Vectorizing and Querying Large XML Repositories

Peter Buneman   Byron Choi   Wenfei Fan   Robert Hutchison   Robert Mann   Stratis D. Viglas
University of Edinburgh
{opb@inf.ed,vibchoi@inf.ed,wenfei@inf.ed,r.hutchison@sms.ed,rgm@roe,sviglas@inf.ed}.ac.uk

Abstract

Vertical partitioning is a well-known technique for optimizing query performance in relational databases. An extreme form of this technique, which we call vectorization, is to store each column separately. We use a generalization of vectorization as the basis for a native XML store. The idea is to decompose an XML document into a set of vectors that contain the data values and a compressed skeleton that describes the structure. In order to query this representation and produce results in the same vectorized format, we consider a practical fragment of XQuery and introduce the notion of query graphs and a novel graph reduction algorithm that allows us to leverage relational optimization techniques as well as to reduce the unnecessary loading of data vectors and decompression of skeletons. A preliminary experimental study based on some scientific and synthetic XML data repositories in the order of gigabytes supports the claim that these techniques are scalable and have the potential to provide performance comparable with established relational database technology.

1 Introduction

This is a preliminary report on a method of storing large XML datasets in a fashion that allows them to be queried with efficiency that is comparable with – and may even surpass – that of conventional relational database technology. The method is based on a combination of two ideas: the first is a generalization of a vertical or "vectorized" organization of tabular data to XML documents; the second is the use of a compression technique that enables us to keep the tree-like structure of an XML document in main memory. As an example of what is achievable by this method, a simple select/project XQuery on an 80 gigabyte astronomy XML dataset took 37 seconds, while the same query in SQL on the same dataset stored in a relational database reportedly takes over 200 seconds on a comparable machine [17]. The reason for this speedup is simple: the XML query performed the equivalent of reading only 3 columns of a 368-column table, and the I/O was thus reduced; the same efficiency could be achieved by conventional vertical partitioning of relational data. The novelty we claim is that the same technique can be applied to a native XML store and will generalize to queries on relatively complex hierarchical data.

The idea of implementing a relational database by contiguously storing the columns of a table is almost as old as relational databases [4]. The benefit is that queries that only involve a small number of columns require less I/O. Moreover, there are dramatic performance improvements to be made if main-memory vector manipulation techniques can be applied to all or parts of these columns. The idea has re-emerged in various places: in [8, 14] for object-oriented databases and in [2] for speeding up transfer between main memory caches. It has also been used commercially in Sybase IQ [19] and recently in financial analysis software where it is combined with vector processing language technology and has been successfully used to support data integration [10]. In order to extend the idea to XML we make use of some ideas in two recent pieces of research:

XMILL. The "semantic compressor" developed by Liefke and Suciu [20] extends the idea of vertical partitioning to XML. The "columns" – we shall call them *vectors* in this case – are the sequences of data values appearing under all paths bearing the same sequence of tag names. In addition to storing these columns, one also needs to store the tree-like structure of the document, the *skeleton*. In XMILL the purpose of this decomposition was to compress the XML document. Here we *do not* compress the columns, and we use a *different method* for compressing the skeleton.

Skeleton Compression. We extend [9] in which the tree-like structure of the skeleton is compressed into a DAG by sharing common subtrees. In that paper the compressed skeleton was then expanded in order to represent the result of XPath evaluation. In contrast, here we generate a *new* – usually highly contracted – *skeleton* to represent the result of *XQuery* without decompression. In fact, query evaluation proceeds along the same general lines as that of relational algebra. Just as each evaluation step of the relational algebra yields a new table, each evaluation step in our evaluation process generates a new skeleton and a new set of vectors.

Our claim is that it is possible to construct a native XML store and query engine that will match or outperform conventional relational database systems on highly regular data and will continue to work well on irregular data sets. We should temper this claim with a few observations. First, our results are highly preliminary and we can hope to do little more than convince the reader of the credibility of this claim. Second, while we support the claim that vectorized representations may provide better query performance,

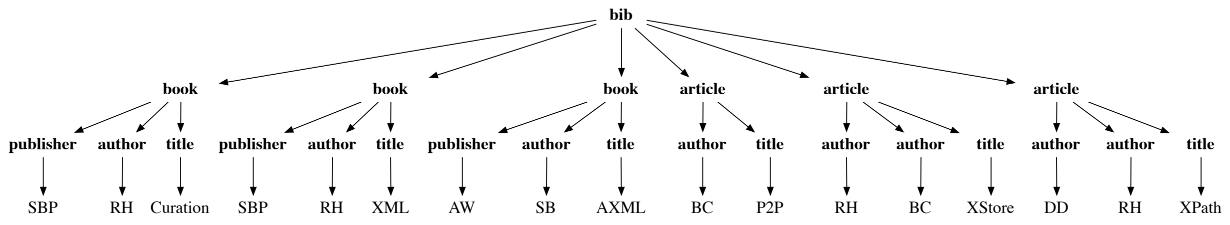


Figure 1. An XML tree T

this is something of a "cheap shot" at established relational technology, which provides much more functionality than efficient query languages. For example, updates and locking may cause grave problems in vectorized representations. Fortunately, XML documents are typically static, and if not, (see Sect. 6) there may be promising techniques for updating vectorized XML data. Finally, we note that *vertical partitioning* is already a well-understood and widely-used technique in the relational context, and *vectorization* is simply an extreme case of vertical partitioning in which each column is a partition.

In the following sections we describe how this decomposition of XML can be used in efficient query processing. The main contributions are as follows.

- graph reduction: we describe a useful fragment of XQuery (Sect. 3) and an evaluation technique for vectorized data (Sect. 4);
- complexity results: lower and upper bounds for query evaluation over vectorized data are given in Sect. 3;
- preliminary experimental results: the effectiveness of this approach is demonstrated in Sect. 5.

Section 6 discusses related work and topics for future work.

2 A Vectorized Representation of XML

In this section we extend techniques from [20] and [9] for representing XML documents. These will be the basis for our implementation of XQuery.

2.1 Vectorizing XML

Figure 2 illustrates the basic method of decomposing an XML document. Consider the depth-first traversal (which is equivalent to a linear scan of the document) of a node-labeled tree representation of the document as depicted in Fig. 1. Each time we encounter a text node, we append the text to a vector whose name is the sequence of tag values on the path to that node. For example, when we first encounter the text node `Curation` we are on a path `/bib/book/title`, so we append `Curation` to the vector named by this path. As we proceed through the tree we subsequently encounter the text nodes `XML` and `AXML`. These values are appended, in order, to the `/bib/book/title` vector. At the end of the traversal we have generated the vectors:

```
/bib/book/publisher:    [SBP, SBP, AW]
/bib/book/author:       [RH, RH, SB]
/bib/book/title:        [Curation, XML, AXML]
/bib/article/author:    [BC, RH, BC, DD, RH]
/bib/article/title:     [P2P, XStore, XPath]
```

Each of these vectors corresponds to a path of labels that leads to a non-empty text node.

Now suppose that during this traversal we also generate a tree in which each of the text nodes is replaced by a mark (#) indicating the presence of text in the original document. This tree is called the *skeleton* and the pair (S, V) consisting of the skeleton and the vectors is the basis for the decomposition in XMILL [20].

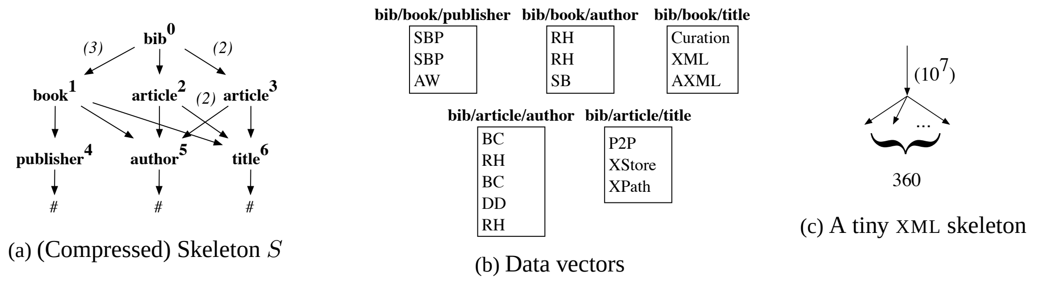
The important observation is that the original XML tree can be faithfully reconstructed from (S, V). To see this consider a depth-first traversal of S. As we traverse S we keep a note of the sequence of path tags (a stack) from the root. When we first encounter a node, we emit its start tag; when we leave it we emit its end tag; and when we encounter a text marker (#), we emit the next text value from the appropriate vector (we keep a cursor or index into each vector).

This method is faithful in that it is an order-preserving reconstruction of the original XML document. It can also handle – though we have not illustrated this – attributes and nodes with mixed content. The idea of XMILL is to achieve good compression of an XML document by separately compressing the vectors and a *serial* representation of the skeleton using standard [31] text compression techniques. However, we *depart from* XMILL in that we do not compress the vectors, and we use an entirely different technique for compressing the skeleton[1]. Moreover, we study efficient evaluation of XQuery directly over compressed skeletons, an challenging issue beyond the scope of [20].

2.2 Skeleton compression

Returning to Fig. 1, consider the three `book` nodes. Once we have replaced the text by markers, these three nodes have identical structure. Therefore we can replace them by a single structure and put three edges from the `bib` node to the top `book` node in this structure. Moreover, since these edges occur *consecutively* we can indicate this with a single edge together with a note of the number of occurrences. Thus, working bottom-up, we can compress

[1]There is some evidence [3] that vector compression in which the components of a vector are individually compressed, can be used effectively in conjunction with query evaluation.



(a) (Compressed) Skeleton S (b) Data vectors (c) A tiny XML skeleton

Figure 2. An XML tree, its skeleton and storage

the skeleton into a DAG as shown in Fig. 2(a). Multiple consecutive edges are indicated by an annotation (n), and an edge without annotation occurs once (in the DAG). In contrast to [20] the DAG skeleton is *compressed* by sharing subtrees with the same structure. In the sequel we consider compressed skeletons only, also referred to as skeletons.

We define the *vectorized representation* (or *vectorized tree*) of an XML tree T, denoted by VEC(T), to be a pair (S, V), where S is the skeleton of T and V is the collection of all data vectors of T. It is easy to verify that for any XML tree T, there exists a vectorized representation VEC(T) of T that is *unique* up to isomorphism.

The compressed skeleton can be implemented as a main-memory data structure. It should be apparent that, with the addition of a counter for multiply-occurring edges, a depth-first traversal of the compressed skeleton can be arranged with exactly the same properties of the original skeleton. Thus we still have a lossless reconstruction of the original XML document from its vectorized representation.

Some statistics for compression are reported in [9] for a range of XML data sets. The compressed skeleton almost always fits comfortably into main memory, and even when this is not possible, there are techniques for recursively vectorizing the skeleton and paying the price of an additional join in the query processor (we do not report on this here).

The advantages of this compression are twofold. First, the skeleton of regularly structured data compresses very well. In an extreme case, the astronomy data set that we use in Sect. 5 consists of a single table with roughly 360 columns and 10^7 rows. The compressed skeleton, as shown in Fig. 2(c), is tiny. In fact any reasonable XML representation of relational or array data will compress similarly. It should be noted that this compression is independent of any type/schema information such as a DTD, and moreover, base type information, such as that provided by XML-Schema, is certainly of help in further improving representation of the vectors.

The second advantage is part of the basis for our results. Although the skeleton can compress extremely well, it is nontrivial to use its compressed form directly for querying. In [9] the skeleton was *expanded*: new nodes were added to represent the set of nodes in the original tree that would be selected by an XPath query. Unfortunately uncompressed skeletons, even after doing the obvious of encoding the tags, can be prohibitively large. Thus our XQuery evaluation will generate new skeletons, which will typically be smaller than the original compressed skeleton. Consider, *e.g.,* the query that selects the books in Fig. 1 or a select-project query on the astronomy table. In both cases the skeleton for the result will be smaller. Moreover, we shall see that in many cases the smaller output skeleton can be constructed from the input skeleton without intermediate decompression.

We end this section with two straightforward results that are central to the later development.

Proposition 2.1: *The vectorized representation and compressed skeleton of an* XML *tree T can be computed in linear time in $|T|$.* □

The only nontrivial part of this observation is that the compressed skeleton can be constructed in linear time. This is essentially the folkloric "hash-cons" method.

Proposition 2.2: *An* XML *tree T can be reconstructed from its vectors and compressed skeleton in linear time in $|T|$.* □

Note that this is linear in the size of the *output* (i.e. the original document.) It is easy to construct pathological cases in which the compression is exponential. Unfortunately we have not encountered any practical XML that compresses quite so well!

3 An XQuery Fragment

In this section we study the problem of XQuery [12] evaluation over vectorized data, and show that this problem introduces new optimization issues. To simplify the discussion, we consider a fragment of XQuery, denoted by XQ. Below we first present XQ. We then state the query evaluation problem and establish complexity results for the problem. Finally we introduce a graph representation of XQ queries.

3.1 XPath and XQuery

XPath. We consider a fragment of XQuery defined in terms of simple XPath [13] expressions. This class of *simple* XPath expressions, denoted by P, is defined by:

$$p \;::=\; l \;\mid\; p/p \;\mid\; p[q], \qquad q \;::=\; p \;\mid\; p = c$$

where l denotes an XML tag, '$/$' stands for the *child*-axis, and q in $p[q]$ is called a *qualifier* in which c is a constant of text value (PCDATA).

A query p of P is evaluated at a *context node* v in an XML tree T, and its result is the set of nodes of T reachable via p from v, denoted by $v[\![p]\!]$. Qualifiers are interpreted as follows: at a context node v, $[p]$ holds iff $v[\![p]\!]$ is nonempty, *i.e.*, there exists a node reachable via p from v; and $[p = c]$ is true iff $v[\![p]\!]$ contains a text node whose value equals c.

A *path term* ρ of P is an expression of the form v/p, where v is either a document name doc or a variable $\$x$, and p is a P expression. By treating v as the context node, ρ computes the set of nodes reachable via p, *i.e.*, $v[\![p]\!]$. We use $[\![\rho]\!]$ to denote $v[\![p]\!]$ when v is clear from the context.

We shall also consider an extension of P by allowing the wildcard '$*$' and the *descendant-or-self*-axis '$//$'. We use $P^{[*,//]}$ to denote this extension.

XQuery. We consider a class of XQuery of the form:

```
<result>
   for      $\$x_1$ in $\rho_1$,
            $\$x_2$ in $\rho_2$,
            ...
            $\$x_n$ in $\rho_n$
   where    $\rho'_1 = \rho''_1$ and ... and $\rho'_k = \rho''_k$
   return   $exp(\varrho_1, \varrho_2, \ldots, \varrho_m)$
</result>
```

where

- ρ_i is a path term of P;
- ρ'_j (ρ''_j) is either a text-value constant or a path term;
- $\rho' = \rho''$ holds iff the sets of nodes reachable via ρ and ρ' are not disjoint; that is, the intersection of $[\![\rho']\!]$ and $[\![\rho'']\!]$ is nonempty (assume $[\![c]\!] = c$);
- ϱ_s is a path term of the form $\$y_s/p_s$, where $\$y_s$ is one of the variables $\$x_1, \ldots, \$x_n$.
- exp is a sequence of XML *element templates*, where each template is the same as an XML element except that it may contain $\varrho_1, \ldots, \varrho_m$ as parameters; the template yields an XML element given a substitution of concrete XML elements/values for $\varrho_1, \ldots, \varrho_m$.

The semantics of such a query Q is standard as defined by XQuery [12]. Posed over an XML tree T, Q returns an XML tree rooted at a `result` node with children defined by $exp(\varrho_1, \ldots, \varrho_m)$. More specifically, let ϱ_s be $\$y_s/p_s$ for $s \in [1, m]$. Then for each tuple of values computed by the `for` and `where` clauses for instantiating the *variable tuple* $\langle \$y_1, \ldots, \$y_m \rangle$, the path terms $p_1, \ldots, p_m$ are evaluated, their results are substituted for $\varrho_1, \ldots, \varrho_m$, and with the substitution a sequence of XML elements defined by $exp(\varrho_1, \ldots, \varrho_m)$ are added to the tree as children of the `result` node. Let us refer to such a value tuple as an *instantiation* of the variable tuple $\langle \$y_1, \ldots, \$y_m \rangle$.

We use XQ for XQuery of this form when the XPath expressions are in the class P. Similarly, we use XQ$^{[*,//]}$ to denote the XQuery fragment defined with $P^{[*,//]}$ expressions.

Example 3.1: Posed over the XML data of Fig. 1, the following XQ query Q_0 finds all book and article titles by authors who have written a book and an article, with the book having been published by 'SBP'.

```
<result>
    for     $d in doc("bib.xml")/bib
            $b in $d/book
            $a in $d/article
    where   $b/author = $a/author and
            $b/publisher = 'SBP'
    return  $b/title, $a/title
</result>
```

The result of the query is shown in Fig. 3(a). □

For any XQ (or XQ$^{[*,//]}$) query Q, there is an equivalent XQ (or XQ$^{[*,//]}$) query Q' such that Q' does not contain any qualifiers in its embedded XPath expressions. Indeed, the XPath qualifiers in Q can be straightforwardly encoded in Q' by introducing fresh variables and new conjuncts in the `where` clause of Q'. Thus, w.l.o.g., in the sequel we only consider queries without XPath qualifiers.

The fragments XQ and XQ$^{[*,//]}$ can express many XML queries commonly found in practice. One can easily verify that even the small fragment XQ is capable of expressing all relational conjunctive queries.

3.2 Query Evaluation over Vectorized Data

The problem of XQuery evaluation over vectorized XML data can be stated as follows. Given the vectorized representation VEC(T) of an XML tree T and an XQuery Q, the problem is to compute the *vectorized representation* VEC(T') of another XML tree T' such that $T' = Q(T)$, where $Q(T)$ stands for applying Q to T. Observe that both the input and the output of the computation are vectorized XML data.

Example 3.2: The vectorized representation of the result of Q_0 given in Fig. 3(a) is (S_0, V_0) shown in Fig. 3(b). □

A *naive evaluation algorithm* for XQuery over vectorized XML trees works as follows. Given a vectorized XML tree VEC(T) and a query Q,

1. decompress VEC(T) to restore the original T;
2. compute $Q(T)$;
3. vectorize $Q(T)$.

Note that steps (1) and (3) take linear time in $|T|$ and $|Q(T)|$ respectively. Thus the complexity for evaluating queries over vectorized XML trees does not exceed its counterpart over the original XML trees. From this and the proposition below we obtain an upper bound for evaluating XQ$^{[*,//]}$ queries over vectorized XML data.

Proposition 3.1: *For any* XQ$^{[*,//]}$ *query* Q *and* XML *tree* T, $Q(T)$ *can be computed in at most* $O(|T|^{|Q|})$ *time.* □

Proof sketch: The complexity can be verified by a straightforward induction on the structure of Q. □

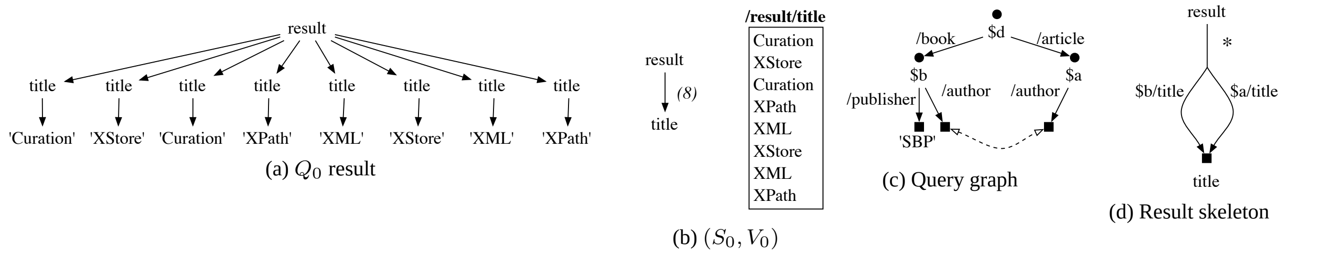


Figure 3. Result and representation of an XQ query

Can we do better than exponential time? Intuitively this is possible under certain conditions: as mentioned in the last section, vectorization can lead to an exponential reduction in size. Furthermore, the proposition below gives us an upper bound for the size of the skeletons and the number of data vectors in the vectorized query results.

Proposition 3.2: *Let* VEC*(T) = (S, V) be the vectorized representation of an* XML *tree T, Q be an arbitrary* $\text{XQ}^{[*,//]}$ *query, and* VEC*(T') = (S', V') be the vectorized representation of $T' = Q(T)$. Then $|S'|$ is at most $O(|S|\ |Q|)$ and the cardinality of V',* i.e., *the number of vectors in V', is no larger than the cardinality of V.* □

Proof sketch: This is because the number of *distinct* subtrees in T' is bounded by $O(|S|\ |Q|)$, and the number of *distinct* paths in T' is no larger than the cardinality of V. □

This suggests that if we can directly compute the vectorized representation of $Q(\text{VEC}(T'))$ without first decompressing VEC(T'), we may be able to achieve an exponential reduction in evaluation time. This presents new optimization opportunities as well as new challenges given rise by query evaluation over vectorized XML data.

Unfortunately, in the worst case the lower bound for query evaluation is exponential, and may be as bad as uncompressed evaluation, even for XQ queries.

Proposition 3.3: *The lower bound for the time complexity of evaluating* XQ *queries Q over vectorized* XML *trees* VEC*(T) is $|T|^{|Q|}$.* □

Proof sketch: This can be shown by constructing a set of XQ queries and a set of XML trees such that for any query Q and tree T in these sets, in the vectorized form (S', V') of $Q(T)$, the size of a data vector in V' is $|T|^{|Q|}$. □

Putting these together, despite the worst-case complexity (Prop. 3.3), one can often expect exponential reduction in evaluation time by avoiding intermediate decompression (Prop. 3.2). Moreover, as will be seen shortly, vectorization allows lazy evaluation and thus reduces I/O costs.

3.3 Query Graphs

An XQ query Q can be represented as a pair (G_q, G_r) of graphs, called the *query graph* and the *result skeleton* of Q, which characterize the `for`, `where` clauses and the result template of Q, respectively.

Query graph. The query graph G_q of an XQ query Q is a rooted acyclic directed graph (DAG), derived from the `for` and `where` clauses of Q as follows.

- The *root* of G_q is a unique node labeled `doc` indicating a document root; to simplify the discussion we assume that Q accesses a single document; this does not lose generality since one can always create a single virtual root for multiple documents.
- For each distinct variable $\$x$ (and each constant c) in the `for` and `where` clauses of Q, there is a distinct node in G_q labeled by $\$x$ (or c).
- For each path term $\rho = v/p$, where v is either `doc` or a variable $\$z$, there exists a node e in G_q representing the *end point* of ρ, and there exists a *tree edge* labeled p from v to e. If Q contains a clause `for` $\$x$ `in` ρ, then the node representing $\$x$ is the same as e.
- For each equality $\rho = \rho'$ in the `where` clause, there is an *equality edge*, indicated by a dotted line, between the end point of ρ and that of ρ'.

For example, Fig. 3(c) depicts the query graph of the XQ query of Example 3.1. Here, circle nodes denote variable nodes, and square nodes indicate end points.

Result skeleton. Abusing the notion of skeletons given earlier, the result skeleton G_r of Q is a tree template that characterizes the `return` clause of Q. For example, Fig. 3(d) depicts the result skeleton of the XQ query of Example 3.1. Note that for each instantiation of the variable tuple $\langle \$y_1, \ldots, \$y_m \rangle$, a sequence of new children of the form $exp(\varrho_1, \ldots, \varrho_m)$ are generated for the root; this is indicated by the '$*$' label tagging the edge below the root in Fig. 3(d).

The query graph and result skeleton of a query can be automatically derived from the query at compile time. Note that for any meaningful XQ query, *i.e.,* a query that is not empty over all XML trees, its query graph and result skeleton are DAGs. Moreover, each node has at most one incoming tree edge. Thus we say that a node v is the *parent* of w (and w is a *child* of v) if there is a tree edge from v to w.

Conceptual evaluation. The result skeleton of a query Q can be readily understood as a function that takes an instantiation of its variables as input and constructs the result XML tree by expanding the skeleton. Evaluation of the query graph of Q is to instantiate variables needed by the result skeleton. We next present a conceptual strategy for evaluating the query graph of Q over *vectorized* XML *data*.

A query graph imposes a dependency relation on its variables: if $\$y$ is the parent of $\$x$, then the value of $\$x$ cannot be determined before the value of $\$y$ is fixed. Furthermore, if there is an equality edge associated with a variable, then the equality condition cannot be validated before the variable is instantiated.

Given a vectorized tree VEC(T) = (S, V), the conceptual evaluation strategy traverses the query graph G_q top-down, mapping the nodes of G_q to the nodes of S or data values in the vectors of V. It starts from the root doc of G_q and maps doc to the root of S. For each node v encountered in G_q, suppose that v is mapped to a node w in S. Then it picks the leftmost child v' of v whose evaluation does not violate the dependency relation. Suppose that the tree edge from v to v' is labeled path p. It traverses S from w to a node w' reachable via p. If w' is in S, then it maps v' to w', *i.e.*, v' is instantiated to be w', and it inductively evaluates the children of v' in the same way. If w' is a text node, then it loads and scans the corresponding data vector of V and picks a data value to instantiate v'. It moves upward to the parent of v and proceeds to process the siblings of v after all the children of v have been processed, or when it cannot find such a node w' (backtrack). It checks equality conditions $\rho' = \rho''$ by checking whether $[\![\rho']\!]$ and $[\![\rho'']\!]$ are disjoint or not, scanning data vectors if necessary. If all these conditions are satisfied, an instantiation of the variable tuple is produced and passed to the skeleton function to increment the result XML tree. The process terminates after all the instantiations are exhaustively computed. Note that process always terminates since the query graph is a DAG.

Example 3.3: Consider evaluating the query graph of Fig. 3(c) over the vectorized XML data (S, V) of Figs. 2(a) and 2(b). The variable $\$d$ is first mapped to the `bib` node of S. It then traverses S via the path `book` to instantiate $\$b$; similarly for $\$a$. For each $\$b$ value, the data vector for `doc/book/publisher` is scanned and the equality condition is checked; furthermore, the data vectors `doc/book/author` and `doc/article/author` are scanned to check whether or not $\$b$ and $\$a$ have a common author. An instantiation ($\$b, \$a$) is passed to the result skeleton if it satisfies all these conditions. Given these instantiations, the result skeleton constructs the output tree and vectorizes it, yielding Fig. 3(b). □

Several subtleties are worth mentioning. First, to simplify the discussion, in a query graph we ignore the order imposed by nested `for` loops in the query, which is easy to incorporate. Note that the query graph captures the dependency relation on variables via tree edges, which is consistent with the order of nested `for` loops. Second, evaluation of XQ queries over vectorized data is more intriguing than evaluation of XPath queries, which was studied in [9]. An XQ query needs to construct a new skeleton; moreover, each instantiation of the variable tuple of its result skeleton increments its output XML document, following a certain order; furthermore, it can be verified that the skeleton of the vectorized output document cannot be decided by the query and source skeleton alone. These are not the concerns of [9].

This conceptual evaluation strategy is obviously inefficient. First, the same data vector is repeatedly scanned for each variable instantiation; this overhead is evident when the main memory has limited capacity to hold all the relevant data vectors, which is typical in practice. Second, at each node encountered during the evaluation, there are typically multiple children available to be processed, and these children can be evaluated in different orderings; experience from relational optimization tells us that different evaluation orderings may lead to vastly different performance. We shall study these optimization issues in the next section.

Another optimization issue concerns query graph minimization. Similar to minimal tableau queries [1], a notion of *minimal query graphs*, *i.e.*, graphs with the least number of nodes, can be defined for XQ queries. Intuitively, a minimal query graph can be evaluated more efficiently than query graphs with redundant nodes. The problem of *query graph minimization* is, given the query graph of an XQ query, to find a minimum query graph equivalent to the input graph. Unfortunately, the problem is intractable.

Proposition 3.4: *The problem of query graph minimization is NP-hard for* XQ *queries.* □

Proof sketch: By reduction from tableau query minimization, which is intractable [1]. □

4 Query Evaluation by Graph Reduction

We next present an algorithm for evaluating XQ queries over vectorized XML data. In light of the inherent difficulty of the problem observed in the previous section, our optimization algorithm is necessarily approximate, *i.e.*, it does not always find the optimum evaluation plan. Our key technical idea is to exploit lazy evaluation, to avoid unnecessary scanning of data vectors and to reduce decompression of skeletons. To this end we propose a novel *graph reduction* framework that allows us to apply relational optimization techniques to querying vectorized XML data. To simplify the discussion we consider XQ queries; but the technique can be readily extended to XQ$^{[*,//]}$ and larger XQuery classes.

4.1 An Evaluation Algorithm

Consider the query graph G_q of an XQ query, *e.g.*, Fig. 3(c). Observe that each edge in G_q can be readily understood as an extension of an operation in the relational algebra:

- A tree edge from a variable $\$y$ to $\$x$ labeled with an XPath expression p, denoted by $p(\$y,\$x)$, is like a *projection*: extracting the p descendant of the $\$y$ node.
- A tree edge from a variable $\$y$ to a constant c, denoted by $p(\$y,c)$, is reminiscent of *selection*: checking whether $\$y$ has a p descendant with value c.
- An equality edge between nodes v_1 and v_2, denoted by $eq(v_1,v_2)$, is similar to a *join*.

To evaluate the query one needs to find an efficient plan for processing these operations. The naive algorithm given in Sect. 3 evaluates each operation for *a node at a time*. For instance, for a projection operation, *i.e.,* a tree edge labelled p from $\$y$ to $\$x$, it repeatedly evaluates p for each $\$y$ value, and thus it repeatedly scans the same data vector for the same operation *w.r.t.* each variable instantiation.

To avoid scanning data vectors unnecessarily, we evaluate each operation for *a collection at a time*. Referring to the projection operation above, we first compute the sequence of all values of $\$y$, called the *instantiation of* $\$y$; we then evaluate p and compute all instantiations for $\$x$ once for the entire collection of $\$y$ values, scanning the corresponding data vector once. Reflected in the query graph, this can be understood as *merging* the $\$y$ and $\$x$ nodes into a single node ($\$y,\$x$), which holds the instantiations of $\$y$ and $\$x$. In other words, this is to *reduce* the graph by removing one edge from it. Thus we refer to this idea as *graph reduction*. In a nutshell, we evaluate a query by reducing its graph one edge at a time; the reduction process terminates after the graph is reduced to a single node, which holds the *instantiation of the query*, namely, a sequence of all value tuples for the variable tuple of the result skeleton. At this point the query instantiation is passed to the result skeleton, which constructs the result XML tree with the instantiation.

The next question is: in what order should we evaluate the operations in a query graph? Certainly such an ordering should observe the dependency relation on the variables in the graph, as described in Sect. 3. But there are typically multiple possible orderings. Leveraging on the connection between edges in a query graph and operations of the relational algebra, we use the well-developed techniques for relational query optimization. In particular, in our algorithm we topologically sort the operations in a query graph by using algebraic optimization rules [30], *e.g.,* performing selections before join. This sorting operation could be cost/heuristics-based, by means of a mild generalization of cost functions and heuristics for relational operations.

Another question concerns simplification of a query graph at compile time. There are possibly *redundant variables* in a query graph. Consider, *e.g.,* a tree edge labelled p from $\$y$ to $\$z$ followed by a tree edge labelled p' from $\$z$ to $\$x$, where $\$z$ is not used anywhere else in the query. Since there is no need to instantiate $\$z$, at compile time we shortcut the redundant $\$z$ by merging the two edges into a single edge labelled p/p' from $\$y$ to $\$x$. We use this simple strategy because it is beyond reach to find an efficient algorithm to minimize XQ queries by Prop. 3.4.

```
Input:  the vectorized XML representation $(S,V)$ of $T$;
        and an XQ query $Q$ represented as $(G_q,G_r)$, which
        are the query graph and result skeleton of $Q$.
Output: the vectorized representation $(S',V')$ of $Q(T)$.
1.  $S' := G_q$;   $V' = \emptyset$;
2.  remove redundant variables from $G_q$;
3.  topologically sort operations in $G_q$ w.r.t. variable dependency
    and by means of relational algebraic optimization rules;
4.  let $L$ be the sequence of operations in the topological order;
5.  for each $e \in L$ do
6.      reduce $(G_q,e)$;
7.  let $I$ be the instantiation of the query from the reduction;
8.  for each $t \in I$ do
9.      $S'$ := expand $(S',t)$;
10.     $V'$ := populate $(V',t)$;
11. return $(S',V')$;
```

Figure 4. Algorithm `eval`

Putting these together, we outline our evaluation algorithm, Algorithm `eval`, in Fig. 4. The algorithm takes as input a vectorized representation VEC(T) of an XML tree T and a graph representation (G_q,G_r) of an XQ query Q; it returns as output the vectorized representation (S',V') of the query result $Q(T)$. Specifically, it first simplifies G_q and then topologically sorts the operations in G_q (steps 2–3) as described above. It then evaluates G_q following the ordering (steps 4–6), reducing each operation by invoking a procedure `reduce`, which will be given shortly. This graph reduction process yields an instantiation I of the query, which is associated with the single node that has resulted from graph reduction, and consists of a sequence of value tuples for the variable tuple of the result skeleton. With each tuple t in I the result skeleton of Q is expanded to increment the skeleton S' of the output tree, sharing subtrees whenever possible (steps 7–9). Note that compression is conducted *stepwise for each tuple* t, instead of first generating the entire result tree and then compressing it. This leads to substantial reduction in decompression of T. Similarly, the data vector V' of the query result is populated with each t (step 10). These are conducted by procedures `expand` and `populate` (due to the space constraint we defer the details of these procedures and the full treatment of stepwise compression to the full version of the paper). The evaluation process always terminates since a query graph is a DAG.

Example 4.1: Given the query (G_q,G_r) of Figs. 3(c) and 3(d) and the vectorized XML tree (S,V) of Figs. 2(a) and 2(b), Algorithm `eval` first sorts the operations of G_q:

```
/bib(doc,$d), book($d,$b), publisher($b, 'SBP'),
author($b,_), article($d,$a), author($a,_),
eq($b/author,$a/author).
```

Here `author($b,_)` only detects whether or not $\$b$ has an author, and '_' indicates an unnamed variable which is not

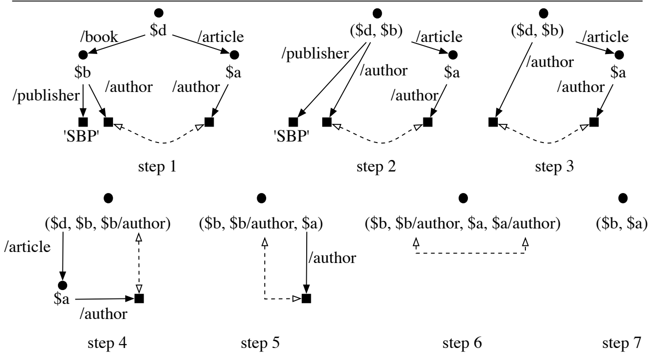


Figure 5. Reduction steps in Example 4.1

\$d	\$b	card
0	1	(2)

(a) inst(/bib/\$d/\$b)

\$d	\$b	value	card
0	1	RH	(2)

(b) inst(/bib/\$d/\$b/author)

Figure 6. Sample extended vectors

instantiated. Based on relational optimization heuristics, `publisher(`$\$b$`, 'SBP')` is scheduled before the equality test `eq(`$\$b/author$`,`$\$a/author$`)`. Given this ordering the operations are then conducted, reducing G_q into a single node $(\$b, \$a)$ in seven steps, as depicted in Fig. 5 (the details of the reduction steps will be seen shortly). When the reduction process is completed, the instantiation of the query is available as ⟨(`Curation`, `XStore`), (`Curation`, `XPath`), (`XML`, `XStore`), (`XML`, `XPath`)⟩. With each value tuple in the sequence, the algorithm expands the result skeleton and data vectors of the output tree. Finally the algorithm returns the vectorized tree shown in Figs. 3(a) and 3(b). □

Algorithm `eval` has several salient features. First, as will be seen shortly from the procedure `reduce`, it exploits lazy evaluation: data vectors are scanned only when they are needed; one does not have to load the entire XML document into memory. Second, its graph reduction strategy allows us to scan necessary data vectors once for each operation (and may further reduce scanning by grouping multiple operations using the same vector). Third, it allows seamless combination with relational algebraic optimization techniques. Fourth, with stepwise compression it avoids unnecessary decompression of the input vectorized XML tree.

4.2 Graph Reduction

We next focus on graph reduction and provide more details about the procedure `reduce`. The procedure processes an operation $op(v_1, v_2)$ in a query graph, where op is either an XPath expression p (for projection, selection) or eq (for equality/join), and v_1, v_2 are either nodes in G_q for projection and selection, or path terms $\$y/p_1, \$x/p_2$ for join eq. If v_2 is a variable $\$x$, its instantiation, *i.e.,* a *sequence* of nodes (or values) in VEC(T), is computed by `reduce` ($op(v_1, v_2)$); the instantiation is denoted by `inst(`$\$x$`)`.

The key challenge for graph reduction is how to correctly combine individual variable instantiations in order to produce the final value-*tuple* instantiation I. To do so, we extend each variable instantiation `inst(`$\$x$`)` by including paths from the document root to the document nodes in VEC(T) that are mapped to $\$x$. More specifically, w.l.o.g. we assume that each node n in the skeleton VEC(T) has a unique id, denoted by $nid(n)$, as shown in Fig. 2(a). For each n in `inst(`$\$x$`)`, where n is either a node in the skeleton of VEC(T) or a value in a data vector of VEC(T), we encode n with an *extended vector*, which is essentially a path in VEC(T) consisting of not only $nid(n)$ (or n for a text value n), but also the ids that are mapped to the ancestors of $\$x$ in G_q. Now `inst(`$\$x$`)` is a bag consisting of extended vectors instead of nodes/values. Referring to Example 4.1, at step 4 of the reduction, `inst(`$\$b$`)` and `inst(`$\$b$`/author)` consist of extended vectors given in Fig. 6, in which *card* indicates the cardinility of each extended vector. As will be seen shortly, extended vectors allow us to generate value tuples for the result skeleton while preserving the semantics of the query.

Observe the following. First, extended vectors are defined for nodes in a *query graph* in contrast to data vectors for values in an XML data tree. Second, extended vectors are computed during query evaluation (graph reduction); initially only the instance of the root of the query graph is available, which is the id of the root of the XML data tree.

We now move on to procedure `reduce` (G_q, e), which evaluates the operation e over a vectorized tree VEC(T) = (S, V). Consider the following cases of $e = op(v_1, v_2)$.

Projection $p(\$y, \$x)$**.** The procedure does the following: (1) compute `inst(`$\$x$`)`; (2) filter `inst(`$\$y$`)` *w.r.t.* `inst(`$\$x$`)`; (3) merge $\$y$ and $\$x$ into a new node $(\$y, \$x)$; and (4) modify G_q in response to the merging.

First, the instantiation `inst(`$\$x$`)` is computed by traversing the skeleton S of VEC(T), following the path p and starting from $\$y$ elements in `inst(`$\$y$`)`; it may also scan the data vector from V identified by the path p'/p, where p' is the path from document root to the instantiated $\$y$ elements, if p'/p leads to text nodes. The extended vectors are created for `inst(`$\$x$`)` by concatenating the extended vectors of `inst(`$\$y$`)` and the nodes/values mapped to $\$x$ during the evaluation. Note that by the variable dependency embedded in the topological ordering, `inst(`$\$y$`)` must be available when the operation $p(\$y, \$x)$ is ready to be evaluated. It should be remarked that the evaluation is *lazy*: only the needed data vector is scanned, and it is scanned once for computing the entire `inst(`$\$x$`)`.

Second, we decrease the cardinality of those extended vectors in `inst(`$\$y$`)` (and remove them if their cardinality is 0) that are not a prefix of any vectors in `inst(`$\$x$`)`, *i.e.,* removing those $\$y$ elements that do not have a p descendant. Note that this is not an issue for relational projection: relational data is regular and thus there is no need to check the existence of columns. For XML– typically semistructured – this is not only necessary for the correctness of query evaluation, but also reduces the evaluation cost. We denote this process as `filter` (`inst(`$\$y$`)`,`inst(`$\$x$`)`).

Third, $\$y$ and $\$x$ are merged into a single node $v = (\$y, \$x)$, which carries the instantiations (`inst(`$\$y$`)`, `inst(`$\$x$`)`) with it. In general, during graph reduction, a node in G_q is labeled with (X, I), where X is a sequence of nodes in G_q and I is their corresponding instantiations. In a nutshell, X contains (1) variables that are in the variable tuple of the result skeleton; or (2) nodes that indicate unprocessed operations. We denote this as `merge` ($\$y, \$x$).

Fourth, the query graph is modified: the new node v is inserted into G_q, the nodes $\$y, \$x$ are removed from G_q, and edges to/from $\$y, \$x$ are redirected to be to/from v. We refer to this process as `modify` (G_q, v).

Examples of projection processing include reduction steps 2, 4, 5 and 6 of Fig. 5. Note that $\$d$ is dropped from the root node at step 5 since it no longer has outgoing edges (*i.e.,* unprocessed operations) to the rest of the query graph.

Selection $p(\$y, c)$**.** The procedure (1) computes `inst(`c`)`, and (2) filters `inst(`$\$y$`)` *w.r.t.* `inst(`$c$`)` to remove/adjust those extended vectors that do not have a c descendant reachable via p. Note that all this selection operation does is to filter `inst(`$\$y$`)`. The constant c is removed from G_q if it no longer has unprocessed incoming edges.

For example, the step 3 of Fig. 5 filters `inst(`$\$b$`)` by decrementing the cardinality of the extended vector $(0, 1)$ from 3 to 2 since one `book` node has no "`SPB`" `publisher`.

Join $eq(\$y/p_1, \$x/p_2)$**.** Again by the variable dependency in the topological ordering, when this operation is processed both `inst(`$\$y$`)` and `inst(`$\$x$`)` are available. This operation is processed as follows: (1) compute projections $p_1(\$y, \$y_1)$ and $p_2(\$x, \$x_1)$, as well as instantiations of new variables `inst(`$\$y_1$`)` and `inst(`$\$x_1$`)`; (2) compute join of `inst(`$\$y_1$`)` and `inst(`$\$x_1$`)`, and (3) filter `inst(`$\$y$`)` and `inst(`$\$x$`)` by adjusting/removing those extended vectors that do not participate in the join. Note that the join result is not materialized. The join is used as a predicate to reduce the cardinality of participating instantiations `inst(`$\$y$`)` and `inst(`$\$x$`)`.

For example, the step 7 of Fig. 5 removes the extended vector $(0, 2)$ from `inst(`$\$a$`)` (not shown in Fig. 5) since it does not participate in the join with `inst(`$\$b$`/author)`.

Value tuples for the result skeleton. Finally, when the query graph G_q is reduced to a single node v, we need to generate value tuples for the variable tuple of the result skeleton. Observe that the node v now carries (X, I), where X is the sequence of all the variables in the result skeleton, and I consists of their corresponding instantiations. For example, after step 7 in Fig. 5, the single node in G_q carries (X, I), where X is $(\$b, \$a)$ and I consists of

`inst(`$\$b$`)`: (0, 1, *card*=2)   `inst(`$\$a$`)`: (0, 3, *card*=2)

To generate value *tuples* for the result skeleton, we need to group these individual instantiations. This is where we need extended vectors: grouping is conducted based on the lowest common ancestor of the participating variables. For our example above, we will merge `inst(`$\$b$`)` and `inst(`$\$a$`)` on the identifier of their ancestor, namely, $\$d$. It is fairly simple for this example since there is only one $\$d$ node; but the usefulness of extended vectors is evident for more complicated cases. The grouping for this example yields the query instantiation $\langle(1, 3)\rangle$ with *card* = 4. Note that ancestor ids are dropped now since they are no longer needed. This instantiation is passed to the result skeleton, which extracts `titles` of these nodes and obtains the query result: $\langle$(`Curation`, `XStore`), (`Curation`, `XPath`), (`XML`, `XStore`), (`XML`, `XPath`)$\rangle$. The process of generating value tuples from extended vectors I is referred to as `group` (I).

Putting these together, we outline procedure `reduce` in Fig. 7, which operates on a vectorized XML tree VEC(T). For the lack of space we omit the details of `group`, `merge`, `modify` and `filter`, which have been described above.

```
Input:  the query graph $G_q$ of an XQ query $Q$,
        and an operation $op(v_1, v_2)$ in $G_q$.
Output: the reduced $G_q$; when $G_q$ has a single node, it returns
        the instantiation $I$ of the query, consisting of
        value tuples for the result skeleton of $Q$.
1.  case the operation $e = op(v_1, v_2)$ of
2.  (1) projection $p(\$y, \$x)$:
3.        compute inst($\$x$) using inst($\$y$) and VEC($T$);
4.        inst($\$y$) := filter (inst($\$y$), inst($\$x$));
5.        $v$ := merge ($\$y, \$x$);
6.        $G_q$ := modify ($G_q, v$);
7.  (2) selection $p(\$y, c)$:
8.        compute inst($c$) using inst($\$y$) and VEC($T$);
9.        inst($\$y$) := filter (inst($\$y$), inst($c$));
10.       $G_q$ := $G_q$ with $c$ and its incoming edges removed;
11. (3) join $eq(\$y/\rho_1, \$x/\rho_2)$:
12.       compute projecttions $\rho_1(\$y, \$y_1)$ $\rho_2(\$x, \$x_1)$ and
          instantiations inst($\$y_1$) and inst($\$x_1$);
13.       $temp$ := join of inst($\$y_1$) and inst($\$x_1$);
14.       inst($\$y$) := filter (inst($\$y$), $temp$);
15.       inst($\$x$) := filter (inst($\$x$), $temp$);
16.       $G_q$ := $G_q$ with the equality edge removed;
17. if $G_q$ has a single node carrying ($X$, $I$)
18. then return group ($I$);
19. return $G_q$;
```

Figure 7. Procedure `reduce`

5 Experimental Study

We next present a preliminary experimental evaluation of our framework. We focus on query evaluation as the compression aspects of our work have been addressed in [9, 20].

We implemented the vectorization scheme (VX) on top of the Shore [11] storage manager. Each vector was stored as a separate clustered file. The hardware we used for our exper-

Dataset	XML Size	# Nodes	# Skel. Nodes	# Skel. Edges	# of Vectors	Vectors' Size
XMark (S.F. = 1)/(S.F. = 10) (*XK*)	111MB/1.2GB	1.7M/16.7M	73K/163K	381K/1.4M	410/410	79MB/782MB
Penn Treebank (*TB*)	54MB	7.1M	475K	1.3M	221,545	7.1MB
MedLine (*ML*)	1.5GB	36M	586K	5.8M	75	627MB
SkyServer (*SS*)	80GB	5.2G	372	371	368	29GB

Table 1. Description of the datasets used in our experiments

iments was a Linux box running RedHat 9. The CPU was a 1.8Ghz Pentium 4, while the system had 2GB of physical memory. The disk we used was a 200GB HDD; the operating system was on a separate disk. The raw disk speed, as measured by Linux's `hdparam`, was 32.5 MB/sec. We also installed two additional systems for comparison: the Galax [16] (GX) XQuery interpreter, which is a main memory implementation of XQuery, and Berkeley DB XML [5] (BDB), which is a native XML document store while, at the same time, provides XPath 1.0 querying functionality. We used the optimized GX executable and turned off type checking to obtain better performance. For every query evaluated on BDB, the appropriate index was built on the retrieved path. We used a buffer pool size of 1GB for Shore. GX could use all available memory.

We experimented with four datasets: the XML benchmark XMark [24] (*XK*), the Penn TreeBank (*TB*) natural language processing dataset, the MedLine (*ML*) biological dataset, and the SkyServer (*SS*) astronomical dataset. The datasets and their properties are summarized in Table 1. The XK dataset is a recognized XML benchmark; we chose the remaining three datasets to point out different aspects of our framework. The TB dataset has a highly irregular structure. Although the smallest in terms of raw XML size, it is decomposed into 221,545 data vectors. The SS dataset is a highly regular dataset. Though the largest in size, it is decomposed into only 368 vectors. The ML dataset is somewhere in the middle. An interesting approximation of each dataset's complexity is the ratio between its number of nodes in the raw XML document and the number of skeleton nodes in its compressed representation. The lower the ratio, the higher the complexity. For instance, this ratio for TB is 15, while the ratios for ML and SS are 61 and $14 \cdot 10^6$ respectively.

We use numbers reported in [24] and [17] for the XK[2] and SS dataset comparisons. The systems used in these papers were the Monet [23] system[3] and an SQL Server setup, respectively. Although our setup will differ, we use it to provide a rough comparison between the frameworks.

The numbers we report are cold numbers. Each query was run five times; the average of those five runs is reported. We calculated that all timings are within 5% of the average value with 99% confidence. The queries are summarized in Table 2 (see Appendix A for the full list of queries). Not all systems were able to process all queries. For instance, TQ2 was a join query that BDB could not process simply because

[2]The reported numbers are for an XMark scaling factor of 1.

[3]Available at `http://monetdb.cwi.nl`.

Query	Dataset	Failing system (reason)
KQ1	XK	—
KQ2	XK	BDB (No XQuery support)
KQ3	XK	Same as above
KQ4	XK	—
TQ1	TB	—
TQ2	TB	BDB (No XQuery support)
TQ3	TB	same as above
MQ1	ML	GX (OoM)
MQ2	ML	BDB (No XQuery support), GX (OoM)
SQ1	SS	BDB (Could not load doc.), GX (OoM)
SQ2	SS	Same as above
SQ3	SS	Same as above
SQ4	SS	Same as above

Table 2. Experimentation query workload; 'OoM' = out of memory

XQuery functionality is not available for that system. We provide the reasons for system failures.

A simple scalability experiment is described in Fig. 8. The x-axis shows the XMark scaling factor for the XK dataset; the y-axis the query evaluation time. VX scales linearly as the input size increases. Intuitively, this makes sense: a linear increase in document size and, hence, in the cardinality of the relevant vector(s) leads to a linear increase in query evaluation time. The cumulative results across all thirteen queries of the workload are presented in Table 3. A shaded cell denotes that the system failed to process the query for the reasons explained in Table 2. 'N/A' denotes that we could not use the system for the query.

In [24] what is called an association-based mapping is used. XML parent-child relationships are captured in binary relations. A dataguide-like structure is then computed so that groupings of binary relations into paths are tracked. The result is that for each path in the query a single table will be scanned. Queries KQ1, KQ2 and KQ3 are all value-based filtering queries that do not return any complex XML; by taking advantage of the "dataguide" each query is reduced to the equivalent relational operation on binary tables. In contrast, VX always has to scan and navigate the skeleton in addition to any data vectors. To be complete, we have put in parentheses the time spent by VX performing the value-based evaluation. Though the evaluation time for these queries is always comparable to Monet's, it can be seen that the bulk of VX's processing is in skeleton navigation. Path indexes on the skeleton present an interesting extension of our system. In Query KQ4, on the other hand, the dominance of VX is evident. Query KQ4 is an entire XML subtree retrieval with a complex navigational component, which cannot be mapped to a single table scan in Monet;

	KQ1	KQ2	KQ3	KQ4	TQ1	TQ2	TQ3	MQ1	MQ2	SQ1	SQ2	SQ3	SQ4
VX	4.4 (1)	9.8 (1.5)	9.4 (1.1)	9.2 (4.5)	9.6	139.4	158.1	133.4	385	36.9	61.4	32.7	30
BDB	83.9			47.7	51.2			6005					
GX	894	> 50000	> 50000	671	445.3	2870	2594						
Monet	0.2	8.7	7.5	1500	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
SQL Server	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	248.5	40.5	1.5	170.3

Table 3. The timing results for the queries of Table 2; elapsed time is measured in seconds

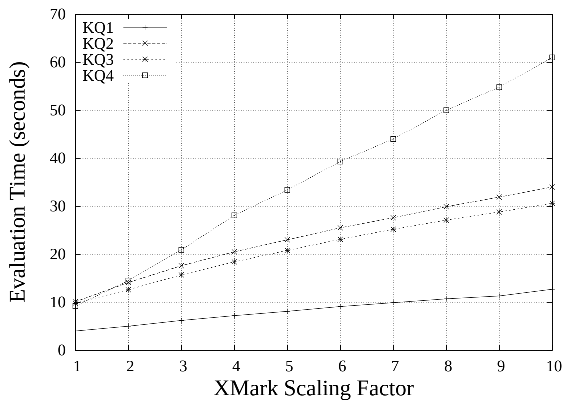


Figure 8. Scalability results

a reconstruction penalty has to be paid. As a consequence, VX outperforms Monet by almost 2.5 orders of magnitude.

GX is a native XQuery processor; however, it has to load the entire document before processing it. VX, on the other hand, accessed only the relevant vectors to evaluate the query. That gives it superior performance for the queries that GX could evaluate, even if the document loading time is ignored. For instance, the loading time of the TB dataset in GX was 439 seconds; even if that time is subtracted GX is still outperformed by VX.

A robust storage manager like BDB was not able to even load documents whose textual representation was significantly smaller than available memory, which is what BDB requires to have at load-time. In order to gather performance results we "chunked" each dataset and inserted it into the same BDB *container*. BDB was then able to perform XPath queries – after having built its special index structures. In all cases, however, it was significantly outperformed by VX.

The real "win" for VX comes when it is compared to a commercial relational database. We have to note here that in [17] SQL Server was rigorously tuned for the SS dataset. In Queries SQ1 and SQ4 VX outperforms SQL Server by almost a factor of six. In Query SQ2 VX is outperformed by SQL Server though performance remains comparable. The performance of VX, however, is not always dominant or even comparable to that of a commercial system; the reason is that we do not leverage all relational evaluation techniques – indexing in particular. Query SQ3 is a join between two relational tables. In [17] an index over one of the join attributes is built and index-nested loops is employed as the evaluation algorithm. The join predicate is highly selective returning only a small portion of the inner relation so the join is evaluated fast. The lack of indexing in VX means that both vectors need to be scanned. There is nothing that prevents efficient vector indexes to be incorporated into our system, and this is one of the enhancements we are currently investigating.

6 Concluding Remarks

We have proposed a new technique, vectorization, for building a native XML store over which a practical subset of XQuery can be evaluated efficiently using graph reduction and established relational database techniques. Our preliminary experimental results indicate that this method provides an effective approach to storing and querying substantial XML data repositories.

There is a host of work on using a RDBMS to store and query XML data (*e.g.,* [7, 15, 26]). Along the same lines [23] encodes parent-child edges (associations) in a binary relation, and it maps XQuery to OQL. The key challenge to the so-called "colonial" approach is how to convert XML queries to SQL queries [27]. Furthermore, most of the colonial systems ignore the order of XML data (one exception is [28]), which is often critical to the semantics of the XML data. Our work differs from the colonial approach in that we do not require the availability of the relational infrastructure, and thus XQuery-to-SQL translation is not an issue; in addition, vectorization preserves the order of XML data.

There has also been recent work on native XML systems (*e.g.,* [21, 22, 25]). These systems typically support text search and value filtering only, and they adopt vastly different representations for the structure and the text values of XML data. As a result their query evaluator has to "switch" processing paradigms as it moves from the realm of trees to the realm of text values; this, at times, poses a high overhead. In contrast, our system supports a uniform interface for querying both the structure and data values.

We have remarked in Sect. 1 on the connection between this work, XML compression [20] and skeleton compression [9]. Skeletons in [20] were compressed, but not in a form suitable for query evalation; in [9] the skeleton was expanded to represent the results of XPath evaluation. By contrast the query evaluation technique we have developed here yields new, usually smaller, skeletons to represent the result of XQuery evaluation.

There is an analogy between our graph reduction strategy and top-down datalog evaluation technique (in particular, QSQ [1]). The major difference is that our technique is for evaluating XQuery over vectorized XML data, whereas QSQ is for datalog queries over relational data. We plan to improve our evaluation strategy by incorporating datalog techniques such as magic sets [1]. Finally, the graph reduction technique we have described here differs fundamentally from graph reduction used in functional programming [18] in that with each reduction step, associated data – which is not manifest in the graph – is also evaluated.

There is naturally much more to be done. First, we have not capitalized on all the technology that is present in relational query optimization. For example, we currently make no use of indexing, and there is no reason why we cannot use it with the same effect as it is used in relational systems. It may also be that we can incorporate limited vector compression as suggested in [3] to further reduce I/O costs. Second, there are interesting techniques for further decomposition of the skeleton and making use of both relational and vector operations for exploiting this decomposition. Third, we intend to extend our graph-reduction technique to larger XQuery classes. Fourth, it is certain that we can exploit base type information (XML Schema [29]) and leverage structural and integrity constraints to develop better compression. Finally, we recognize the challenges introduced by updating vectorized XML data, and we are currently studying incremental [6] and versioning techniques for efficient maintenance of vectorized data. It should be mentioned that vectorization may simplify schema evolution, *e.g.*, adding/removing a column.

Acknowledgements The authors would like to thank Christoph Koch for allowing us to use his skeleton construction and XPath evaluation code from [9] and the anonymous referees for their comments. This work was supported in part by EPSRC grant GR/S13194/01 and a Royal Society Wolfson Merit Award.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] A. Ailamaki et al. Weaving Relations for Cache Performance. In *VLDB*, 2001.

[3] A. Arion et al. Efficient query evaluation over compressed XML data. In *EDBT*, 2004.

[4] D. S. Batory. On searching transposed files. *TODS*, 4(4):531–544, 1979.

[5] Berkeley DB XML v1.2, 2004. `http://www.sleepycat.com/products/xml.shtml`.

[6] P. Bohannon, B. Choi, and W. Fan. Incremental evaluation of schema-directed XML publishing. In *SIGMOD*, 2004.

[7] P. Bohannon et al. From XML Schema to Relations: A Cost-Based Approach to XML Storage. In *ICDE*, 2002.

[8] P. A. Boncz, A. N. Wilschut, and M. L. Kersten. Flattening an object algebra to provide performance. In *ICDE*, 1998.

[9] P. Buneman, M. Grohe, and C. Koch. Path Queries on Compressed XML. In *VLDB*, 2003.

[10] P. Buneman et al. Data integration in vector (vertically partitioned) databases. *IEEE Data Eng. Bull.*, 25(3):19–25, 2002.

[11] M. Carey et al. Shoring up persistent applications. In *SIGMOD*, 1994.

[12] D. Chamberlin et al. XQuery 1.0: An XML Query Language. W3C Working Draft, June 2001.

[13] J. Clark and S. DeRose. XML Path Language (XPath). W3C Working Draft, Nov. 1999.

[14] G. P. Copeland and S. Khoshafian. A decomposition storage model. In S. B. Navathe, editor, *SIGMOD*, 1985.

[15] D. Florescu and D. Kossmann. Storing and Querying XML Data using an RDMBS. *IEEE Data Eng. Bull.* , 22(3):27–34, 1999.

[16] Galax: An implementation of XQuery, 2003. `http://db.bell-labs.com/galax/`.

[17] J. Gray et al. Data mining the SDSS Skyserver database. Technical Report MSR-TR-2002-01, Microsoft, 2002.

[18] S. L. P. Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[19] P. Krneta. A new data warehousing paradigm for user and data scalability. Technical report, Sybase, 2000.

[20] H. Liefke and D. Suciu. XMILL: An Efficient Compressor for XML Data. In *SIGMOD*, 2000.

[21] J. F. Naughton et al. The Niagara Internet Query System. *IEEE Data Eng. Bull.*, 24(2):27–33, 2001.

[22] S. Paparizos et al. TIMBER: A Native System for Querying XML. In *SIGMOD*, 2003.

[23] A. Schmidt et al. Efficient relational storage and retrieval of XML documents. In *WebDB*, 2002.

[24] A. Schmidt et al. XMark: A benchmark for XML data management. In *VLDB*, 2002.

[25] H. Schöning and J. Wäsch. Tamino - An Internet Database System. In *EDBT*, 2000.

[26] J. Shanmugasundaram et al. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB*, 1999.

[27] J. Shanmugasundaram et al. Querying XML Views of Relational Data. In *VLDB*, 2001.

[28] I. Tatarinov et al. Storing and querying ordered XML using a relational database system. In *SIGMOD*, 2002.

[29] H. Thompson et al. XML Schema. W3C Working Draft, May 2001. `http://www.w3.org/XML/Schema`.

[30] J. D. Ullman. *Database and Knowledge Base Systems*. Computer Science Press, 1988.

[31] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. of Information Theory*, 23(3):337–349, 1977.

A Queries of the Test Suite

Queries KQ1,KQ2, KQ3, KQ4 rename Q5, Q11, Q12 and Q13 in [24], while SQ1, SQ2, SQ3, SQ4 are Q3, Q6, SX6 and SX13 in [17], respectively. The remaining queries are:

```
TQ1: /alltreebank/FILE/EMPTY/S/NP[JJ='Federal']
TQ2: for $s in /alltreebank/FILE/EMPTY/S
       for $nn in $s//NN
         for $vb in $s//VB
     where $nn = $vb return $s
TQ3: for $s in /alltreebank/FILE/EMPTY/S
       for $nn1 in $s/NP/NN
         for $nn2 in $s//WHNP/NP/NN
     where $nn1 = $nn2 return $s
MQ1: /MedlineCitationSets/MedlineCitation/
       [Language = "dut"][PubData/Year = 1999]
MQ2: for $x in /MedlineCitationSet/MedlineCitation
          $y in /MedlineCitationSet/MedlineCitation/
                 CommentCorrection/CommentOn
       where $x/PMID = $y/PMID return $x/MedlineID
```