Document Decomposition for XML Compression: A Heuristic Approach

Byron Choi

Nanyang Technological University
kkchoi@ntu.edu.sg

Abstract. Sharing of common subtrees has been reported useful not only for XML compression but also for main-memory XML query processing. This method compresses subtrees only when they exhibit identical structure. Even slight irregularities among subtrees dramatically reduce the performance of compression algorithms of this kind. Furthermore, when XML documents are large, the chance of having large number of identical subtrees is inherently low. In this paper, we proposed a method of decomposing XML documents for better compression. We proposed a heuristic method of locating minor irregularities in XML documents. The irregularities are then projected out from the original XML document. We refered this process to as *document decomposition*. We demonstrated that better compression can be achieved by compressing the decomposed documents separately. Experimental results demonstrated that the *compressed skeletons*, for *all* real-world datasets, to our knowledge, fit comfortably into main memory of commodity computers nowadays. Preliminary results on querying compressed skeletons validate the effectiveness our approach.

1 Introduction

XML has been the *defacto* standard for data exchange on the web. While XML has been useful for passing small messages between heterogeneous applications [15], XML has also been used to represent large amount of data [19, 12, 8, 13]. However, a major drawback of this use of XML is the increase in the size of data, due to the verboseness of XML syntax. What is desirable is an efficient compression technique for XML.

The main reason for storing data as XML is that (part of) the documents may need to be queried efficiently later. The two kinds of compression techniques, "syntactic" and "semantic", perform differently regarding query processing. "Syntactic" compression (*e.g.,* [22]) treats data as a sequence of bytes. While syntactic compression often produces good compression performance on a wide range of datasets, the semantics of data is often lost during compression. This often reduces query performance on the compressed data. An alternative is to derive a "semantic" compression technique, *e.g.,* [14, 4]. Typically, such technique compresses data based on its semantics. The semantics embedded in the compressed data has been reported useful for query evaluation [4, 3]. In this paper, we shall focus on semantic compression. It should also be remarked [10] that applying semantic compression followed by syntactic compression often results in better overall compression and query performance.

At the core of the "semantic" XML compression technique [4], it is a procedure of compressing/sharing identical subtrees. Its performance depends on the number and the

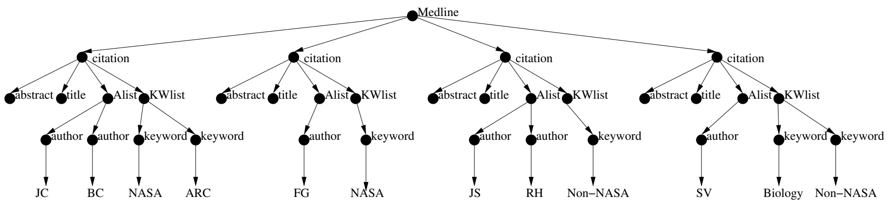

Fig. 1. An XML tree T

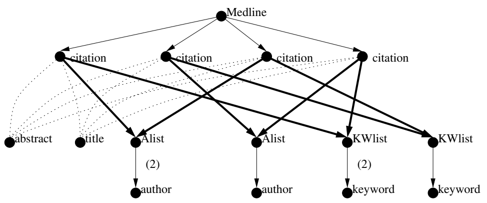

Fig. 2. Skeleton of T, G

size of identical subtrees being shared. However, when XML documents are large, the chance of having large identical subtrees is *inherently* low. Unfortunately, in practice, we encounter a case where the compressed instance produced by [4] is larger than the size of the memory of a commodity computer nowadays. Worse still, query evaluation on compressed XML [4, 3] assumed the compressed instance was stored in main memory. This necessitates further investigations on XML compression techniques.

To illustrate the problem studied in this paper, we present a real-world example and show the result of our proposed solution. Consider the simplified MEDLINE bibliography dataset [19] shown in Figure 1. A MEDLINE document contains a large number of citations, although four citations are shown in the example. Each citation has an abstract, a title, a list of authors and a list of keywords, among other things. We shall illustrate the compression presented in [4, 3] with this document.

Consider a depth-first traversal on the document. Suppose that during this traversal we also generate a tree in which each of the text nodes is replaced by a marker (#) indicating the presence of text nodes in the original document. We refer this tree to as the *skeleton* of the document. Consider the first two author (`author`) nodes. Once we have replaced the text nodes by the markers, these nodes exhibit identical structure. Therefore we can replace them by a single structure and put multiple edges from the `citation/Alist` node on top of the `author` node. Moreover, since these `Alist-author` 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 compress the skeleton into a DAG as shown in Figure 2. Multiple consecutive edges are indicated by an annotation (n), and an edge without annotation occurs once (in the DAG). This technique has been known as *subtree-sharing* [4, 3].

The edges in the middle of Figure 2 illustrate the reason of inefficient subtree-sharing. The `citation` nodes are not compressed because each `citation` node has a slightly different author list and keyword list. In fact, the identical sub-structures in

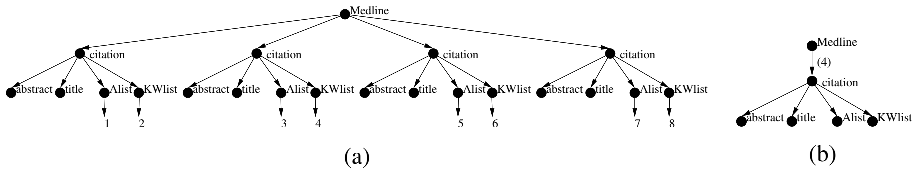

Fig. 3. (a) The reduced document of Figure 1 and (b) its compressed skeleton

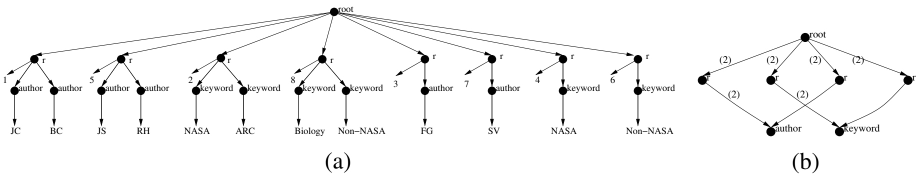

Fig. 4. (a) The outlier document of Figure 1 and (b) its compressed skeleton

`citation` dominate the others. Since the `citation` nodes are not compressed, the dashed edges are also necessary.

In this paper, we proposed an improved compression algorithm for XML. Our technique is inspired by *projected clustering techniques* for data mining applications (see Section 4). Our technique projects out the subtrees which stop [3] from compressing an input XML. The projected subtrees are grouped in an *outlier* document. The remaining document formed the *reduced* document. We call this process *document decomposition*.

Let us return to the simplified MEDLINE document. Suppose that we decompose the document at P, where P = {`/Medline/citation/Alist`,`/Medline/citation/KWlist`}. It means that we shall project out the subtrees underneath P and group them in an outlier document. The reduced document and the outlier document are shown in Figure 3 (a) and Figure 4 (a), respectively. We shall compress the reduced and the outlier documents by using [3] individually. The compressed skeletons of the reduced and the outlier documents are presented in Figure 3 (b) and Figure 4 (b), respectively. Note that the compressed skeletons contain neither the bold nor the dashed edges. The complicated edges in the middle of Figure 1 are encoded by data values. That is, they are no longer *embedded in* the compressed skeleton. Consequently, the decomposed documents can be compressed efficiently. However, we need to store these (uncompressed) edges on disk. Furthermore, there is a tradeoff between skeleton compression and query processing. Queries involving both the reduced and the outlier documents require extra joins to recover the relationship between the two documents. In this paper, we proposed a heuristic method to determine these edges.

The main contributions of this paper are listed as follows.

– We propose an algorithm for XML compression by decomposing an XML document into a reduced and an outlier document. The decomposition causes irregular subtrees to be grouped in the outlier document and leaves the subtrees remaining in the reduced document fairly similar. We noted that the decomposed-compressed skeletons of real-world XML documents fit in main memory comfortably.

- We proposed a query-rewriting algorithm for evaluating queries on the decomposed documents by leveraging existing query evaluation algorithms [3].
- We present experimental results on the effectiveness of the compression and preliminary experimental results on query evaluation on decomposed skeletons.

The remainder of this paper is structured as follows. Section 2 contains notations used and background information of this paper. Section 3 presents the representation, the construction and query evaluation of decomposed XML. Section 4 presents our heuristic algorithm for determining good decomposition. Section 5 shows an experimental study of our proposed algorithm. We discuss some related work in Section 6. Conclusions and future work are presented in Section 7.

2 Notations and Background

In this section, we list some notations used in the paper. We consider the compression algorithm VEC presented in [3] in this paper. Consider an XML document T. VEC(T) $\equiv$ (G, V), where G is the compressed skeleton of T and V is the representation for data nodes. A *cut* is the set of edges at where the decomposition occurs. We consider the cut to be specified by a set of simple downward paths P, which can also be considered as "projections" of subtrees. Thus, we may refer P to the cut. Suppose the DTD of T is available. The possible variations in subtree structure will be essentially [1] indicated by stars "*". Denote the set of stars in the DTD to be $\mathcal{S}$. For identifying irregularities, projections make sense at stars only. In addition, we assume that the cuts in T are not nested. Justifications shall be provided followed by the discussion of our solution in Section 4. Given a projection P, we decomposed an XML document into the reduced document T_r^P and the outlier document T_o^P. We may omit P from the notation if it is clear from the context. Similarly, we denote the decomposed, compressed document as DVEC: DVEC(T,P) $\equiv$ (VEC(T_r), VEC(T_o)) $\equiv$ ((G_r, V_r), (G_o, V_o)).

The main challenge of our problem is to determine P at which the document is decomposed. The search space of the problem is $O(2^{|\mathcal{S}|})$. This daunting complexity indicates that there is a need to develop heuristics for the problem. In addition, the number of subtrees in the document is $O(2^{|T|})$, where $|T|$ is the number of nodes in T.

3 Document Decomposition

Consider a projection P of an input document T determined by the heuristic proposed in Section 4. We discussed the construction of the compressed reduced and outlier documents in one scan of T in Section 3.1 and Section 3.2. Section 3.2 contained no new ideas but completed the discussions on compression VEC on our simplified MEDLINE document. In Section 3.3, we re-write queries on T into queries on T_r^P and T_o^P and discuss how [4, 3] is used for efficient query evaluation. Technique presented in this section can be applied recursively to support multiple decompositions.

[1] For simplicity, we skip the discussions on "?".

3.1 Construction of the Reduced and the Outlier Documents

In this subsection, we present an algorithm for producing the reduced document T_r and the outlier document T_o of a given P, shown in Figure 5. The algorithm consists of a single depth first traversal of the original document T. The construction of T_o and T_r and the compression algorithm VEC can be easily incorporated into a single traversal of T. We decoupled the discussions of the two for simplicity.

The details of the algorithm is as follows. During the traversal of the document, we maintained the parent n' of the current node n and the path p from the root to n. We use a boolean variable *top* to indicate whether the current node belongs to T_r and r_{last} to record the last consecutive subtrees crossing the "boundary" of a path in P. A counter $\#ordinal$ is used to record the number of cut edges encountered.

Initially, T_r and T_o are empty. Line 12-13 show the simplest case where the traversal does not cross the cut: if *top* is true (resp. false), we continue to construct T_r (resp. T_o). If the traversal crosses the boundary of the projection (Line 01-10), we modified T_r (Line 02-06) and T_o (Line 07-10) as follows. First, we remove the cut edge from T_r (Line 03). Denote $n.l$ as the tag of a node. If n does not form consecutive l nodes

```
Procedure decompose($T$, $P$)
Input: A doc. tree $T$ and a projection $P$
Output: $T_r$ and $T_o$
$T_r$ = empty; $T_o$ = empty; top = true; ordinal# = 0; $r_{last}$ = null
Depth first traversal on $T$:
On entry of a node $n$:
Denote $p$ to be the path from the root to $n$ and ($n'$,$n$) to be an edge in $T$
01 if path to $n' \in P$      //across the boundary
02   top = false
03   remove ($n'$,$n$) from $T_r$      //due to Line 14
04   if the last child of $n'$ is not an ordinal number
       //for the reduced doc.
05     append a new ordinal node w. ordinal# $o$ and the edge ($n'$, $o$) to $T_r$
06     ordinal#++; top = false
       //for the outlier doc.
07     merge_last_subtree($r_{last}$,$T_o$)
08     create $o'$ as a clone of $o$
09     create artificial nodes $r$; append $o'$ to olist of $r$; create an edge ($r$, $n$)
10     $r_{last}$ = $r$
     else
11     append $n$ and ($r_{last}$, $n$) to $T_o$
   else
12   if top == true     append $n$ and ($n'$, $n$) to $T_r$
13   else     append $n$ and ($n'$, $n$) to the $r_{last}$-subtree
On exit of a node $n$:
if $p \in P$ then top = true
```

Fig. 5. Construction of T_P^r and T_P^o, the decompose procedure

with previously visited children of n' (Line 04), we create a new ordinal (text) node o with a unique ordinal number $\#ordinal$ (Line 05) and append o to T_r. The construction of the outlier T_o involves grouping subtrees based on their structure. If the guard condition in Line 03 ensures that n does not form consecutive l-subtrees with r_{last}, this implies r_{last} has been completely traversed. We use the `merge_last_tree` procedure to append r_{last} to a group, in T_o, according to its structure. The grouping can be efficiently implemented by hashtables [4]. Then we create a new r node and append its corresponding ordinal number to r. r is set to be the new r_{last}-subtree. If the guard condition is satisfied, we continue to build the r_{last}-subtree (Line 11). The algorithm requires exactly one scan on T and maintains one r_{last}-subtree in main memory during the scan.

3.2 Compression of the Reduced and the Outlier Documents

The reduced and the outlier documents are yet another XML documents. Existing XML compression techniques can be directly applied to compress these documents. We resume our discussion on compressing the skeleton of XML [4]. Skeleton compression is also implemented in a depth first traversal of T. The implementation requires a main-memory hashtable of subtrees encountered during the traversal. On the exit of a node n, *i.e.,* the entire subtree rooted at n is traversed, we probe the hashtable and check if such a subtree (structure) is encountered before. If this is the case, we compress/share the subtree by adding a reference to the existing subtree in G, the compressed skeleton. Otherwise, we insert n into both G and the hashtable. For example, the outlier document shown in Figure 4 (a) is compressed to the structure shown in Figure 4 (b).

The data nodes are handled as follows. When a data node is encountered during the traversal, we *append* the data node to a container (vector) which is uniquely identified by the root-to-leaf path. For instance, at the end of the traversal, the data nodes in the outlier document shown in Figure 4 (a) are listed below.

```
/root/r/author: [JC, BC, WF, FG, JS, RH, SV]
/root/r/keyword: [NASA,ARC,NASA,ARC,Non-NASA,Biology,Non-NASA]
/root/r/@olist: [1, 5, 2, 8, 3, 7, 4, 6]
```

We shall discuss the implementation of the containers for ordinal numbers together with query processing in the next subsection. It should also be remarked that the compression algorithm can be readily incorporated into the `decompose` procedure. Neither the reduced document nor the outlier document is fully materialized.

3.3 Query Evaluation on Decomposed Documents

In this subsection, we illustrate how a query on a document is rewritten into a query on its decomposed documents. Subsequently, query evaluation on compressed XML [3] is reused for evaluating queries on decomposed documents.

Denote the query evaluation of [3] as $eval$. Consider a path query p, /e_1/e_2/.../e_n. The evaluation of p on VEC(T) are rewritten into a query on DVEC(T,P) as follows.

$$\begin{aligned}
&eval(p,\textsc{vec}(T))\\
&\quad\equiv eval(p, \textsc{dvec}\ (T, P))\\
&\quad\equiv eval(p, \textsc{dvec}.1)\\
&\qquad\cup\ eval(F(/e_1, \textsc{dvec}\ (T, P))/e_2/.../e_n, \textsc{dvec}.2),\\
&\qquad\cup\ eval(F(/e_1/e_2, \textsc{dvec}\ (T, P))/e_3/.../e_n, \textsc{dvec}.2), ...\\
&\qquad\cup\ eval(F(/e_1/e_2/.../e_{n-1}, \textsc{dvec}\ (T, P))/e_n, \textsc{dvec}.2),\\
&\qquad\qquad\qquad\text{where } F(p, \textsc{dvec}\ (T, P)) = \texttt{for}\ \$x\ \texttt{in}\ \textsc{dvec}.2/root\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{where}\ \$x/\texttt{@o} =\ eval(\textsc{dvec}.1,p/\texttt{text()})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{return}\ \$x/r\\
&\quad\equiv eval(p, \textsc{dvec}.1)\\
&\qquad\bigcup\nolimits_{1..n-1}\ eval(F(/e_1/../e_i, \textsc{dvec}\ (T, P))/e_{i+1}/.../e_n, \textsc{dvec}.2)
\end{aligned}$$

The rewritten query on the right hand side of the formula comprises two parts. The first part states that the result of $eval(p,\textsc{vec}(T))$ includes the results found in the reduced document, *i.e.,* DVEC.1 while the second part states that the result of $eval(p,\textsc{vec}(T))$ also includes the ones found in (1) evaluating $/e_1/e_2/.../e_i$ in DVEC.1 followed by (2) evaluating $/e_{i+1}/e_{i+2}/.../e_n$ on the outlier document, *i.e.,* DVEC.2. This requires joins, denoted as F, of the intermediate results from (1) and (2) on ordinal numbers, which recover cross edges between DVEC.1 and DVEC.2.

Implementation. The overhead introduced by the rewriting involves exactly joins on ordinal numbers and projections on $\$x/r$. The joins are often needed, *e.g.,* queries with descendant steps "//". Hence, it is desirable to pre-compute the joins as well as the projection in F. A clustered index is built on the result of the joins [20]. That is, we do not store the containers for ordinal numbers but the join result in F. Consequently, F are implemented as a scan on the index, as opposed to a few joins on-the-fly. Cost estimation techniques can be incorporated to further optimize the joins. We plan to incorporate these techniques into our method in future.

4 Heuristic Algorithm for Determining a Cut

In previous sections, we illustrated the idea of document decomposition and showed how decomposition may improve compression. The key of the problem is to determine a good cut P of an input document T. The pseudo-code of our algorithm for this issue is shown in Figure 6. The overall algorithm can be roughly divided into four phases. (1) We infer a "schema" $\mathcal{S}$ from the input document T. (2) As we construct $\mathcal{S}$, we construct histograms N to summarize the structural property of T (Line 01). We reduce the number of stars in $\mathcal{S}$ in this phase. (3) Based on the histograms on reduced $\mathcal{S}$ and our cost function, we use a simulated-annealing procedure (Line 02) to progressively search for a good cut. (4) Finally, we refine the solution obtained (Line 03).

Next, we present a detailed discussion on the four phases of our proposed solution. The meaning of the parameters in Figure 6 are discussed as we proceed.

Phase 1. Schema inference phase. As remarked earlier, the major variations of the structure are indicated by the stars in DTDs. We shall consider stars as "*structural dimensions*" of a subtree and subsequently represent a subtree as a data point in a multi-dimensional space. In this phase, we shall determine all possible stars in a document.

```
**Input:** $T$, an XML tree; $\theta_{sup}$ $\theta_H$, $\theta_C$, $\theta_S$, $K$
$\theta_{sup}$: the minimum support of major stars; $\theta_H$: the minimum entropy of major stars;
$\theta_C$: the weight of the query part of Formula 1; $\theta_S$: the weight of the storage part of Formula 1;
$K$: the number of scans used in the refinement phase
**Output:** $\mathcal{S}$: a set of stars where decomposition occurs,
01 ($\mathcal{S}$, $N$) = infer_major_stars($T$, $\theta_{sup}$, $\theta_H$)        //Phase 1 and 2
02 $\mathcal{S}$ = simulated_annealing($\mathcal{S}$, $N$, $\theta_C$, $\theta_S$)        //Phase 3
03 for $i$ from 0 to $K$                                          //Phase 4
       $N_{2^i}$ = recover_order($\mathcal{S}$, $2^i$)
       $\mathcal{S}$ = simulated_annealing($\mathcal{S}$, $N_{2^i}$, $\theta_C$, $\theta_S$)
04 return $\mathcal{S}$
```

Fig. 6. Algorithm `determine_cut(`T`)`

When the DTD of a document is present, we obtain the stars for free. Otherwise, we infer the probable stars from the document. First, we construct a prefix tree of the document. (The prefix tree will also be used in later phases.) A node in a prefix tree represents a prefix occurred in a document and is associated with the support, *sup*, of the prefix in the document. Second, we define a *support ratio* between each pair of parent-child nodes (A, B) to estimate the possible location of stars. There are three possible cases for the support ratio:

1. The support ratio is between 0 to 1. This implies B is probably A's optional child;
2. The support ratio is 1. This often implies a one-to-one relationship;
3. The support ratio is greater than one. This often indicates a one-to-many relationship. We regard the edges in this class as *star edges*.

There are exceptions of the above implications. Consider a pathological document in which half of the A nodes do not have B-child and half of the A nodes have exactly two B-children. The support ratio indicates a false one-to-one relationship. However, such exceptions are rare, in practice.

Example 1. We illustrate the support ratio with an example shown in Figure 7. The prefix tree is derived from the XML document shown in Figure 1. The support of the node is indicated inside the square bracket and the support ratio is indicated on the edge. We use a "*" to indicate the location of stars.

Phase 2. Initialization phase. A subtree can be readily summarized by a vector: each star is associated with an entry in the vector and the value of the entry is the number of repetitions of the star edge in the subtree. For example, consider again the `citation`

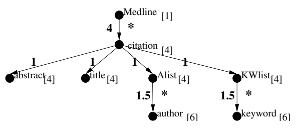

Fig. 7. The prefix tree of T in Figure 1

subtrees in the document shown in Figure 1. The vector of the subtrees are (2, 2), (1, 1), (2, 1) and (1, 2), respectively. Alternatively, subtrees can be viewed as data points in a structural-dimensional space.

Consider a depth first traversal on a given document T again. The vector of partially-traversed subtrees are kept in main memory which requires $O(d|\mathcal{S}|)$ space. Typically, the number of stars $|\mathcal{S}|$ in a prefix tree is small. However, large $|\mathcal{S}|$ causes problems: (1) Summary structures are built for each stars later; when $|\mathcal{S}|$ is large, large amount of memory is required; (2) A search in a high dimensional space is often inaccurate [2]. Unfortunately, we find a real-world case where $\mathcal{S}$ is large: The prefix tree of TREEBANK (linguistic dataset) contains thousands of stars. This motivates us to distinguish major and minor stars (dimensions) in the initialization phase. Subsequent search focuses on the major stars only. This phase consists of two methods.

The first method is to skip processing the stars with small support. A star with small support may lead to small impact on overall compression. Though simple, this method has been found effective. For example when we considered the minor stars to be the ones with a support smaller than 0.5% of the total number of edges in T, the method prunes more than 95% stars in the prefix tree of TREEBANK.

Another method involves computing the information content of a star (structural dimension). Specifically, we compute the entropy H of a (local) histogram N of a star s $\in \mathcal{S}$ as: -$\sum_{x \in B} p_x log(\frac{1}{p_x})$, where B is the set of bins in the histogram, each bin represents a class of subtrees, p_x is the probability of encountering x in N, where $x \in B$ and two s-subtrees belong to the same bin (class) if and only if they have the same number of outgoing s-edges. We build such histogram of each star in $\mathcal{S}$ in one scan of T and compute the entropy of such histograms at the end of the scan. Large entropy implies the corresponding (star) edges in T are inherently incompressible and are considered candidates of irregularities in T. The intuition is to project out these irregularities from T which may leave the reduced subtree more compression-friendly. On the contrary, in later phases, we skip the stars with an entropy smaller than a threshold.

Specifically, we use two parameters θ_{sup} and θ_H to specify the minimum support and entropy of a major star. Any star with a support (resp. entropy) smaller than θ_{sup} (resp. θ_H) is considered a *minor star*. We shall remove minor stars from $\mathcal{S}$ and pass a reduced $\mathcal{S}$ to the next phase for determining good cuts. We refer this process to as *reduction of structural dimensions of subtrees*.

We remark that the histograms constructed in this phase summarize local structural information only. This method is sound: The entropy of histograms with global information is at least as large as the one with local information. The reduction based on local information, though space-efficient, may exclude some globally optimal cuts.

Phase 3. Simulated-annealing phase. Similar to most data-mining algorithms, our algorithm consists of a simulated-annealing phase which progressively improves the quality of the solution. We represent a subtree as a vector/data point in the reduced dimensions. For each star, a histogram of reduced vectors is constructed. Our search finds a set of stars P_{cur} whose decomposition cost is minimized, in the reduced dimensions.

Initially, we randomly choose a P_{cur}. We assume that the stars in P_{cur} are not nested. This property is preserved as the search proceeds. (Nested stars in P_{cur} are nested cuts, which interact and cause a model inaccurate.) The simulated-annealing process is guided by the cost (*a.k.a.* energy) function defined in Formula 1 and 2.

$$energy(T,P) = \theta_C \times \sum_{s \in P \cup \{r\}} s.sup + \theta_S \times \sum_{s \in P \cup \{r\}} |s.N| \times s.sup \times f(s,P), \tag{1}$$

$$f(n,P) = \begin{cases} 1 & \text{if n} \neq \text{r} \\ \prod_{s \in P}(1 - f(s)) \;\; where & \\ \quad f(s) = \prod_{a \in A(s)} a.sup/S(a).sup & \\ \qquad where\ A(s) \ = s.ancestors\ and\ S(a) \ = \ a.siblings \cup \{a\} & \text{if n = r} \end{cases} \tag{2}$$

The cost function models the query cost and the storage cost of a cut P. The parameter θ_C and θ_S are used to model the relative importance of the query cost and the storage cost, respectively. Below describes the meaning of the formulae for these costs.

Query cost. The query cost is linearly proportional to the total number of edges across the cut. The reason is that when a query involves multiple decomposed skeletons, joins are required to reconstruct (part of) the skeletons. With modern join algorithms, the joins can be implemented with runtime linear to the number of edges across the cut. Hence, we have $\sum_{s \in P \cup \{r\}} s.\mathtt{sup}$ in Formula 1.

Storage cost. The storage cost models the size of the compressed skeletons after decomposition. Assume that the size of compressed skeleton is proportional to the number of structurally distinct subtrees in T. Furthermore, as we shall see in experiments, nested projections often lead to small advantages in compressions. Since such projections are typically hard to estimate accurately and indeed complicated our model, we assume nested projections are not allowed. Based on these assumptions, we define the storage cost as follows. (1) The space required to store s-subtrees is proportional to the size of the histogram of s $|s.N|$ and the number of s-subtrees $\mathtt{sup}$. Hence we have $\sum_{s \in P} |s.N| \times s.\mathtt{sup}$. (2) To model the size of the reduced document (*i.e.,* the r-subtree), we need to model the effect of projecting out P on the r-subtree. We define an additional function f for this purpose. Consider an edge (n_1, n_2) in a prefix tree. We assume the storage required to store n_2-subtrees is proportional to $f(n_2)$, the percentage of $n_2.\mathtt{sup}$ among all children of n_1, *i.e.,* $n_2.\mathtt{sup}/S(n_2).\mathtt{sup}$, where $S(n_2)$ is the siblings of n_2 together with n_2. We model the cost of storing n_1 after projecting out n_2-subtree to be 1 - $f(n_2)$. Since we want to compute the effect of projecting out s-subtrees on the root r, we "propagate" the effect to the root by multiplying the value of $f(a)$ for all a in the ancestors of s. Therefore, we yield Formula 2.

The two costs described above interact in a non-trivial manner: (1) A star s with a small depth often implies a small *sup* and a small query cost. (2) However, the number of structurally distinct s-subtrees, $|s.N|$, could be large. (3) Projecting s has proximate impact on the compression of r, modeled by $f(r, P)$. The reverse of the three conditions applies to stars with a large depth.

Phase 4. Refinement phase. In this phase, we handle the node order (Line 04 of Figure 6). The order of nodes may cause (1) *false negatives* when the entropy of N

is small but identical subtrees occur mainly alternately or (2) *false positive* when the entropy of N is large but consecutive identical subtrees are frequently found. Possible false positives/negatives can be detected by additional scans on T: Similar to string compression, we construct histograms of k-consecutive s-subtrees. The order of XML is recovered as the value of k increases. The stars with sharp increase (resp. decrease) in entropy as k increases are the candidates of false positives (resp. negatives).

Complexities. The construction of prefix tree and the initialization phase are implemented in one scan of T. The simulated-annealing phase requires a scan of T for building histograms in the reduced dimension. Depending on the importance of the ordered-ness in determining the cut for T, another K scans on T are needed in the refinement phase. Hence, the I/O cost of the algorithm is $(2 + K) \times |T|$.

5 Experimental Evaluation

We conducted an experimental evaluation on the proposed document decomposition and the heuristic algorithm. We focused mainly on the quality of the cuts returned by the heuristics presented in Section 4 and briefly studied query performance on decomposed documents. To evaluate the query performance on decomposed documents, we used the query modules in [3]. We have implemented a prototype of the heuristics and decomposition algorithm in C/C++. The prototype is run on a LINUX box running REDHAT 9.0. The CPU was 1.8GHz PENTIUM 4, while the system had 2GB of physical memory. We allowed the heuristics five tries and a maximum 100K search steps. We defined a variable I, ranges from 0 to 1, whose value is directly proportional to the maximum number of stars (paths) allowed in a cut. We considered the stars with the support less than 0.5% of the total number of edges in the document as minor stars. θ_C and θ_S are the weights of the query component and the storage component of Formula 1, respectively.

Experiments on different datasets. We have applied the heuristics/decomposition algorithm on a few XML datasets: the Penn TREEBANK linguistic dataset, the XML benchmark XMARK with scaling factor 1, the computer science bibliography dataset DBLP, Shakespeare plays in XML, protein dataset SWISSPROT, MEDLINE biological dataset, and the SKYSERVER astronomical dataset. I and θ_S/θ_C are 1. We summarized our results in Table 1. $T^{|V|}$, $G^{|V|}$ and, $G^{|V|}_{r,o}$ are the number of nodes in skeleton

Table 1. Compression result

Doc	$T^{\|V\|}$	$G^{\|V\|}$	$G^{\|E\|}$	$G^{\|V\|}_{r,o}$	$G^{\|E\|}_{r,o}$
TREEBANK	7.1M	475K	1.3M	475K+0K	1.3M+0M
XMARK	1.7M	73K	381K	15K+45K	44K+272K
DBLP	2.6M	4.4K	225K	1.0K+0.4	83K+1K
Shakespr.	180K	1.5K	32K	0.5K+0.5K	2.6K+2.2K
SWISSPROT	3M	59K	778K	2K+7K	33K+241K
ML (3 yr)	36M	586K	5.8M	9.5K+219K	324K+2.1M
ML (all)	NA	NA	NA	54K + 2.8M	6.9M + 66M
SKYSERVER	5.2G	372	371	372+0	371+0

without compression, compressed skeleton and decomposed-compressed skeletons, respectively. Similarly, we use $|E|$ to denote the number of edges in these three structures.

We begin our discussions with the simple cases. The results from TREEBANK and SKYSERVER show that document decomposition produces negligible or no improvement on compression. TREEBANK contains numerous linguistic trees, where each tree often exhibits a unique structure. Almost all stars in the prefix tree of TREEBANK are minor. Hence document decomposition does not yield more common subtrees, when it is compared to the one without. In contrast, SKYSERVER dataset encodes a large relation; its prefix tree contains one star. The heuristics correctly returns an empty cut.

For the remaining datasets except XMARK, the heuristics returned cuts which improved compression over *already compressed skeletons* by using five tries only. The number of nodes in decomposed skeletons ranges from 15% (SWISSPROT) to 66% (Shakespeare) of that of original compressed skeleton; And the number of the edges in decomposed skeletons is reduced to 15% (Shakespeare) to 41% (MEDLINE) of the original compressed skeleton. Furthermore, by decomposing (all) MEDLINE dataset (39G bytes), we can store its compressed skeletons in main memory of a commodity computer, which was impossible before.

When the heuristics is applied to XMARK, we observed that the heuristics hits false local maxima frequently. The reason can be illustrated with the example shown in Figure 8. Figure 8 (a) shows a simplified XMARK data, in which open and closed auctions contain lists of paragraphs, specifically $listpar$-subtrees. Common subtree-sharing does not perform efficiently on $listpar$-subtrees because there are many distinct paragraph structures in XMARK. Hence, we encountered the complicated edges shown in Figure 8 (b). The heuristics sometimes places $/XMark/closed\_auction$ (alone) into the cut because this would separate *some* problematic subtrees from the original document. However, after this decomposition, *both* documents contain the problematic subtrees (see Figure 8 (c)). To project out all $listpar$-subtrees from XMARK, a path like $//listpar$ is needed. Unfortunately, $listpar$ is recursive. This means $//listpar$ specifies nested cuts, which is not modeled by our formulas. Worst still, $listpar$-subtrees appear at a few places in XMARK's prefix tree which lead to many false local maxima in the search space. Since the current heuristics does not model correlation between stars, the search skips such local maxima by chance only.

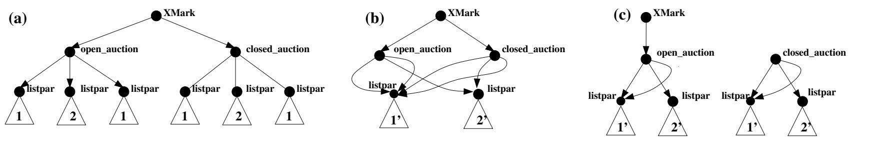

Fig. 8. Problematic case in XMARK: (a) sketch of XMARK; (b) compressed skeleton without decomposition; (c) compressed skeletons with decomposition

Experiments on parameters. We conducted another set of experiments to study the effects of some parameters of our method on XMARK and DBLP datasets. We reported the *average* of the local maxima returned by five tries of the heuristics. We fixed θ_C/θ_S to be 1 and varied the cut size by varying I. When I is 0, there is no decomposition.

I	0	0.2	0.4	0.6	0.8	1
$G_{r,o}^{\|V\|}$	73K	58K	66K	64K	58K	68K
$G_{r,o}^{\|E\|}$	381K	303K	339K	315K	300K	329K

Fig. 9. Dec. skeleton size vs cut size (XMark)

I	0	0.2	0.4	0.6	0.8	1
$G_{r,o}^{\|V\|}$	4.4K	2.1K	1.5K	2.0K	1.2K	1.6K
$G_{r,o}^{\|E\|}$	225K	153K	65K	130K	59K	134K

Fig. 10. Dec. skeleton size vs cut size (DBLP)

θ_C/θ_S	0.01	0.1	1	10	100
$G_{r,o}^{\|V\|}$	71K	74K	70K	74K	71K
$G_{r,o}^{\|E\|}$	360K	370K	351K	372K	356K
$\|C\|$	35K	38K	35K	32K	32K

Fig. 11. Dec. skeleton size vs θ_C/θ_S (XMark)

θ_C/θ_S	0.01	0.1	1	10	100
$G_{r,o}^{\|V\|}$	3K	2.8K	2.9K	2.9K	3.6K
$G_{r,o}^{\|E\|}$	163K	156K	166K	164K	195K
$\|C\|$	1.2M	932K	944K	810K	808K

Fig. 12. Dec. skeleton size vs θ_C/θ_S (DBLP)

For both XMARK and DBLP datasets, we noted that the effectiveness of our approach increases as the value of I increases until I is close to 1. The size of the search space of the heuristics increases as I increases. Thus, the heuristics has a higher chance of returning good cuts. However, when I is close to 1, the search space, hence the number of local maxima, becomes too large. In such cases, the quality of cuts returned by the heuristics reduces. The results from XMARK and DBLP datasets exhibited similar trends. However, the average case of DBLP (Figure 10) is relatively closer to the results in Figure 1, which were obtained from the best of the five tries. This can be explained by the problematic case in XMARK discussed earlier.

Consider each pair of adjacent columns in Figure 9 and Figure 10. We obtained the best compression improvement when I was switched from 0 to 0.2. The improvement between other consecutive columns was relatively minor. This indicated that in practice, if decomposition helped compression at all, a small number of stars was sufficient.

In the next experiment, we altered the value of θ_S and θ_C and observed the quality of cuts returned by the heuristics. I has been set to 0.8. The numbers reported are the average of local maxima returned by five tries. In addition, we reported the number of edges across the cut $|C|$. The results were summarized in Figure 11 and Figure 12. The heuristics reports better compression but worse $|C|$ as θ_C/θ_S decreases. The trend is not observable from the results of XMARK dataset as it contains poorly-compressed subtrees ($e.g.,listpar$) not modeled by the cost function.

Figure 13 presented the effect of applying decomposition recursively on DBLP dataset. Consider the first decomposition. The number of nodes and edges in the decomposed skeletons are reduced to 23% and 37% of their original values. However, extra storage is needed to store 359K edges crossing the cut in data vectors. When decomposition is applied on the reduced document, further improvement on compression (40% for the nodes and 70% for the edges) can be achieved with an overhead of storing 116K cross edges. Not surprisingly, when the outlier document is further decomposed, the improvement on compression is negligible: The heuristics aimed at separating compression-unfriendly subtrees from the original skeleton and grouped them in the outlier document. Furthermore, the decomposition of the outlier document requires storing additional 153K edges. This experiment showed that the compression improvement of our method reduces as more decompositions are applied.

# dp.	$G_{r,o}^{\|V\|}$	$G_{r,o}^{\|E\|}$	$\|C\|$
0	4.4K	225K	0
1	1.0K + 0.6K	83K + 2K	359K
3	(0.6K + 48) + (0.5K + 36)	(25K + 0.1K) + (2K + 67)	359K + (116K + 153K)

Fig. 13. Efficiency of recursive cuts on DBLP

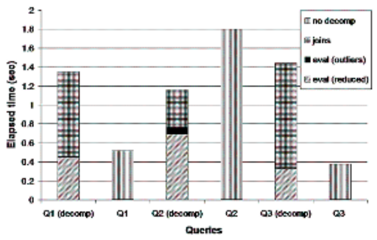

Fig. 14. Performance of XMark queries involving cross edges

Experiments on XMARK queries. We conducted an experiment on querying XMARK dataset with or without decomposition. The paths in the cut returned by our heuristics are listed below.

$/site/regions/europe/item/incategory$
$/site/regions/namerica/item/incategory$
$/site/people/person/watches/watch$
$/site/open\_auctions/open\_auction/annotation/description/parlist/listitem$
$/site/closed\_auctions/closed\_auction/annotation/description/parlist/listitem$

Except $Q6$, $Q7$, $Q15$, $Q19$, all queries in XMARK benchmark [17] can be evaluated by using the reduced document alone and hence query performance is improved by evaluating the queries on smaller skeletons. We summarized the performance of the queries involving cross edges in Figure 14. $Q1$, $Q2$ and $Q3$ are renaming of the relevant path queries in $Q6/Q19$, $Q15$ and $Q7$ in [17], respectively.

Sort-merge join algorithm is used for the joins on data vectors encoding the cross edges. The result of $Q1$ and $Q3$ are similar. The outlier skeleton participates the query because of the descendant step in the path queries. The join on the cross edges introduces a significant overhead on query processing. We noted retrospectively that the outlier skeleton is small and the queries on the outlier skeletons are evaluated to empty sets. In this case, the join could be eliminated by evaluating the corresponding path queries on the two skeletons prior to the join. By doing so, query performance on skeletons with and without decomposition were comparable. The selectivity of $Q2$ is low. The join in $Q2$ required less time than the joins in $Q1$ and $Q3$. In addition, path evaluation on the decomposed skeletons is faster simply because smaller skeletons are being processed.

6 Related Work

XML compression techniques can be roughly categorized into syntactic technique and semantic technique. The compression technique considered in this paper is a semantic compression technique derived from sharing of common subtrees [4, 3]. Semantic compressions have also been proposed to support data mining applications [1, 10, 11]. The objective of their schemes is to compute representative tuples of a relation. However, [1, 10, 11] assumed relational data and their support on XML remains unexplored.

Closest to our work is the STORED system [7]. The system transforms XML into a set of relations and subsequently, store, query and manage XML in a relational database system. The major distinction between our scheme and STORED is that we shred XML to XML, as opposed to relations. Note also that an extreme of our method, full decomposition, yields the edge table of an input document, where skeleton compression is no longer relevant. At the core of STORED is a data-mining algorithm for typical tree structures [21] in a set of trees. However, without projections, as discussed in [7], [21] would generate a relational schema that covers only a small portion of the data. Due to the impedance mismatch of the tree model and the relational model, storing the outliers (irregular or dissimilar structures) in relations can be space-inefficient. In comparison, we treat the outliers as an XML document and compress them with XML compression.

There is a host of work on mining transactional data [9]. Typically, a database consists of a set of transactions, each of which represents a set of items. There is a natural connection between our algorithm and this class of algorithms. Subtrees can be readily regarded as transactions. Unfortunately, the number of subtree structures in a document is $O(2^{|T|})$. We tackled this problem by pruning the minor subtrees (stars) through a coarse estimation followed by a scalable way of summarizing the subtree structures.

Finally, efforts are spent on syntactic XML compressors [6, 16, 5, 18]. [6, 16, 18, 5] treat XML data as tokens of elements, attributes and text. Customized syntactic compression is derived for handling these data separately. These techniques (*e.g.,* arithmetic coding, dictionary-based static coding) are fundamentally different from ours.

7 Conclusions and Future Work

We have proposed a heuristic approach of decomposing XML document for yielding better compression. By using our method, we have not encountered a real-world dataset whose decomposed-compressed skeletons could not be fit into the main memory of a commodity computer, which was not the case before. Despite the improvement on compression, the new compressed representation may introduce overhead on query processing. This paper presented an experimental study on the decomposition and the heuristic algorithm and preliminary results on querying decomposed-compressed skeletons.

We have planed to extend our algorithm for optimizing compression in the presence of query workload and statistics to optimize queries. We are investigating on applying the decomposition as a data partition algorithm of distributed XML query processing.

References

1. S. Babu, M. N. Garofalakis, and R. Rastogi. Spartan: A model-based semantic compression system for massive data tables. In *SIGMOD*, pages 283–294, 2001.
2. S. Berchtold, C. Bohm, D. A. Keim, and H.-P. Kriegel. A cost model for nearest neighbor search in high-dimensional data space. In *PODS*, pages 78–86, 1997.
3. P. Buneman, B. Choi, W. Fan, R. Hutchison, R. Mann, and S. Viglas. Vectorizing and querying large xml repositories. In *ICDE*, pages 261–272, 2005.
4. P. Buneman, M. Grohe, and C. Koch. Path Queries on Compressed XML. In *VLDB*, pages 141–152, 2003.

5. J. Cheney. Compressing XML with multiplexed hierarchical PPM models. In *Data Compression Conference*, pages 163–172, 2001.
6. J. Cheng and W. Ng. Xqzip: Querying compressed xml using structural indexing. In *EDBT*, pages 219–236, 2004.
7. A. Deutsch, M. F. Fernandez, and D. Suciu. Storing semistructured data with STORED. In *SIGMOD*, pages 431–442. ACM Press, Jun. 1999.
8. J. Gray, D. Slutz, A. Szalay, A. Thakar, J. vandenBerg, P. Kunszt, and C. Stoughton. Data mining the SDSS Skyserver database. Technical Report MSR-TR-2002-01, Microsoft, 2002.
9. J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, 2000.
10. H. V. Jagadish, J. Madar, and R. T. Ng. Semantic compression and pattern extraction with fascicles. In *VLDB*, pages 186–198, 1999.
11. H. V. Jagadish, R. T. Ng, B. C. Ooi, and A. K. H. Tung. Itcompress: An iterative semantic compression algorithm. In *ICDE*, pages 646–657, 2004.
12. Language and Information in Computation at Penn. Penn treebank project. Available at `http://www.cis.upenn.edu/~treebank/`.
13. M. Ley. Dblp bibliography. Available at `http://www.informatik.uni-trier.de/~ley/db/`, Mar 2005.
14. H. Liefke and D. Suciu. XMill: an efficient compressor for XML data. In *SIGMOD*, pages 153–164, 2000.
15. E. Miller, R. Swick, D. Brickley, B. McBride, J. Hendler, G. Schreiber, and D. Connolly. Semantic Web. W3C Working Group, August 2005. `http://www.w3.org/2001/sw/`.
16. J.-K. Min, M.-J. Park, and C.-W. Chung. Xpress: a queriable compression for xml data. In *SIGMOD*, pages 122–133, 2003.
17. A. Schmidt, F. Waas, M. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB*, pages 974–985, 2002.
18. P. M. Tolani and J. R. Haritsa. Xgrind: A query-friendly xml compressor. In *ICDE*, pages 225–234, 2002.
19. U.S. National Library of Medicine. MEDLINE distributed in XML format. Available at `http://www.nlm.nih.gov/bsd/licensee/data_elements_doc.html`.
20. P. Valduriez. Join indices. *TODS*, 12(2):218–246, 1987.
21. K. Wang and H. Liu. Discovering typical structures of documents: a road map approach. In *SIGIR*, pages 146–154, 1998.
22. J. Ziv and A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.