Review

An efficient XML query pattern mining algorithm for ebXML applications in e-commerce

Tsui-Ping Chang

Department of Information Technology, Ling Tung University, Taichung 408, Taiwan, R.O.C.

Received 15 March, 2011; Accepted 4 September, 2014

Providing efficient query to XML data for ebXML applications in e-commerce is crucial, as XML has become the most important technique to exchange data over the Internet. ebXML is a set of specifications for companies to exchange their data in e-commerce. Following the ebXML specifications, companies have a standard method to exchange business messages, communicate data, and business rules in e-commerce. Due to its tree-structure paradigm, XML is superior for its capability of storing and querying complex data for ebXML applications. Therefore, discovering frequent XML query patterns has become an interesting topic for XML data management in ebXML applications. The study presents an efficient mining algorithm, namely ebX2Miner, to discover the frequent XML user query patterns. Unlike the existing algorithms, the study proposes a new idea by encoding the XML user queries and then storing these codes to generate the frequent XML user query patterns. Furthermore, the simulation results show that the ebX2Miner outperforms other algorithms in its execution time and used memory space.

Key words: XML query pattern mining, XML query, encoding scheme, ebXML, e-commerce.

INTRODUCTION

XML (Cunningham, 2005) has become the de facto standard for data representation and exchange in e-commerce. The self-describing property empowers XML to represent data without losing semantics, and the semi-structure nature allows XML to model a wide variety of data. As a result, in e-commerce, many applications utilize XML and then follow the ebXML specifications (Bio, 2003) to exchange their data over the Internet. In consequence, the rapid growth of XML data in e-commerce has provided the impetus to design and develop the systems that can efficiently store and query XML data for ebXML applications. ebXML (Bio, 2003) is a set of specifications which are designed by OASIS (Moberg, 2007) for companies to exchange data in e-commerce. These specifications together enable a modular electronic business framework and are designed based on XML technology. Following the ebXML specifications, companies have a standard method to exchange business messages, communicate data, and business rules in e-commerce. These business messages, communicate data, and rules are described by XML and with the same data frame between different companies. Therefore, most of XML data in ebXML applications has the same standard data structure and
results in most of their queries may have the same structure with query XML data. Since XML data in ebXML applications can be treated as trees with elements, attributes, and texts, the query languages, that is, XPath (Clark, 1999) and XQuery (Boag, 2010) are tree patterns with selection predicates on multiple elements that specify the tree-structured relationships. Thus, matching tree patterns against XML data is a core operation in XML query evaluation. This operation can be expensive since it involves navigation through the tree structure of XML data. As a result, the research efforts (Kwon et al., 2008; Lu et al., 2005; Raj et al., 2007) have been focused on the efficient evaluation of tree paths in XML queries.

Another approach (Bei et al., 2009; Chen et al., 2006; Gu et al., 2007; Yang et al., 2008) of improving XML query performance is to discover frequent XML query patterns and to design an index mechanism or cache the results of these patterns. Bei et al. (2009) and Yang et al. (2008) design a transaction summary data structure (that is, the global tree) to merge all of XML user query patterns. At the global tree, the XML candidate query sub trees are generated and their frequencies are thus counted by executing the tree-join process or database scans. As a result, the frequent XML query patterns are efficiently discovered on the processed global tree. In addition, in order to reduce the number of XML candidate query sub trees, Bei et al. (2009) and Yang et al. (2008) use the minimum support constraint to prune the infrequent XML query patterns on the global tree.

The existing approaches (Bei et al., 2009; Chen et al., 2006; Gu et al., 2007; Yang et al., 2008) may not be suitable to discover the frequent XML query patterns in ebXML applications and thus, degrade the system performance. Bei et al. (2009) and Yang et al. (2008) generate the XML candidate query sub trees from the global tree and use costly containment testing to prune the invalid candidate ones for the queries. However, in ebXML applications, most of XML queries have the same structure and results in most of the same query trees are processed. Also, in order to correctly count the frequencies of XML candidate query sub trees, the tree-join process or database scans are executed in their mining process. As a result, Bei et al. (2009) and Yang et al. (2008) still follow the traditional idea of generate-and-test paradigm, for XML query pattern mining and may not be suitable for ebXML applications.

This paper presents a novel algorithm, ebX2Miner, to mine the frequent query patterns for ebXML applications in e-commerce. ebX2Miner has the following advantages over the existing approaches. First, ebX2Miner focuses on the characteristic (that is, most of XML queries have the same structure) of ebXML applications and thus discovers the frequent XML query patterns with at most one database scan in the mining process. Although the existing algorithms could efficiently mine the frequent query patterns by constructing a tree model, two database scans are nonetheless necessary in order to correctly count the frequencies of candidate sub trees, thus, downgrading the system performance. Second, ebX2Miner encodes an XML query tree and stores its nodes’ codes to enhance the mining performance. The key concept in ebX2Miner is that the leaf nodes’ codes of a user query tree can preserve the tree’s structure information. This will greatly reduce the effort of exploring the search space and computing time.

The rest of this paper is organized as follows. Section 2 discusses the previous works related to ebXML applications and XML query pattern mining. Section 3 formalizes the XML frequent query pattern mining problem in this paper. Section 4 describes the details of ebX2Miner algorithm. Section 5 compares the ebX2Miner algorithm with other existing XML query pattern mining algorithms. Section 6 shows the results of the performance study, and Section 7 illustrates the conclusion and further work in this paper.

LITERATURE REVIEW

In this section, some related works are reviewed, including the papers of Bei et al. (2009), Bio (2003), Green et al. (2005), Kim (2002) and Yang et al. (2008) on the ebXML applications and frequent XML query pattern mining.

ebXML provides a modular suite of specifications that enables enterprises of any size and in any geographical location to conduct business over the Internet (Green et al., 2005; Kim, 2002). It purports to support the exchange and query of structured business documents between the applications of trading enterprises so as to support business processes within the trading partner organizations. Indeed, OASIS, one of the joint developers of ebXML, claims that ebXML takes advantage of cost effective Internet technology, is built on EDI experience with input from the EDI community. Therefore, by using ebXML over the Internet, an industry needs to define and collect its business processes, scenarios, and company business profiles, and makes them available through an industry ebXML registry (typically defined using UDDI). Then, structured business documents can be exchanged and queried between trading parties using the automated flow and sequence of interactions that ebXML prescribes.

Many new XML query pattern mining algorithms (Bei et al., 2009; Yang et al., 2008) have been proposed to discover the frequent XML query patterns. Yang et al. (2008) collect all of XML user queries to construct a global tree (T-GQPT) and then employ a rightmost expansion enumeration on the T-GQPT tree to generate XML candidate query sub trees. The main idea of rightmost expansion is that a query tree containing k nodes is generated by appending a new node to the right most path of a frequent sub tree containing (k-1) nodes. Thus,
many infrequent k-node trees are not enumerated if their (k-1)-node sub trees are infrequent. In addition, to compute the frequency of each candidate query sub tree, Yang et al. (2008) scan the database only when the candidate is a single branch tree. Among these algorithms, Fast XMiner (Yang et al., 2003) is the most efficient since the frequency of a non-single branch tree can be computed by joining the ID list of its proper rooted sub trees. On the other hand, 2PXMiner (Yang et al., 2008) extends Fast XMiner to discover the frequent XML query patterns that contain sibling repetitions. In order to speed up the mining performance, 2PXMiner computes the upper bound frequencies of XML candidate query sub trees and uses the minimum support constraint to early prune the infrequent query sub trees.

The VBU XMiner algorithm (Bei et al., 2008; Bei et al., 2009) also maintain a tree-like data structure, the CGTG tree, to merge all of XML queries to discover the frequent XML query patterns. In Bei et al. (2008), all of XML candidate query sub trees are enumerated based on the CGTG tree, and in Bei et al. (2009), the candidates whose frequencies are bigger than the minimum support value are enumerated. Thus, in Bei et al. (2009), before generating the candidate sub trees, the infrequent nodes in the CGTG tree are pruned. Also, the nodes in the CGTG tree are joined with their ancestor nodes which have the same IDs. Therefore, VBU XMiner generate candidate sub trees directly from the CGTG tree without scanning the database. In sum, it discovers the frequent XML query patterns on the processed CGTG tree.

Bei et al. (2008, 2009) and Yang et al. (2008) still follows the traditional idea of generate-and-test paradigm to mine the frequent XML query patterns and thus, have the following drawbacks for ebXML applications in e-commerce. First, they employ the rightmost expansion technique to enumerate all of XML candidate query sub trees on the global trees (that is, T-GQPT and CGTG tree). This approach merges all path and sub tree information of a user query tree in the global trees and thus requires unacceptable costs of tree-join process or database scan during the mining process. Second, a great deal of system space is used to process XML query databases, and the frequent XML query trees. In Bei et al. (2008, 2009) also maintain a tree-like data structure, the CGTG tree. In addition, Definition 4 defines the problem in this paper.

Definition 1: An XML query can be modeled as an unordered tree \( T_i = <N_i, E_i> \), where \( N_i \) is the node set, and \( E_i \) is the edge set. Nodes \( n \in N_i \) represent the elements, attributes, and string values in an XML query, and edges \( e \in E_i \) represent the parent-child relationships denoted by "/".

Definition 2: Given an XML query tree \( T_i = <N_i, E_i> \), an XML query rooted sub tree \( t_{ij} = <N_{ij}, E_{ij}> \), \( t_{ij} \) is considered to be the rooted subtree of \( T_i \) iff there exists:

1. \( \text{Root} (t_{ij}) = \text{Root}(T_i) \), where \( \text{Root}(t_{ij}) \) and \( \text{Root}(T_i) \) are the functions which return the root nodes of \( t_{ij} \) and \( T_i \) respectively.
2. \( N_{ij} \subseteq N_i, E_{ij} \subseteq E_i \).

Definition 3: Given an XML tree database \( D = \{T_1, T_2, ..., T_n\} \), where \( T_1, T_2, ..., T_n \) represent multiple XML query trees in \( D \).

Definition 4: Given an XML tree database \( D \) and a minimum support value \( m \) ranging from \((0, 1]\). The frequent XML query pattern mining problem is finding the set \( S \) of rooted subtrees \( t_i \) such that for each \( t_i \) in \( S \), \( \text{sup}(t_i) \geq m \) holds, where \( \text{sup}(t_i) \) is the equation: the number of \( t_i \) / the number of XML query trees in \( D \).

Problem statement

In this section, the problem statement is given to be solved. It begins by defining the XML query trees, their corresponding rooted sub trees, XML query tree databases, and the frequent XML query trees. Definition 1 defines an XML query tree. Definition 2 illustrates a rooted sub tree of an XML query tree. Definition 3 describes an XML query tree database, while Definition 4 defines the problem in this paper.
FREQUENT XML QUERY PATTERN MINING FOR ebXML APPLICATIONS

In this section, the study proposes an encoding scheme (namely XCode) to represent an XML tree with its corresponding query trees, a data structure (namely XList) to store the codes of XML nodes based on the XCode scheme, and a mining algorithm (namely ebX2Miner algorithm) based on XCode and XList to discover the frequent XML query patterns for ebXML applications in e-commerce.

An encoding scheme: XCode

XCode encodes the nodes of an XML tree in a xy coordinate system where xy is the coordinate of the two-dimensional space. The following symbols $T_i$, $r$, $k$, $p$, $l$, $fc$, and $nc$ are used to represent the nodes in an XML tree.

Symbol $T_i$ represents an XML tree, $r$ indicates the root node in $T_i$, $k$ represents a node in $T_i$, $p$ indicates the parent node of $k$, $l$ represents the left sibling node of $k$, $fc$ denotes the first child node of $k$, and $nc$ represents the child node of $k$ except the first child $fc$. The encoding rules are described for the nodes in an XML tree $T_i$ and listed as follows:

1. For an XML tree $T_i$, the root node $r$ is set on the origin whose coordinates $x$ and $y$ are $(0, 0)$.
2. For any node $k$ in the tree $T_i$, if $k$ is the first child node of its parent node $p$ and $p$'s coordinates are $(x_p, y_p)$, then $k$'s coordinates are $(x_p + 1, y_p + 1)$.
3. For any node $k$ in the tree $T_i$, if $k$ is the non-first child node of its parent node $p$ and its left sibling node $l$ has $m$ descendant nodes with the coordinates $(x_l, y_l)$, then $k$'s coordinates are $(x_l + m, y_l)$.

Note that, for simplify, hereafter, the coordinates of a
node in an XML tree based on the XCode scheme are namely the xcode of a node.

**Example 1.** Consider the XML tree in Figure 1. Suppose that all of nodes in the tree are encoded by the rules of the proposed XCode scheme. The **xcodes** of these nodes are shown in Figure 3. According to Rule (1), the root node book in the XML tree in Figure 1 is set on the origin and its **xcode** is (0, 0). According to Rule (2), the nodes title, XML, author1, john, jane, 2000, head1, origins, and head2 are the fc nodes of a node in the tree and their xcodes are (1, 1), (2, 2), (3, 2), (5, 3), (4, 3), (5, 2), (6, 2), (7, 3), and (8, 3) respectively. Also, by Rule (3), the nodes allauthor, year, chapter, author2, section1, and section2 are the nc nodes of a node in the tree and their xcodes are (2, 1), (4, 1), (5, 1), (4, 2), (7, 2), and (9, 3) respectively.

Derived from the XCode encoding rules, Lemmas 1, 2, 3 and 4 show the features of xcodes of an XML tree. Lemma 1 describes that an xcode reveals the level of a node in an XML tree, Lemmas 2 and 3 illustrate the relationship between two xcodes of nodes in an XML tree, and Lemma 4 illustrates that the values of xcode are bigger than or equal to 0.

**Lemma 1** for any two nodes f1 and f2 in an XML tree Ti with the xcodes (x1, y1) and (x2, y2) respectively, if node f2 is a child node of f1, then y2 = y1 + 1.

**Proof:** If f2 is the first child node of f1, according to Rule (2), the xcode (x2, y2) of f2 is equal to (x1+1, y1+1); otherwise, that is equal to (x+m, y), where (x, y) is the xcode of f1’s first child node f2 and f1 has m descendant nodes. Thus, if f2 is the first child node of f1, y2 = y1 + 1. In addition, since y2 = y and y1 = y1 + 1 which result in y2 = y = y1 + 1. As a result, y2 = y1 + 1.

**Lemma 2:** For any node f in an XML tree Ti, if f’s xcode is (x, y), then the value of y is equal to the level l of the node f in Ti.

**Proof:** We prove the lemma by showing that the value of y is equal to that of l. There are three cases, depending on whether node f is the root, fc, or nc node in Ti.

**Case 1:** Suppose that node f is the root node in Ti. According to Rule (1), the xcode of f is (0, 0). Thus, the value of y is equal to 0. Also, since f is the root node, f’s level/ is equal to 0. As a result, the value of y is equal to that of l.

**Case 2:** Suppose that f is the fc node in Ti. Since f is not the root node and with the level l, it has the ancestor nodes p0, p1, ..., p1, where pi is f’s parent node, p1 is i’s parent node, ..., and p0 is the root node. According to Rule (1), the xcode of p0 is (0, 0). Thus, yp0 is equal to 0. Also, according to Lemma 2, pi’s xcode ypi = ypi+1. Thus, ypi = ypi + 1 = 0 + 1 = 1. In consequence, p1’s xcode y2 = y2 + 1 = 1 + 1 = 2. Therefore, pi’s xcode ypi = l-1. Since f is the child node of pi, f’s xcode y = ypi + 1 = l - 1 + 1 = l. As a result, the value of y is equal to that of f’s level l.

**Case 3:** Suppose that f is the nc node and thus has a sibling node fc in Ti. According to Case 2, the fc’s xcode yc = l. In consequence, according to Rule (3), f’s xcode y is equal to yc. As a result, y = yc = l and the value of y is equal to that of f’s level l.

Based on Case 1, Case 2, and Case 3, we thus prove this lemma.

**Lemma 3:** For any two nodes f1 and f2 in an XML tree Ti with the xcodes (x1, y1) and (x2, y2) respectively, if node f2 is a descendant node of f1, then both of the values of x2 and y2 are bigger than those of x1 and y1 respectively.
Lemma 4: node

Proof: the study proves the lemma by showing that $x_2 > x_1$ and $y_2 > y_1$. There are two cases, depending on whether node $f_2$ is a child node or not of $f_1$.

Case 1: Suppose that node $f_2$ is a child node of $f_1$. If $f_2$ is the first child node of $f_1$, according to Rule (2), the xcode $(x_2, y_2)$ of $f_2$ is equal to $(x_1+1, y_1+1)$; otherwise, that is equal to $(x_1+m, y_1)$, where $(x_1, y_1)$ is the xcode of $f_1$'s first child node $f$, and $f$ has $m$ descendant nodes. Thus, if $f_2$ is the first child node of $f_1$, $x_2 = x_1 + 1$ and $y_2 = y_1 + 1$ which result in $x_2 > x_1$ and $y_2 > y_1$, respectively. In addition, since $x_2 = x_1 + m$, $y_2 = y_1$, $x_1 = x_1 + 1$, and $y_2 = y_1 + 1$ which result in $x_2 >= x_1 > x_1$ and $y_2 > y_1$. As a result, $x_2 > x_1$ and $y_2 > y_1$.

Case 2: Suppose that node $f_2$ is not a child node of $f_1$ and has a parent node $f_1$, which is a child node of $f_1$. According to Case 1, node $f_2$'s xcode $x_2 > x_1$ and $y_2 > y_1$. Also, since $f_2$'s xcode $x_2 > x_1$ and $y_2 > y_1$, they result $x_2 > x_1$ and $y_2 > y_1$.

Based on Case 1 and Case 2, we thus prove this lemma.

Lemma 4: For any node $f$ in an XML tree $T$, the values in $f$'s xcode $(x, y)$ are bigger than or equal to 0.

Proof: There are three cases, depending on whether node $f$ is the root, $fc$, or nc node in $T$.

Case 1: Suppose that node $f$ is the root node in $T$. According to Rule (1), $f$'s xcode $(x, y)$ is $(0, 0)$. As a result, the values in $f$'s xcode $(x, y)$ are equal to 0.

Case 2: Suppose that $f$ is the $fc$ node and $f$ has ancestor nodes $p_0, p_1, ..., p_n$ in $T$, where $p_0$ is $f$'s parent node, $p_n$ is $p_i$'s parent node, ..., and $p_0$ is the root node. According to Case 1, the values of $p_0$'s xcode are equal to 0. Also, according to Rules (2) or (3), the values of $p_i$'s xcode are the sum of those of $p_0$'s xcode with 1 or the number of descendant nodes of its sibling node. Therefore, the values of $p_i$'s xcode are bigger than 0. In consequence, according to Rules (2) or (3), the values of $f$'s xcode are bigger than 0. Since, according to Rule (2), the values in $f$'s xcode are the sum of those of $p_i$'s xcode with 1. As a result, the values in $f$'s xcode are bigger than 0.

Case 3: Suppose that $f$ is the nc node and thus has a sibling node $fc$ in $T$. According to Case 2, the values of $fc$'s xcode are bigger than 0. In consequence, according to Rule (3), the values in $f$'s xcode are the sum of those of $fc$'s xcode with 1 or the number of $fc$'s descendant nodes. As a result, the values in $f$'s xcode are bigger than 0.

Based on Case 1, Case 2, and Case 3, the study proves this lemma.

XList

In this subsection, the data structure XList that plays an important role in the design of our mining algorithm is described. XList is designed to record the xcodes of nodes in XML query trees. In order to store an XML node, in XList, a new node (namely xNode) with two variables and two pointers is created. Figure 4 (a,b) presents an example of the XML nodes stored in XNodes in XList.

Figure 4. The structures and contents of xNodes in XList.
node’s parent and sibling nodes are the \textit{book} and \textit{allauthor} nodes and linked by its \textit{parent} and \textit{sibling} pointers respectively. The \textit{xcodes} of nodes \textit{book} and \textit{allauthor} are (0, 0) and (2, 1) respectively, while the numbers of occurrences of those nodes are 5 and 3 respectively. In addition, the \textit{s-count} variable between the title and allauthor nodes is 2.

In the mining scheme, XList is constructed to store the nodes of XML query trees including their \textit{xcodes} and the number of their occurrences in an XML query tree database. Construction of the XList consists of two steps. In the first step, the path information of an XML query tree is concerned (that is, the XL-Path algorithm), while in the second step, the subtree information of an XML query tree is considered (that is, the XL-Subtree algorithm). In the XL-Path algorithm, the leaf nodes of XML query trees are concerned to record the path information of an XML query tree. If no \textit{xNode} exists in XList, these leaf nodes are stored in the new created \textit{xNodes} of XList; otherwise, their \textit{xcodes} are compared with the variables code of the existing \textit{xNodes}. On the other hand, in the XL-Subtree algorithm, the relationship of a pair of leaf nodes of XML query trees is considered to deal with the subtree information of an XML query tree. If the relationship is not recorded in XList, the \textit{sibling} pointers of \textit{xNodes} are used; otherwise, the number of their occurrences is recorded in the existing variables \textit{s-count}. The following symbols \(T_i, li, (li, l_j), ti, a_i, n_i, \) and \(d_i\) are used in the XL-Path and XL-Subtree algorithms to represent how to record the information of XML query trees in XList. Symbol \(T_i\) represents an XML query tree, \(li\) indicates a leaf node of \(T_i\), and \((x, y)\) denotes the \textit{xcodes} of \(li\). On the other hand, for the data structure XList, symbol \(n_i\) represents a new created \textit{xNode}, \(t_i\) represents the \textit{xNodes} which are not lined by any \textit{parent} pointer of an \textit{xNode}, \(a_i\) indicates an ancestor node of \(t_i\), and \(d_i\) shows a descendant node of an \textit{xNode}.

Lines 2-5 store all of \(T_i\)'s leaf nodes into the new created \textit{xNodes} since there is no \textit{xNode} in XList. Lines 7-28 compare the \textit{xcodes} \((li, l_j)\) with the variable \textit{code} of \(ti\) in XList. Line 10 adds the value 1 to the variables \textit{count} of \(ti\) and all of \(t_i\)'s ancestor nodes \(a_i\) since \(t_i\)'s \textit{code} is the same as the \textit{xcodes} of \(li\). Lines 13-15 store \(li\) into a new created \textit{xNode} \(n_i\) and link \(t_i\)'s \textit{parent} pointer to \(n_i\); since \(li\) is an ancestor node of \(t_i\) and \(t_i\) has no ancestor node. Line 17 adds the value 1 to the variables of node \(a_i\) and all of \(a_i\)'s ancestors since \(a_i\) is the same as \(li\). Lines 19-22 find an \textit{xNode} \(a_i\) which is a descendant node of \(li\), store \(li\) into a new created \textit{xNode} \(n_i\), and insert \(n_i\) between \(a_i\) and \(a_i\)'s parent node. Lines 24-25 store \(li\) into a new created \textit{xNode} \(n_i\) and link \(n_i\)'s \textit{parent} pointer to \(t_i\); since \(li\) is a descendant node of \(t_i\). Finally, Line 27 stores \(li\) into a new created \textit{xNode} \(n_i\) since \(li\) and \(t_i\) have no ancestor-descendant relationship (Figure 5).

For example, suppose that all of the query trees \(T_1, T_2, \ldots, T_5\) are sequential read and processed by the XL-Path algorithm as shown in Figure 6. Firstly, \(T_1\) is read and Lines 2-5 are executed since there is no \textit{xNode} in XList. Thus, the leaf nodes \textit{XML} and \textit{john} of \(T_1\) are stored in the new \textit{xNodes} \(n_1\) and \(n_2\) of XList. Then, \(T_2\) is read and Line 10 is executed since the leaf node \textit{XML} of \(T_2\) is the same as the \textit{xNode} \(n_1\). Therefore, the value 1 is added into the variable \textit{count} of \(n_1\) and results. In consequence, \(T_3\) is read and Lines 13-15 are executed since \(T_3\)'s leaf nodes \textit{title} and \textit{allauthor} are the ancestors of \textit{xNodes} \(n_1\) and \(n_2\) respectively. Thus, two new \textit{xNodes} \(n_3\) and \(n_4\) are created to store the two leaf nodes and \textit{xNodes} \(n_1\) and \(n_2\)'s parent pointers are linked to \(n_3\) and \(n_4\) respectively. Also, the values of variables \textit{count} of \(n_3\) and \(n_4\) are set by the values 3 and 2 which are the sum of the value 1 and those values in variables \textit{count} of \(n_1\) and \(n_2\), respectively. After reading \(T_4\), Lines 2-5 are executed and the new \textit{xNode} \(n_5\) is thus created for \(T_4\)'s leaf node \textit{chapter}. Finally, \(T_5\) is read and Lines 24-25 are executed. The new \textit{xNodes} \(n_6, n_7, n_8\) are created for \(T_5\)'s leaf node \textit{head1}, \textit{head2}, and section2. Also, the parent pointers of \(n_6, n_7,\) and \(n_8\) is linked to \(n_5\).

In Figure 7, Line 3 links the \textit{sibling} pointers between the two leaf nodes \(l_i\) and \(l_j\)'s corresponding \textit{xNodes} \(n_i\) and \(n_j\) in XList. Lines 5-10 add the value 1 to the variables \textit{s-count} between \textit{xNodes} \(n_i\) and \(n_j\).

For example, suppose that all of query trees \(T_1, T_2, \ldots, T_5\) are sequential read and processed by the XL-Subtree algorithm as shown in Figure 7. Firstly, \(T_1\) is read and Lines 3-8 are executed since the relationship between the leaf nodes \textit{XML} and \textit{john} are not recorded in their corresponding \textit{xNodes} \(n_1\) and \(n_2\). Thus, the \textit{sibling} pointer of \(n_1\) is linked to \(n_2\) and the variable \textit{s-count} is set to the value 1. Then, \(T_2\) is read and is not processed since it has no a pair of leaf nodes. In consequence, \(T_3\) is read and Lines 7-8 are executed since \(T_3\)'s leaf nodes \textit{title} and \textit{allauthor} are the ancestors of \textit{xNodes} \(n_1\) and \(n_2\) respectively. Thus, the \textit{sibling} pointer between \textit{xNodes} \(n_3\) and \(n_4\) are created. Also, the value of variable \textit{s-count} is set by the sum of value 1 and the value of \(d_i\)'s \textit{s-count}. In addition, \(T_4\) is read and not to be processed since it has no a pair of leaf nodes. Finally, \(T_5\) is read and then Lines 3-5 are executed to show the result in Figure 8.

An XML frequent pattern mining algorithm for ebXML applications: ebX2Miner

This subsection provides an overview of the \textit{ebX2Miner} algorithm to mine frequent XML query patterns from an XML query tree database for ebXML applications. The \textit{ebX2Miner} is an efficient mining algorithm to discover frequent XML query patterns based on the novel encoding scheme XCode and data structure XList. Figure 9 shows the procedure of the \textit{ebX2Miner} algorithm. The following symbols \(n_i, li, p_i, (c_x, c_y), z_i, temp, n_i, ct, fp,\) and \(fs\) are used to describe the \textit{ebX2Miner} algorithm. In XList,
Algorithm XL-Path \((T_i)\)

Input: An XML query tree \(T_i\)

Output: XList

1. if there is no \textbf{xnode} in XList then
2. create the new \textbf{xnodes} \(n_1, n_2, \ldots, n_i\) for all of \(T_i\)'s leaf nodes \(l_1, l_2, \ldots, l_i\) respectively
3. store the \textbf{xcodes} of nodes \(l_1, l_2, \ldots, l_i\) into the variables \textbf{code} of \textbf{xnodes} \(n_1, n_2, \ldots, n_i\) respectively
4. set the variables \textbf{count} of \textbf{xnodes} \(n_1, n_2, \ldots, n_i\) with the value 1
5. else
6. for each leaf node \(l_i\) of \(T_i\)
7. compare the \(l_i\)’s \textbf{xcodes} \((l_x, l_y)\) with the variable \textbf{code} of each \(t_i\) in XList
8. if \(l_x, l_y\) is the same with \(t_i\)’s \textbf{code} then
9. add value 1 to the \textbf{count} variables of \(t_i\) and all of \(t_i\)'s ancestor nodes \(a_i\)
10. else
11. if \(l_i\) is the ancestor node of \(t_i\) and \(t_i\) has no ancestor node \(a_i\)
12. store the node \(l_i\) into a new created \textbf{xnode} \(n_i\)
13. link the \textbf{parent} pointer of \(t_i\) to \(n_i\)
14. set the value of variable \textbf{count} of \(n_i\) is the sum of that of \(t_i\) with 1
15. if \(l_i\) is an ancestor of \(t_i\) and \(t_i\) has an ancestor \(a_i\) which is the same as \(l_i\)
16. add value 1 to the variable \textbf{count} of \(a_i\) and all of \(a_i\)'s ancestor nodes
17. if \(l_i\) is an ancestor of \(t_i\) and all of \(t_i\)'s ancestor \(a_i\) are different from \(l_i\)
18. find the \textbf{xnode} \(a_i\) which is a descendant node of \(l_i\)
19. store node \(l_i\) into a new created \textbf{xnode} \(n_i\)
20. link the \textbf{parent} pointer of \(n_i\) to \(a_i\)'s \textbf{parent} pointer
21. link the \textbf{parent} pointer of \(a_i\) to \(n_i\)
22. if \(l_i\) is a descendant node of \(t_i\)
23. store node \(l_i\) into a new created \textbf{xnode} \(n_i\)
24. link \(n_i\)'s \textbf{parent} pointer to \(t_i\)
25. add value 1 to the \textbf{count} variables of \(n_i\) and all of \(n_i\)'s ancestor nodes
26. if \(l_i\) and \(t_i\) have no ancestor-descendant relationship
27. store node \(l_i\) into a new created \textbf{xnode} \(n_i\) in XList
28. end if
29. end for
30. end if
31. return XList

Figure 5. Algorithm XL-path.
Figure 6. The XList for the XML query trees in Figure 4 after executing the XL-Path algorithm.

Algorithm XL-Subtree($T_i$)
Input: An XML query tree $T_i$
Output: XList
1. for each pair of leaf nodes $l_i$ and $l_j$ of $T_i$
2. if the relationship between $l_i$ and $l_j$ is not recorded in $xnodes$ $n_i$ and $n_j$
3. link the sibling pointers between $n_i$ and $n_j$
4. if there is no variable $s$-count of the descendant nodes $d_i$ of $n_i$ and $n_j$
5. set the variable $s$-count between $n_i$ and $n_j$ with the value 1
6. else
7. set the variable $s$-count between $n_i$ and $n_j$ with the value which is the sum of
8. the value of variable $s$-count of $d_i$ and value 1
9. else
10. add 1 to the $s$-count variables between $n_i$ and $n_j$
11. end if
12. end for
13. end while
14. return XList

Figure 7. Algorithm XL-Subtree.
shows a set of frequent subtrees.

In Figure 9, firstly, all of XML user query trees in $D$ are read and encoded by the proposed scheme XCode to construct XList. This step is done by the algorithms XL-Path and XL-Subtree. Secondly, the study prunes the infrequent query trees in XList by executing Lines 6-13. Finally, the study enumerates the frequent XML query pattern from XList by executing Lines 14-26.

For example, suppose that the database $D$ has five query trees $T_1$, $T_2$, ..., and $T_5$ and the value of $m$ is 0.4. Firstly, after executing Lines 2-5, the content of XList is shown. Then, Figure 10 shows the results after executing Lines 6-13. Finally, sets $fp$ and $fs$ after executing Lines 14-26 are shown.

COMPARISONS

In this section, there is the comparison of eb$X^2$Miner with other algorithms, including the VBUXMiner (Bei et al., 2009), XQPMiner, XQPMinerTID, and 2PXMiner (Yang et al., 2008) algorithms.

Comparing with VBUXMiner

eb$X^2$Miner is more suitable for ebXML applications in e-commerce than the VBUXMiner algorithm. First, most of XML queries in ebXML applications have the same data structure. However, the VBUXMiner algorithm does not consider the characteristic of the XML queries in ebXML applications and thus merges all of queries into the CGTG tree. Therefore, to obtain the frequent XML query trees, the incomplete information of an XML query tree on the CGTG tree is collected by executing the tree-join process. In contrast, eb$X^2$Miner considers the characteristic of ebXML applications and thus encodes the nodes of XML user query trees. As a result, the path and...
subtree information of an XML query tree are preserved in the leaf nodes’ codes and the tree-join process for producing the frequent query trees can be ignored. For example, the query trees are merged by the VBUXMiner algorithm and result in the CGTG tree as shown in Figure 11. In Figure 11, the incomplete information of a frequent XML query tree is shown and results in the VBUXMiner algorithm to execute the tree-join process or database scans. However, the complete information (that is, path and subtree) of a frequent query tree is preserved by the XCode and XList schemes in ebX2Miner. Therefore, the tree-join process and database scans cannot be used in ebX2Miner for generating frequent XML query trees.

Comparing with XQPMiner, XQPMinerTID, and 2PXMiner

One reason confirms that ebX2Miner may outperform XQPMiner, XQPMinerTID, and 2PXMiner. XQPMiner, XQPMinerTID, and 2PXMiner construct the T-GQPT tree to summarize all of query trees in database D and then generate all of single branch candidate subtrees from the T-GQPT tree. Through tree joining process (that is, constructing data structure ECTree), the single branch candidate subtrees are merged to produce the frequent query trees. Therefore, for ebXML applications, more XML query trees are processed on the T-GQPT tree and thus cost a lot of time to produce frequent XML query trees. In contrast, ebX2Miner encodes the nodes of an XML query tree and thus preserves the path and subtree information of the query tree in the system to reduce time and space costs.

PERFORMANCE STUDY

Two experiments are performed to illustrate the performance under ebX2Miner and VBUXMiner algorithms. Parameters and their settings in the simulation are listed in Table 1. The parameter n denotes the number of XML query trees in the database D, while the parameter s
Table 1. Simulation parameters and settings.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Descriptions</th>
<th>Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>Number of XML query trees</td>
<td>10000 ~ 50000</td>
</tr>
<tr>
<td>S</td>
<td>Minimum supports</td>
<td>3% ~ 8%</td>
</tr>
</tbody>
</table>

Figure 12. The execution time with varying number of XML query trees.

represents the value of minimum support in the system.

The first experiment (Figures 12 and 13) observes the execution time and memory space (Y-axis) of these algorithms under different number of XML query trees (X-axis). The memory space used in ebX²Miner and VBUXMiner is measured by their created nodes in XList and CGTG tree respectively. The specified minimum support $s$ is set to be 5%. ebX²Miner outperforms VBUXMiner on the execution time. Both curves for VBUXMiner and ebX²Miner increase as the number of XML query trees increases. Obviously, ebX²Miner changes slightly as the number of XML query trees increases. In contrast, VBUXMiner changes heavy. One reason could be the high efficiency and stability of the ebX²Miner. VBUXMiner does not consider the path and subtree of XML user query trees in its CGTG tree. Thus, the tree-joining process and database scans are executed to combine this information. As a result, more execution time is used in VBUXMiner for generating the frequent XML query patterns. This is consistent with the experimental result. The used nodes generated from ebX²Miner in XList are less than those from VBUXMiner in CGTG tree. A possible reason is that the XCode scheme encodes the path and subtree information in the nodes of XList and results in a few XML nodes in query trees stored in XList.

The second experiment (Figure 14) observes the execution time (Y-axis) of ebX²Miner and VBUXMiner under different minimum supports (X-axis). The specified number of XML query trees is set to 30000. ebX²Miner outperforms VBUXMiner on the execution time. Both curves for VBUXMiner and ebX²Miner change slightly as the specified minimum support increases. A possible reason is that when the specified minimum support increases, most of the candidate subtrees of ebX²Miner and VBUXMiner are produced from XList and CGTG tree.
respectively. The execution time of ebX²Miner is less than that of VBUXMiner. The reason is that VBUXMiner cost a lot of time to execute the tree-joining process to produce the frequent XML query patterns.

The two experiments as mentioned above show that ebX²Miner has higher mining performance than
This is because by XCode and XList schemes, the path and subtree information are preserved in the leaf nodes of query trees and result in less space and time cost in the ebX2Miner.

Conclusion

This paper presents an efficient mining algorithm ebX2Miner to discover frequent XML query patterns. Unlike the existing algorithms, the study proposes a new idea by encoding XML user query trees (that is, XCode) and thus, stores these codes (that is, XList) to preserve the path and subtree information of query trees. With this idea, it becomes obvious that ebX2Miner is not capable of maintaining all of the user queries and thus takes less execution time and memory space to produce frequent XML query patterns for ebXML applications. The future work in this study includes expanding XML query patterns with repeating-siblings, since ebX2Miner cannot mine the frequent XML query patterns with sibling repetitions.

Conflict of Interests

The author has not declared any conflict of interests.

REFERENCES


