Abstract

Numerous researchers have proposed to use relational databases to store and query XML documents. One important component of such systems is the XML subtree reconstruction, which reconstructs the subtrees rooted at the matching nodes of an XML query and returns them to the user as the query result. Existing reconstruction algorithms either do not support recursive XML view schema, or require expensive nested queries or joins of multiple relations. In this paper, we propose an efficient XML subtree reconstruction algorithm, Reconstruct, which overcomes these limitations and uses an efficient stack-based structural join algorithm to recover all the parent-child relationships between elements. One salient advantage of this algorithm is that it employs the inlining feature of the inlining-based storage of XML documents, which is known as one of the best relational XML storage schemes. Both our algorithmic analysis and experimental study show that Reconstruct is efficient and scalable.

1. Introduction

Numerous researchers have proposed to use relational databases to store and query XML documents [4, 8, 16, 21, 12, 14, 2, 13]. In these systems, the elements selected by an XML query are returned to an application in one of the following two modes [23]: 1) Select mode: the unique identifiers (IDs) of the selected elements are returned to the application, which can extract the contents of these elements if necessary, or 2) Reconstruct mode: the XML subtrees rooted at the selected elements are extracted and reconstructed from the storage of XML documents and returned to the application. One constraint is that these XML subtrees must conform to the structure imposed by the original input XML schema or the input XML documents. Therefore, XML subtree reconstruction is one important component of such systems and its performance has great impact on the query response time.

While existing work on XML publishing [19, 18, 5, 6, 7, 3] focuses on presenting existing relational data in XML format, these algorithms are not directly applicable to the XML subtree reconstruction problem we pose here for the following reasons: 1) the output XML view is defined by an application in XML publishing, while in XML subtree reconstruction, the structure of the output XML subtrees must conform to the structure imposed by the original input XML schema or the input XML documents, 2) they only support non-recursive XML view schema while an input XML schema might be recursive or the input XML documents might contain element e under another element e, 3) they do not exploit the inlining feature of the relational storage of XML documents, where the main idea of inlining is to store an element f in the same tuple as its parent e if e contains at most one f. The inlining approach has been proved extremely effective in storing and querying XML documents in relational databases [21, 13], 4) they use expensive nested queries and joins of multiple relations.

While some authors briefly discussed the XML subtree reconstruction in various contexts [11], no algorithmic details were provided. Similarly, [23, 24, 8] only presented some experimental results on XML subtree reconstruction without the algorithmic details. Although [20] did present an algorithm for generating reconstruction XML view, the reconstruction relies on an XML query engine that is already in place: the output XML view is specified in terms of an XML query and the reconstruction is performed by executing the XML query against a default XML view of the underlying relational data over the XML query engine. The assumption of the existence of such an XML query engine, only pushes the problem of XML subtree reconstruction inside the query engine, with the immediate limitation that the performance of the reconstruction will be highly dependant on the performance of the XML query engine. In addition, as the XML query engine is one major goal of our system, we cannot assume the existence of such an XML query en-
gine which internally supports XML reconstruction to implement XML reconstruction to avoid the egg and chicken problem.

Recently, [11] proposes a strategy to deal with recursive XML view schemas in reconstructing XML subtrees, unfortunately, it needs to construct the root-to-leaf path dynamically and reconstruction involves joins of multiple relations. In addition, the inlining feature of the relational storage of XML documents is not exploited.

The main contributions of this paper are:

1. We propose an efficient XML subtree reconstruction algorithm, Reconstruct, which overcomes the limitations of these existing algorithms and uses an efficient stack-based structural join algorithm to recover all the parent-child relationships between elements. One salient advantage of this algorithm is that it employs the inlining feature of the inlining-based storage of XML documents, which is known as one of the best relational XML storage schemes.

2. We present our algorithmic analysis to show that our Reconstruct algorithm is both correct and efficient with the time complexity of \( O(n + n_r \log n_e) \), where \( n_e \) and \( n_r \) are the numbers of elements and attributes respectively in the output XML subtree, and \( n = n_e + n_r \).

3. We conduct our experimental study to show that the proposed algorithm Reconstruct is efficient and scalable.

Organization. The rest of the paper is organized as follows. In section 2, we discuss related work, and in Section 3, we give a brief overview of our schema mapping algorithm. In Sections 4, 5, and 6, we present XML subtree reconstruction algorithm Reconstruct, analyze its time complexity, prove its correctness, and provide experimental study results. Finally, we present our conclusions and future work in Section 7.

2. Related Work

Numerous researchers have proposed to use relational databases to store and query XML documents [4, 8, 16, 21, 12, 14, 2, 13]. The main challenge of this approach is that one needs to resolve the conflict between the hierarchical, ordered nature of the XML data model and the flat, unordered nature of the relational data model. This conflict can be resolved by the following three XML-to-Relational mappings:

- Schema mapping. Either a fixed generic relational schema (Schema-oblivious XML storage [4, 8, 16, 9]) is used, or a relational schema is generated from an XML schema or DTD (Schema-based XML storage [21, 12, 2, 14, 13]) for the storage of XML documents. To support the ordered nature of the XML data model, an element order encoding scheme such as those proposed in [23] can be used and additional columns are introduced to store the ordinals of XML elements.
- Data mapping, which translates an input XML document into relational tuples and inserts them into the relational database [1].
- Query mapping, which translates an XML query into SQL queries, executes them against the database and returns the query result to the user [11, 3, 6, 10, 15, 18, 20]. If the query result is to be returned as XML documents, then a reconstruction algorithm is needed to reconstruct the XML subtrees rooted at the matching nodes.

Although XML publishing [19, 18, 5, 6, 7, 3] is very relevant to the XML subtree reconstruction posed in this paper, there is a fundamental difference between these two problems: XML publishing focuses on presenting existing relational data in XML format, hence, there is no direct correspondence between the relational schema and the output XML view schema. However, XML subtree reconstruction is conducted over relational data which is the storage of XML documents. The relational schema is generated from the input XML schema, and it is required that the reconstructed XML documents must conform to the input XML schema in structure.

3. Relational Storage of XML documents

In this section, we give an overview of our schema-based relational storage of XML documents. In particular, we present a schema mapping algorithm, OD SSDMap, which generates a relational schema from an XML DTD for storing and querying ordered XML documents. Although our proposed XML subtree reconstruction algorithm is described in this context, it can be easily adapted to other relational storage schemes of XML documents.

First, to facilitate the processing of the input XML DTD\(^1\), we model it with a DTD graph, in which nodes represent element or attribute types, and edges represent parent-child relationships. In addition, for each edge from node \( e \) to \( f \) in the graph, if \( e \) can contain at most one \( f \), then this edge will not be labeled; otherwise, it will be labeled by a \( @ \). For example, given the DTD in Figure 1, the corresponding DTD graph is shown in Figure 2 where attribute labels are preceded by a \( @^2 \).

---

1. Since the input DTD might be very complex due to its hierarchical nesting nature, the simplification step described in [13] will be applied first to simplify the input DTD.
2. In implementation, to ensure the uniqueness of attribute labels, we can use the concatenation of an attribute name and its owner element name as the attribute label. For example, attribute \( id \) of element \( book \) can have label \( book.id \).
deal with a set-valued attribute, the edge between the attribute and its parent (the owner element) is labeled by a *.

In Figure 2, cites is a set-valued attribute.

Second, we classify the nodes in the DTD graph into three categories:

- **Root node**, a node without incoming edges. For example, xbib is the root node of the DTD graph shown in Figure 2.
- **Shared node**, a node with more than one incoming edge. For example, paragraph is a shared node of the DTD graph shown in Figure 2.
- **Inlinable node**, a node with exactly one incoming edge and that edge is not labeled by a *.

For example, given the inlined DTD graph shown in Figure 3, the relational schema shown in Figure 4 is generated. For each node, a column (e.ID or f.ID), which uses the global order encoding proposed in [23]. However, our algorithm can be easily adapted to other order encodings [23]. Attribute parentID is introduced for each non-inlinable element to preserve the tree structure of an XML document. To facilitate the processing of recursive XML queries (queries with // axis), each element e is associated with an endID, which stores the maximum ID of e’s descendant. For a shared node, parentType is introduced to indicate the parent type of an element. A column (e or f) is introduced for each leaf element or an attribute to store its string value.

For example, given the inlined DTD graph shown in Figure 3, the relational schema shown in Figure 4 is generated.
Algorithm Reconstruct

Input: element \( e \) with \( e.ID \), \( e.endID \) and \( e.name \) have been initialized

Output: return the XML document rooted at element \( e \)

Begin

Element list elist = ReturnDescendants(e)
String xmlstr = ""
Stack S = EmptyStack()
root = elist.next()
xmlstr.append("<\$root.name") // reconstruct the opening tag
For each attribute \( a_i \) of \( root \) do // reconstruct the attribute list
xmlstr.append("\$root.a_i.name = \$root.a_i.value")
End For
xmlstr.append(">")
S.push(root)
// reconstruct the descendants of root
e = elist.next()
While e is not NULL do
If e.endID \( < \) S[top].endID then // e must be a child of S[top]
xmlstr.append("<\$e.name") // reconstruct the opening tag of subelement e
For each attribute \( a_i \) of \( e \) do // reconstruct the attribute list of subelement e
xmlstr.append("\$e.a_i.name = \$e.a_i.value")
End For
xmlstr.append(">")
S.push(e)
e = elist.next()
Else // all the children of S[top] have been reconstructed
xmlstr.append("\$/S[top].name")
xmlstr.append("<</S[top].name>")
S.pop()
End If
End While
While S is not empty do
xmlstr.append("S[S[top].value"]
xmlstr.append("<</S[top].name>")
S.pop()
End While
Return xmlstr
End Algorithm

Figure 5. Algorithm Reconstruct

4. XML Subtree Reconstruction

In this section, we propose an efficient XML reconstruction algorithm Reconstruct, which reconstructs the XML subtree rooted at a given element (or the root element to reconstruct the whole original XML document). The algorithm can be used to reconstruct the subtrees of the matching nodes of an XML query, which are returned to the user as the query result.

Figure 5 presents the pseudo code of algorithm Reconstruct for reconstructing an XML subtree rooted at a non-inlinable element \( e \). The algorithm for reconstructing an XML subtree rooted at an inlinable element is similar and we leave it as an exercise. Basically, Reconstruct performs two tasks: (1) to retrieve a structure-encoded sequence of the to-be-reconstructed XML subtree rooted at element \( e \), and then (2) to perform a stack-based structural join to reconstruct the XML subtree.

Algorithm ReturnDescendants given in Figure 6 corresponds to the first task which returns the structure-encoded sequence in list elist, where the structure-encoded sequence of an XML tree is defined as the list of elements in the tree sorted by their IDs, and each element \( e \) in the list stores complete information about a single XML element:

- \( e.name \), the name of the XML element.
- \( e.ID \), the global ID of each XML element (preorder numbering scheme).
- \( e.endID \), the largest ID of \( e \)’s descendants and \( e.endID = e.ID \) if \( e \) is a leaf of an XML tree.
- \( e.attributes \), the set of XML attributes of \( e \). We also denote the attributes of \( e \) by \( e.a_1, \ldots, e.a_n \), and the names and values of these attributes by \( e.a_i.name \) and \( e.a_i.value \) respectively (\( i = 1, \ldots, n \)).
- \( e.value \), the textual data enclosed between opening and closing tags of \( e \), and \( e.value = \text{NULL} \) if \( e \) has no textual content.

In order to select tuples of the descendants of the input element \( e \), algorithm ReturnDescendants uses the input DTD to find element types and corresponding table names where such tuples can be stored (line 06). The mapping function from DTD element type names to relational table names is denoted by \( \sigma \) and generated at the schema mapping stage. The algorithm uses the property that a descen-
## Algorithm ReturnDescendants

**Input:** element \( e \) with \( e.ID \), \( e.endID \) and \( e.name \) have been initialized

**Output:** return the list of all the descendants of \( e \) including \( e \) in the ascending order of element IDs

**Begin**

1. Element list \( elist = \text{EmptyList}() \)
2. For each descendant-or-self element type \( d \) of \( e \) do
   - \( tlist = \text{Select tuples From} \ \sigma(d) \ \text{Where} \ ID > e.ID \ \text{AND} \ ID < e.endID \)
3. For each tuple \( t \) in \( tlist \) do
   - For each element type \( f \) inlinable to \( d \) do
     - // decompose tuple \( t \) into multiple elements
       - Create an element \( g \) of type \( f \)
       - \( g.ID = t.f.ID \)
       - \( g.endID = t.f.endID \)
       - \( g.name = f.name \)
       - For each attribute \( a_i \) of \( f \) do
         - \( g.a_i.value = t.f.a_i.value \)
   - \( \text{elist.append}(g) \)
4. \( \text{End For} \)
5. \( \text{End For} \)
6. Order elements in \( elist \) in the ascending order of element IDs
7. \( \text{Return} \ elist \)
8. \( \text{End Algorithm} \)

**Figure 6. Algorithm ReturnDescendants**

Algorithm ReturnDescendants returns the structure-encoded sequence of the XML subtree rooted at \( e \).

**Proof:**

1. Line 06 selects element types \( d \) from the DTD for the input element \( e \) and all its descendants.
2. For each distinct table name \( \sigma(d) \), line 07 selects tuples for element \( e \) and its descendants based on the

## 5. Correctness and Complexity Analysis

To show that \( \text{Reconstruct} \) indeed returns the original XML subtree, we need to establish the correctness of the algorithm. In the following, Lemma 5.1 through Lemma 5.3 identify some properties of algorithms ReturnDescendants and Reconstruct, and then in Theorem 5.4, we prove that algorithm \( \text{Reconstruct} \) correctly returns the XML subtree rooted at element \( e \).

**Lemma 5.1** Algorithm ReturnDescendants returns the structure-encoded sequence of the XML subtree rooted at \( e \).

**Proof:**

- Line 06 selects element types \( d \) from the DTD for the input element \( e \) and all its descendants.
- For each distinct table name \( \sigma(d) \), line 07 selects tuples for element \( e \) and its descendants based on the
following property: e’s descendant has greater ID and not greater endID. To select the input element e, SQL query also select the element with ID equal to e.ID. Therefore, line 07 selects elements where ID >= e.ID AND ID <= e.endID.

- Selected tuples may contain information about one or many elements in case when there are inlined elements. For each tuple (line 08), based on the information about inlined to σ(d) and the corresponding tuple element types (line 09), lines 11-16 decompose tuple to multiple elements.

- Finally, line 17 adds elements to the list and line 21 sorts the list by element ID.

**Lemma 5.2** In algorithm Reconstruct, let e be the first element of elist in the order of the list, where elist is a structure-encoded sequence of an XML subtree (Lemma 5.1), we have S[top].id < e.id. In addition, we have e.endID <= S[top].endID as the condition of this lemma. Therefore, e must be a child of S[top].

**Proof:** Since in algorithm Reconstruct we process elements in elist in the order of the list, where elist is a structure-encoded sequence of an XML subtree (Lemma 5.1), we have S[top].id < e.id. In addition, we have e.endID <= S[top].endID as the condition of this lemma. Therefore, e must be a descendant of S[top].

Suppose e is not a child of S[top]. Let f be the parent of e. f must be a descendant of S[top]. We have S[top].id < f.id < e.id. f must remain on the stack above S[top] since: (1) f will be popped out from the stack only when an element after f has an endID that is greater than f.endID, but the elements between f and e in elist must be descendants of f, therefore, their endIDs are smaller or equal to f.endID; and (2) f is a descendant of S[top] and f.endID < S[top].endID. This contradicts the fact that S[top] is the top element of the stack. Hence, e must be a child of S[top].

**Lemma 5.3** In algorithm Reconstruct, let e be the first element of elist, and S[top] be the top element of stack S. If e.endID > S[top].endID, then no element in elist following e is a child of S[top].

**Proof:** Suppose there is an element f following e in elist that is a child of S[top], then all the elements between S[top] and f in elist including e must be descendants of S[top]. Therefore, we have e.endID < S[top].endID. This contradicts the condition of the lemma. Hence, no element in elist following e is a child of S[top].

**Theorem 5.4** Algorithm Reconstruct returns the XML subtree rooted at element e.

**Proof:** [Sketch] First, we call algorithm ReturnDescendants to retrieve the structure-encoded sequence elist of the to-be-reconstructed XML subtree. Second, we output the opening tag of the first element of elist, which is the root element, and push it into the stack. Third, for each next element e from elist, if e.endID < S[top].endID, then according to Lemma 5.2, e must be a child of S[top], we output its opening tag and attributes and push it into the stack.
As a result, the stack contains the path from the root to the current element. Otherwise \((e. endID > S[top]. endID)\), according to Lemma 5.3, all the children of \(S[top]\) have been output. We then output the value and closing tag of the top element of the stack and pop the stack. Finally, after all elements in \(elist\) are processed, lines 33-35 output a value and a closing tag for each element in the stack and pop it. Since \(Reconstruct\) correctly establishes parent-child relationships and respects the order of the structure-encoded sequence, thus, it will output correctly all the children of an element in the document order. Therefore, \(Reconstruct\) correctly returns the XML subtree rooted at element \(e\).

Finally, Theorem 5.5 and Theorem 5.6 state the time complexity of algorithms \(ReturnDescendants\) and \(Reconstruct\).

**Theorem 5.5** The time complexity of algorithm \(ReturnDescendants\) is \(O(n + n_e \log_2 n_e)\), where \(n_e\) and \(n_a\) are the numbers of elements and attributes respectively in \(elist\), the output of \(ReturnDescendants\), and \(n = n_e + n_a\).

**Proof:** [Sketch] It follows from the fact that we retrieve each descendant element/attribute of \(e\) from the database in constant time and the sorting of \(elist\) in the order of \(id\) takes time \(O(n_e \log_2 n_e)\).

**Theorem 5.6** The time complexity of algorithm \(Reconstruct\) is \(O(n + n_e \log_2 n_e)\), where \(n_e\) and \(n_a\) are the numbers of elements and attributes respectively in the output XML subtree, and \(n = n_e + n_a\).

**Proof:** [Sketch] Calling \(ReturnDescendants\) takes time \(O(n + n_e \log_2 n_e)\). The algorithm processes each element in the output XML subtree in constant time: (1) output opening tag and push the element to a stack; 2) pop the element from the stack and output its closing tag. Similarly, the algorithm processes each attribute in constant time. Therefore, the time complexity of algorithm \(Reconstruct\) is \(O(n + n_e \log_2 n_e)\).

6. Experimental Study

For our experimental study, we implemented algorithm \(Reconstruct\) and three versions of algorithm \(ReturnDescendants\):

1. **Naive:** naively retrieves each descendant element from the database with a separate query.
2. **Server-side – \(ReturnDescendants\) (S – RD):** retrieves all descendant elements in ID order with a few queries according to Figure 6 (inline elements are retrieved in the same query of their non-inlinable parents) and stores them in a database server in-memory buffer, which serves as \(elist\) described in the algorithm. Client software stores only a cursor to access \(elist\) elements.
3. **Client-side – \(ReturnDescendants\) (C – RD):** retrieves all descendant elements in ID order with a few queries according to Figure 6 (inline elements are retrieved in the same query of their non-inlinable parents) and stores them in a client in-memory buffer, which serves as \(elist\) described in the algorithm.

The above algorithms were coded using Java 1.4.1 software development kit, Personal Oracle Database 10g Release 10.1 was used as an XML storage. Experiments were run on the computer with CPU Pentium IV 2.4 GHz and RAM 512 MB operated by Windows XP Professional. For each experiment, we performed the reconstruction of the whole XML document for 6 times and computed the average of last 5 runs ignoring the first run. In all experiments, we reconstructed XML data in memory and did not output it into a file or on a screen to avoid unnecessary I/O operations.

Further in this section, when we refer to \(Naive\), \(S – RD\) or \(C – RD\) as a version of \(ReturnDescendants\), the reader should assume it is used in conjunction with \(Reconstruct\), which performs actual XML document reconstruction.

6.1. **Comparison of Naive, S-RD and C-RD**

Using the XMark benchmark [26, 17], we generated a set of XML documents, which conform to the auction.dtd, and stored them in the database. To evaluate scalability and performance of our algorithms, we reconstructed XML documents of size 5, 10, 15, 20, 25, and 30 MB. Experimental results for \(Naive\), \(S – RD\) and \(C – RD\) are given in Figure 9 (time axis has a logarithmic scale). Based on the performance, implementations are ranked from the best to the worst as follows: \(S – RD\), \(C – RD\) and \(Naive\). Although \(S – RD\) and \(C – RD\) are algorithmically similar, \(C – RD\) is slower, because it additionally performs an allocation of a client-side in-memory buffer to copy data from the server to the client. \(Naive\) is the slowest one, as it runs a lot of queries, equal to the number of elements in an XML document. We did not consider \(Naive\) in our further experiments, because it is too slow to compete with \(S – RD\) and \(C – RD\).

6.2. **S-RD and C-RD performance on different data sets.**

To evaluate the performance of \(S – RD\) and \(C – RD\) on different data sets, we ran our program on 500KB XML documents that conform to auction.dtd [26, 17], article.dtd [25] and sigmodrecord.dtd [22]. Table 1 shows important statistical information about these data sets, such as the number of elements in DTD documents, the number of generated relational tables, the number of elements in XML documents, and the number of tuples in
The histogram given in Figure 10 shows the experimental results for $S-RD$ and $C-RD$ performance on different data sets. The best performance is achieved on the XML instance `article.xml`, which conforms to `article.dtd`. Even though `article.xml` reveals that only 1796 – 1738 = 58 XML elements were inlined, `article.dtd` allows inlining of 26 – 8 = 18 distinct DTD elements, which decreases the number of tables and queries, making $S-RD$ and $C-RD$ run faster. Additionally, the number of XML elements in `article.xml` is smaller than in the other data sets. For the second winner `sigmodrecord.xml`, the number of tables is small (4), however, the number of XML elements is large (11526), which decreases the performance. Finally, `auction.xml` contains many elements (6639) which are stored in many tables (26), so that our algorithms take the most time in this case.

In summary, our `Reconstruct` and `ReturnDescendants` ($S-RD$ and $C-RD$) implementations are efficient and scale well with the size of XML documents. Their performance highly depend on intensity of inlining and the quantity of elements in an XML instance. In particular, the performance increases when the number of inlined DTD elements increases and the number of XML elements decreases. Preliminary experimental results on indexing techniques imply an advantage of the multi-column index on ID and endID over the other approaches (no index, the index on ID), however, further study should be conducted on files of much larger sizes.
7. Conclusions and Future Work

We presented algorithm Reconstruct to reconstruct XML subtrees, rooted at an arbitrary node, from a relational storage of XML documents. We showed that this algorithm is applicable in conjunction with ODTDMAP, however, it can be easily adapted for usage with other schema mapping algorithms.

Reconstruct has the following advantages over existing algorithms. It can reconstruct XML documents with recursive schema and does not require the user to specify XML view manually. It is efficient in terms of time complexity – order of $\log n$, number of relational joins – it exploits the inlining feature of the inlining-based storage of XML documents. Reconstruct is formally proven to be correct. The extensive experimental study showed that Reconstruct and ReturnDescendant implementations are efficient and well-scalable.

Our ongoing work includes: (1) the development of a generic reconstruction algorithm that can be used across various XML database platforms; (2) the improvement of the reconstruction algorithm so that it is linear rather than logarithmic.

References