\mathcal{E}(v_1), \mathcal{E}(v_2))$ satisfies (v_1, v_2).*

Let o be the output vertex of the tree representation of an absolute XPath expression. The output of an XPath expression is defined as all $n \in N_D$ such that there exists an embedding where $\mathcal{E}(o) = n$. The definition can be extended easily to relative XPaths by replacing the embedding of the ROOT element with the context node.

For example, an embedding of the XPath expression tree of Figure 4 into the XML document from Figure 5 is the following : $\mathcal{E}(ROOT) = \{1\}, \mathcal{E}(Book) = \{5\}$, $\mathcal{E}(Author) = \{6\}$, $\mathcal{E}(Title) = \{8\}$ and $\mathcal{E}(Publisher) = \{3\}$.

3.4 Projection

A projected document is defined by Marian and Siméon in terms of an input document D and a set of XPath expressions P, where some of the expressions may be marked with the special output flag # [8]. Each XPath expression in P is an absolute XPath expression (that is, it is evaluated with respect to the root of the document). Only uses of the `child` and `descendant` axes are allowed (predicates and backward axes are not allowed). Given P and D, the projected document D' is defined as follows: The projected document contains all elements that are in the result set of an XPath expression in P, as well as, their ancestors. All subtrees rooted at some result of an XPath expression marked # are materialized as well. The definition guarantees that the projected document D' satisfies the key property that the evaluation of any XPath expression in P on D' returns the same result as the evaluation of that XPath expression on D. As a result, one can substitute D' for D without changing the behavior of query evaluation with respect to P.

For example, consider the XPath expression, `//Title`, and assume that it is marked with a #. Figure 5 depicts the elements that would be constructed in the projection of the document with respect to this XPath expression.

When XPath expressions with axes other than `child` and `descendant` are allowed in P, projection as defined in [8] can no longer be applied; the evaluation of an XPath expression on the projected document D' may differ from that on D. Consider the XPath expression, `//Author/ancestor::Publisher//Title` executed on the document in Figure 5. Only the elements highlighted in Figure 5 belong to the projected document D'. The result of the XPath expression on D' will be the empty set since it does not contain any `Author` elements.

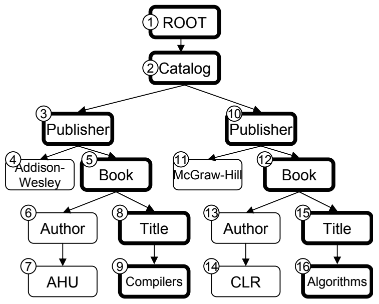

Fig. 5. Tree representation of an XML document. Highlighted nodes depict nodes selected by the algorithm of Marian and Siméon.

The embeddings of XPath expressions into a document D can be used as the basis for a general definition of projection when complex axes such as `ancestor` are allowed. The definition we provide subsumes that of [8] and serves as the basis for the algorithm presented in Section 4.

Definition 3. *Let D be a document and P be a set of absolute XPath expressions, where some XPath expressions in P are marked with a special flag #. The projected document D' is composed of the set of all elements n in D that satisfy at least one of the following conditions:*

- *For some XPath expression p in P, there is an embedding $\mathcal{E}$ of p into D such that $\mathcal{E}(v) = n$, where v is some vertex in p, or*
- *For some XPath expression p in P, there is an embedding $\mathcal{E}$ of p into D such that $\mathcal{E}(v) = n'$, where v is some vertex in p, and n is an ancestor of n' in D, or*
- *For some XPath expression p in P marked with the symbol #, n is the descendant of an element in the result set of the evaluation of p on D.*

In other words, the projected document consists of all elements that participate in an embedding and their ancestors. Moreover, for each element in the result set of the evaluation of a specially marked XPath expression, that element and all its descendants belong to the projected document.

4 Inflatable Tree Construction

In this section, we present an algorithm for constructing an inflatable tree from a given set of XPath expressions while parsing the document. The challenge is in being able to handle complex XPath axes such as `ancestor` efficiently in a single pass over the input document. Our algorithm may be imprecise in that it may materialize some elements that do not satisfy any of the conditions of Definition 3. The algorithm is, however, careful in limiting the construction of these inessential nodes.

Our algorithm works in two stages. First, the set of input XPath expressions P is normalized into a canonical form. In the second stage, a document (or a subtree of the document) is traversed to build the inflatable tree. Our algorithm will not distinguish XPath expressions marked "#" from those that are not. Since the bytes corresponding to the document are readily available, there is no need to inflate the subtrees under output nodes, unless portions of these subtrees may participate in an embedding (that is, satisfy the first two conditions of Definition 3).

4.1 Normalizing XPath Expressions

The XPath axes `following`, `preceding`, `following-sibling` and `preceding-sibling` are order-based axes (the result set for these axes depends on the order between sibling tree nodes). The first step in our normalization is to rewrite instances of these axes in XPath expressions into order-blind axes (such as `parent` and `ancestor`). The rules for rewriting XPath expression trees are shown in Figure 6. In the figure, v_1 and v_2 are vertices in a given XPath expression tree, connected by an edge labeled with one of the order-based axes. The rewriting rules may introduce new vertices. The rules are ordered so that the rules of Figure 6a and Figure 6b are applied until there are no instances of `following` and `preceding` in the XPath expression tree. The rules of Figure 6c and Figure 6d are then applied to the XPath expression tree.

For example, for the `following-sibling` axes, we replace instances of the pattern v_1`/following-sibling::`v_2 with instances of v_1`/parent::`$*/v_2$. The rewritten XPath expression is an approximation of the original one — it chooses v_2 elements that both precede and follow v_1 elements. The rewritings guarantee that for any document, if an element n participates in an embedding of the original XPath expression tree into the document, n also participates in an embedding of the rewritten tree into the document.

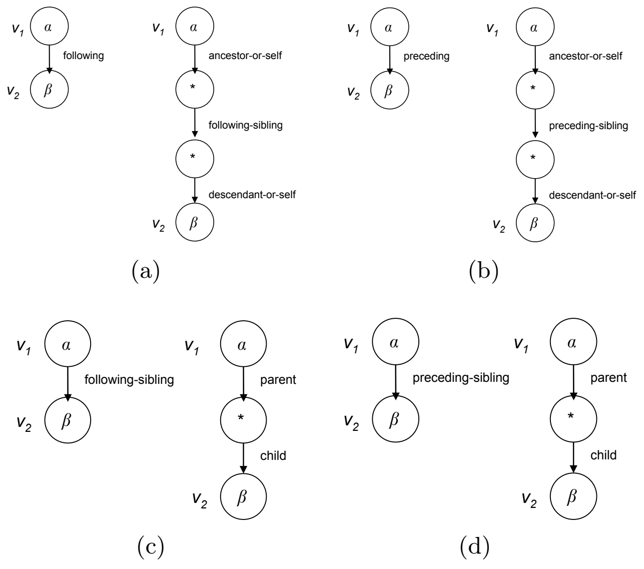

Fig. 6. (a) Rule for rewriting `following` edges. (b) Rule for rewriting `preceding` edges. (c) Rule for rewriting `following-sibling` edges. (d) Rule for rewriting `preceding-sibling` edges.

4.2 Constructing an Inflatable Tree

The inflatable tree construction algorithm can be invoked by the client in one of two states. In the first case, the document is being processed for the first time and must be read from an external source. In the second case, an inflatable tree already exists for the document in question, and the inflatable tree must be modified to account for the new projection set of XPath expressions. In either of the two cases, the algorithm traverses the document in a depth-first manner and generates events similar to SAX [10]. A *start* element event is generated when the traversal first visits an element, and an *end* element event once the traversal of the subtree rooted at that element is finished. We will assume that an event contains all information about the relevant element, such as its tag and unique identifier (we will use the offset in the byte array for this purpose). At each of these events, an event handler is invoked to perform actions related to the construction of the tree.

In the case where a document is read for the first time from an external source, the traversal records the bytes corresponding to the XML document into an array. It simultaneously parses the document and generates appropriate events. In the other case, where an inflatable tree already exists, the document traverser walks over the inflatable tree and generates events. When it reaches an inflatable node, it parses the portion of the byte array corresponding to that node and generates appropriate events.

Definitions and Data Structures The description of our algorithm will use the following definitions.

Definition 4. *The* backward vertex set*, $\mathcal{B}(v)$, of a vertex $v \in V_T$ in an XPath expression tree is defined as* $\{v'|(v,v') \in E_T, \textsc{axis}(v,v') \in \{$***parent, ancestor, ancestor-or-self, self*** $\} \cup \{v''|(v'',v) \in E_T, \textsc{axis}(v'',v) \in \{$ ***self, child, descendant, descendant-or-self*** $\}$*. A* backward constraint *is an edge between v and a vertex in its backward vertex set.*

In other words, the *backward vertex set* with respect to a vertex v consists of those vertices to which an outgoing edge from v is labeled with a backward axis and those from which an incoming vertex into v is labeled with a forward axis. We have a dual definition for a *forward vertex set* with respect to a vertex v.

Definition 5. *The* forward vertex set*, $\mathcal{F}(v)$, of a vertex $v \in V_T$ in an XPath expression tree is defined as* $\{v'|(v,v') \in E_T, \textsc{axis}(v,v') \in \{$***child, descendant, descendant-or-self, self*** $\} \cup \{v''|(v'',v) \in E_T, \textsc{axis}(v'',v) \in \{$ ***self, parent, ancestor, ancestor-or-self*** $\}$*. A* forward constraint *is an edge between v and a vertex in its forward vertex set.*

Our algorithm maintains an *active stack*, which contains, at any time, the list of elements for which a start event has been received, but no end event has been received yet. For each element e in the stack we maintain and update the following information as we traverse the document:

- TAG(e) which corresponds to the tag of the element.
- Sets of vertices from the XPath expression tree: SELF(e), ANCESTORS(e), PARENT(e), CHILDREN(e), and DESCENDANTS(e). A vertex v is in SELF(e) if e may embed into v. v is in PARENT(e) if the parent element of e may embed into v. $v \in$ CHILDREN(e) implies that some child of e may embed into v; $v \in$ DESCENDANTS(e), if some descendant of e may embed into v, and finally, $v \in$ ANCESTORS(e) implies that some ancestor of e in the tree may embed into v.
- An ordered set SUBTREES(e) of inflatable trees. This set corresponds to the inflatable trees constructed for the children of e.

For each vertex v in the XPath expression, the algorithm maintains COUNT(v), which represents how many elements e in the active stack contain v in SELF(e).

Algorithm Overview We first describe our algorithm with respect to a projection set that contains a single XPath expression, and then, discuss how to extend the algorithm for multiple XPath expressions. The essence of the algorithm is simple — materialize an element if it could participate in an embedding. As a tree is traversed and events are generated, for each vertex in the tree representation of the input XPath expression, the algorithm keeps track of the forward and backward constraints that have been satisfied. The following two conditions are used to determine whether a given element may participate in an embedding:

- Satisfaction of Backward Constraints : Let an element e belong to an embedding $\mathcal{E}$ of T into D such that for some vertex v, $\mathcal{E}(v) = e$. For each vertex v' in $\mathcal{B}(v)$, there must be some ancestor of e, e' such that $\mathcal{E}(v') = e'$, and the relation between e and e' satisfies the edge constraint between v and v'. This is a straightforward consequence of the definition of embeddings. At a start element event for an element, we verify that if the label of e matches some vertex v, then for each vertex $v' \in \mathcal{B}(v)$, one can find such a candidate e'. The vertex sets $\textsc{self}(e)$, $\textsc{parent}(e)$ and $\textsc{ancestors}(e)$ are used for this purpose. For example, if $\textsc{axis}(v, v') =$ `ancestor`, we require that $\textsc{ancestors}(e)$ contains v'. Otherwise, e cannot participate in an embedding for v. For `ancestor-or-self` constraints, we require that v' be present in the $\textsc{ancestors}(e)$ or $\textsc{self}(e)$ vertex sets.
- Satisfaction of Forward Constraints : A similar statement can be made for forward vertex sets. Let an element e belong to an embedding $\mathcal{E}$ of T into D such that for some vertex v, $\mathcal{E}(v) = e$. For each vertex v' in $\mathcal{F}(v)$, there must be some descendant of e, e' such that $\mathcal{E}(v') = e'$, and the relation between e and e' satisfies the edge constraint between v and v'. At the end element event, the algorithm can verify that if the label of e matches some vertex v, then such a candidate e' exists for all vertices $v' \in \mathcal{F}(v)$. The vertex sets $\textsc{self}(e)$, $\textsc{children}(e)$ and $\textsc{descendants}(e)$ are used for this purpose in a similar manner to the use of the $\textsc{self}(e)$, $\textsc{parent}(e)$ and $\textsc{ancestors}(e)$ sets for backward constraints.

At an end element event, the algorithm determines (given the current information) whether the current element e or some node in its subtree is a possible candidate for an embedding. If so, the algorithm materializes the element; otherwise, it creates an inflatable node for the element. The $\textsc{count}$ data structure is used to prune information, as will be described shortly.

The handling of multiple XPath expressions is a straightforward extension to the handling of a single XPath expression — the algorithm evaluates each of them in parallel. An element is materialized if it is required by any of the XPath expressions.

Algorithm Details The inflatable tree construction algorithm processes a given XPath expression $T = (V_T, E_T)$ and a document $D = (N_D, E_D)$ to construct the inflatable tree in a bottom-up manner — at each end element event for an element, the algorithm decides whether to build a materialized node or an inflatable node for that element based on decisions taken for its children.

- Initially, set the active stack to be empty.
- At a start element event for an element e, push e on to the active stack.
  1. Set $\textsc{ancestors}(e)$, $\textsc{children}(e)$, $\textsc{descendants}(e)$ to be empty.
  2. If e is the root of the document, set $\textsc{parent}(e)$ to be empty, otherwise set $\textsc{parent}(e)$ to equal $\textsc{self}(e')$, where e' is the parent of e in the tree.
  3. Set $\textsc{self}(e)$ to be all vertices v in the XPath expression tree such that $\textsc{tag}(e)$ matches $\textsc{label}(v)$. For each vertex v in $\textsc{self}(e)$ try to satisfy all

the constraints in $\mathcal{B}(v)$ using SELF(e), PARENT(e) and ANCESTORS(e) as described previously. If all constraints for v cannot be satisfied, remove v from SELF(e). Continue this process until no further vertices can be removed from SELF(e). For each vertex v remaining in SELF(e), increment COUNT(v).

– At an end element event for an element e:
  1. If SELF(e) is non-empty, for each vertex v in SELF(e), check for the satisfaction of forward constraints using the SELF(e), CHILDREN(e) and DESCENDANTS(e) vertex sets. If the forward constraints cannot be satisfied for v, remove v from SELF(e) and decrement COUNT(v). If COUNT(v) becomes 0, we can prune DESCENDANTS(e). If DESCENDANTS(e) does not contain v, and COUNT(v) is 0, then all vertices v' that are descendants of v in the XPath expression tree can be removed from DESCENDANTS(e). Consider a v' that is in DESCENDANTS(e) such that v' is a descendant of v in the XPath expression. For an element e' in the subtree rooted at e to be mapped to v' in some embedding, there must be an element e'' that is mapped to v in that embedding. Since v' is a descendant of v in the XPath expression tree, e'' must be an ancestor of e'. If COUNT(v) is 0 and DESCENDANTS(e) does not contain v, then observe that there can be no such e'' in the tree.
  2. Repeat Step 1 for vertices in SELF(e) until no more vertices can be removed from SELF(e).
  3. If SELF(e) and DESCENDANTS(e) are *both* empty, construct an inflatable node for e (and the subtree rooted under it), and discard the contents of SUBTREES(e).
  4. If SELF(e) is not empty and DESCENDANTS(e) is empty, construct a materialized node for e. If SUBTREES(e) is not empty, construct a single inflatable node that represents all the children of e and insert this inflatable node as a child of the materialized node corresponding to e.
  5. Otherwise, construct a materialized node for e and insert SUBTREES(e) as the children of this materialized node.
  6. Let e' be the parent of e in D. Update CHILDREN(e') to CHILDREN(e') $\bigcup$ SELF(e). Set DESCENDANTS(e') to DESCENDANTS(e') $\bigcup$ DESCENDANTS(e) $\bigcup$ SELF(e). For each vertex v remaining in SELF(e), decrement COUNT(v).

  In all cases, once the node for e is constructed, e is popped off the active stack and the node corresponding to e is appended to SUBTREES(e'), where e' is the current head of the stack (corresponds to e's parent in the document). If the node corresponding to e and the tail of SUBTREES(e') are both inflatable nodes, the two nodes are merged.

5 Implementation

We use a custom parser to generate the start and end element events corresponding to the depth-first traversal of the document. A key characteristic of the parser is the ability to support controlled parsing over a byte array — we can specify the start and end offsets of the byte array that the parser should use as the basis for parsing. This property is essential for the parsing of subtrees corresponding to inflatable nodes. Another feature of the parser is that at element event handlers, it provides offset information rather than materializing data as SAX does. For example, rather than constructing a string representation of the element tag's name, it returns an offset into the array and a length.

One challenge in the implementation of a projection algorithm is efficiency when complex axes are used. For example, Marian and Siméon report that document instance construction can degrade when XPath expressions involving `descendant` axes are used [8]. As we will demonstrate in Section 6, our algorithm scales well even in the presence of complex axes. The main reason for the efficiency of our implementation is a careful design of the data structures used to implement the algorithm of Section 4. We use bitmaps to represent much of the information that is necessary — set containment and union operations are encoded using efficient bitmask operations. As an optimization, our algorithm skips processing a subtree if it can detect that the subtree below the element cannot participate in any embedding. This happens if all the paths in the XPath set contain prefixes without any `ancestor` or `descendant` axes. For example, if the set of XPath prefixes is {/a/b/c, /a/d}, then if we encounter a start tag of a followed by an f, we can skip processing the subtree rooted at f.

Our system is implemented in Java. We use the Xerces [2] DOM representation as the underlying representation for the inflatable tree. Materialized nodes are represented as normal DOM nodes. Inflatable nodes have a special tag "__INFLATABLE__" and they contain two attributes indicating the start and end offsets in the byte representation of the document. The ability to use DOM as our underlying representation is a key advantage — we are able to run DOM-based XPath processors without modification on our inflatable trees; the semantics of projection guarantees that the inflatable nodes do not affect the result of evaluation of any XPath in the projection set!

6 Experiments

We used the queries of the XMark [11] benchmark set to evaluate the performance of our algorithm. In our experiment, the same benchmark code was used for both DOM and InflateX; the only difference being that for InflateX, the document was first projected with respect to a set of XPath expressions derived from the queries using the rules in [8]. In both cases, we used Xalan [1] as our XPath engine. We used a custom parser to generate appropriate events to construct both the inflatable tree, and in the DOM version, the full DOM data model instance. We used a custom parser rather than a standard XML parser such as Xerces [2] because our parser generates appropriate byte offset information in the events. We compared the performance of our parser for the construction of a full DOM instance with that of Xerces and found them comparable.[5] All

[5] The cost of constructing a DOM instance from a 10MB XMark file using our parser was 1312ms compared to 1612ms for Xerces.

Table 1. Projection sets involving uses of axes other than `child` and `descendant`.

Q21	{`//item[ancestor::africa]/name[following-sibling::payment]//mailbox//from`}
Q22	{`/site/closed_auctions/closed_auction/itemref[preceding-sibling::buyer]`, `/site/person/name[ancestor::people]`, `/site/regions//item[parent::europe]/name`}

experiments were run on a 1GHz IBM ThinkPad with 256MB of memory — the Java heap size was set at 128MB.

We will explore the efficiency of InflateX versus DOM in several dimensions: document construction time, query evaluation time, memory requirements, serialization, and dynamic projection. For both InflateX and DOM, the document is read from a file in the file system, the query is evaluated, and the results are serialized to a file. We will use the 20 original queries of the XMark benchmark. Since the XMark query set does not include queries that use axes such as `parent` and `following-sibling`, we have added two additional queries consisting of XPath expressions that use these axes. The projection sets corresponding to these two queries, which we refer to as Q21 and Q22 are provided in Table 1. All experiments were run on a 10 MB XMark file.

Construction Time Figure 7 compares the time taken to construct the in-memory projection using InflateX with that for constructing a DOM instance. As can be seen from the figure, our scheme is 2-3 times more efficient than DOM depending on the size of the projection. In Marian and Siméon, the document construction performance degrades with the presence of the `descendant` axis [8]. Our scheme is robust for descendant axes and performs well even when axes such as `ancestor` or `preceding-sibling` are used (as can be seen from the results for Q21 and Q22). The reason for the robustness is in the implementation of our algorithm. Our algorithm does not maintain much state apart from the projection tree that is being constructed; we encode much of the state using compact bitmaps.

Query Evaluation As in Marian and Siméon, our projection scheme improves query evaluation because the queries are evaluated over a smaller document. Figure 8 compares the execution of the XMark queries with that of a similar evaluation over a full DOM instance. Most of the XMark queries contain only child axes. The performance of these queries improves marginally as such XPaths can be efficiently evaluated without having to search subtrees. In the presence of descendant axes (Q7, Q19), we obtain factors of improvement of 13 and 2.5. This is because the XPath processor searches entire subtrees to match descendant nodes.

Memory Requirements In terms of the absolute memory sizes that can be handled, for DOM, the largest document that could be constructed in memory was 25 MB on our system (irrespective of the query). The amount of data that InflateX was able to handle depends on the projected set. For the projection path Q21 in Table 1, and for most other XMark queries, our projection scheme was

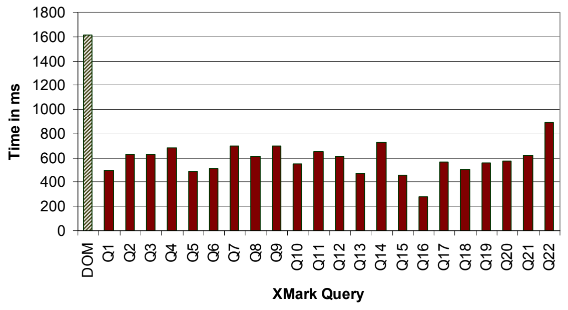

Fig. 7. Comparison of document construction time on a 10MB XMark file. The first column shows the cost of constructing a DOM in-memory instance. The remaining columns provide times for projection construction on the various queries.

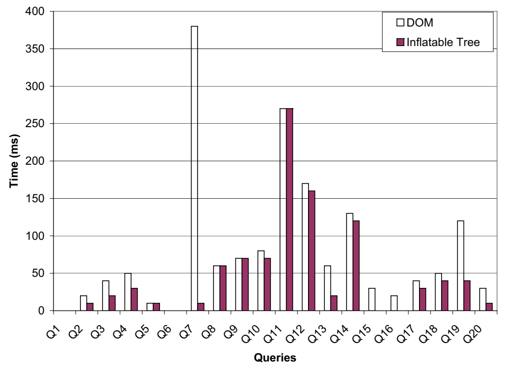

Fig. 8. Comparison of query evaluation time on a 10 MB XMark file.

able to handle documents of size upto 100 MB. For other queries, the largest document we could process was somewhere between 50 and 100MB. The size of the projection is small relative to the overhead of storing the byte array in memory.

Figure 9 measures the number of nodes in the inflatable trees for each of the XMark queries. On average, we materialize about 10% of the nodes. The number of inflatable nodes that we construct is of the order of the projection, and therefore, does not add much overhead.

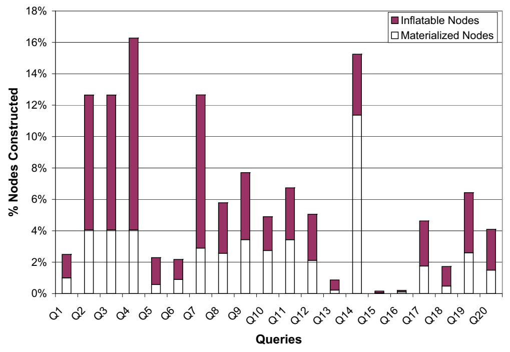

Fig. 9. Comparison of memory overhead on a 10MB XMark file. The total height of a column is the percentage of nodes in the original tree that are constructed (the tree contains 510946 nodes). Each column shows the breakdown in terms of materialized nodes and inflatable nodes constructed.

Serialization Many queries return large result sets that need to be serialized out as a sequence of bytes to a client. The definition of projection by Marian and Siméon would construct all nodes that might have to be serialized. These nodes would be traversed to generate the bytes corresponding to the result. Our inflatable trees allow for efficient serialization directly from the byte array when possible. Furthermore, we avoid the cost of having to construct all elements that are materialized solely because they are required for the output.

Table 2 compares the cost of query execution of the XPath expression `/site/regions/namerica/item` using different projections. The first uses our algorithm to build a projection based on inflatable trees. The second, *Output Projection*, constructs the subtrees of all output nodes in the document (as in Marian and Siméon).

The presence of the byte array corresponding to the document allows for a drastic reduction in the size of the projection, which in turn, reduces construction time. Furthermore, the cost of serialization reduces by a factor of four. The serialization of XML from a data model instance can be slow since the serializer must traverse the entire data model instance and output the appropriate XML constructs. The byte array allows our serialization mechanism to avoid this cost.

Table 2. Comparison of inflatable tree query execution time to the scheme that constructs the subtrees of all output nodes.

	Inflatable Tree	Output Projection
Construction	470ms	680ms
Serialization	70ms	380ms
Number of Nodes	5119	78923

Dynamic Projection One advantage of the inflatable tree representation over projection as defined by Marian and Siméon is that it allows clients to expand portions of the tree dynamically. For example, a client may choose to expand with respect to one set if an `if` branch is taken and another if the corresponding `else` branch is taken. Figure 10 explores the performance of dynamic projection in the common situation where a client first issues a query and then refines the query based on the results. In the experiment, the document is first projected with respect to the XPath expression `/site/regions/namerica`, and subsequently, the client refines the query with respect to XPath expression `/site/regions/namerica/item`. We compare the cost of dynamic projection over the inflatable tree to the cost of constructing a new projection (as would be done in Marian and Siméon). As can be seen, there can be a significant advantage to dynamic projection.

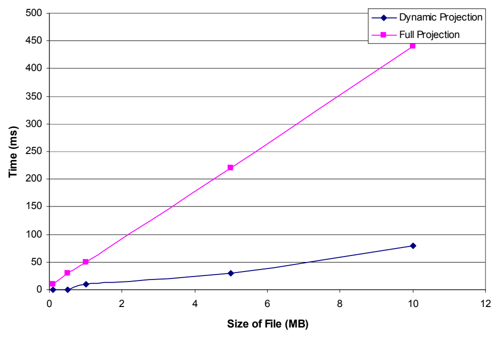

Fig. 10. Comparison of dynamically projecting a subtree of the document rather than projection over the entire document.

7 Conclusions

In this paper, we have proposed the inflatable tree data structure as a viable in-memory representation of XML. Our representation also supports dynamic projection of XML documents and efficient serialization of results to clients.

We have also developed a projection algorithm that can handle complex axes such as `ancestor` and `following-sibling`. Our experiments demonstrate that our algorithm constructs inflatable trees that are small compared to the full data instance, even when these complex axes are used. In addition to reducing the memory overhead of the in-memory representation of XML, our algorithm is efficient and can reduce the cost of constructing the instance significantly.

In the future, we plan to explore the use of schema information to drive the derivation of projections. Schema information in conjunction with the projection set of XPath expressions can be used to prune projections more precisely. Another area of interest is the exploration of automatically *deflating* trees, that is, determining from an XQuery expression, when a subtree in the XML document is no longer required.

References

1. Apache Software Foundation. *Xalan-Java*. `http://xml.apache.org/xalan-j`.
2. Apache Software Foundation. *Xerces2 Java Parser*. `http://xml.apache.org/xerces2-j`.
3. C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath processing with forward and backward axes. In *Proceedings of the 19th IEEE International Conference on Data Engineering (ICDE)*, pages 455–466, March 2003.
4. P. Bohannon, S. Ganguly, H. F. Korth, P. P. S. Narayan, and P. Shenoy. Optimizing view queries in ROLEX to support navigable result trees. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB)*, pages 119–130, 2002.
5. P. Buneman, M. Grohe, and C. Koch. Path queries on compressed XML. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB)*, pages 141–152, 2003.
6. C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *The VLDB Journal*, 11(4):354–379, 2002.
7. Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Transactions on Database Systems*, 28(4):467–516, 2003.
8. A. Marian and J. Siméon. Projecting XML documents. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB)*, pages 213–224, 2003.
9. G. Miklau and D. Suciu. Containment and equivalence for a fragment of XPath. *Journal of the ACM*, 51(1):2–45, 2004.
10. *Simple API for XML*. `http://www.saxproject.org`.
11. A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. Xmark: A benchmark for XML data management. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, pages 974–985, 2002.
12. World Wide Web Consortium. *XML Path Language (XPath) Version 1.0*, November 1999.
13. World Wide Web Consortium. *Document Object Model Level 2 Core*, November 2000.
14. World Wide Web Consortium. *XQuery 1.0: An XML Query Language*, August 2003. W3C Working draft.