academicJournals

Vol. 8(18), pp. 777-790, 28 September, 2014
DOI: 10.5897/AJBM2011.644
Article Number: 0350C7647676
ISSN 1993-8233

http://www.academicjournals.org/AJBM

African Journal of Business Management

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 specification 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 ebX^{2}Miner, to discover the frequent XML query patterns for ebXML applications. 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 ebX^{2}Miner 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

E-mail: apple@teamail.ltu.edu.tw, apple@mail.ltu.edu.tw. Tel: 886-4-23892088. Fax: 886-4-23895293

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, ebX^{2}Miner, to mine the frequent XML query patterns for ebXML applications in e-commerce. ebX^{2}Miner has the following advantages over the existing approaches. First, ebX^{2}Miner 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, ebX^{2}Miner encodes an XML query tree and stores its nodes' codes to enhance the mining performance. The key concept in ebX^{2}Miner 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 ebX^{2}Miner algorithm. Section 5 compares the ebX^{2}Miner 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 follow 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 trees in these algorithms and degrades their mining performance. Unlike Yang et al. (2008), Bei et al. (2009) accumulate the frequencies of XML candidate query sub trees directly from the CGTG tree by executing the tree-join process. Therefore, Bei et al. (2009) are more efficient than Yang et al. (2008). However, Yang et al. (2008) still cost a lot of system time to execute the tree-join process for merging the path and sub tree information to generate frequent XML query patterns on the CGTG tree.

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.

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 $T_i = <N_i, E_i>$ and 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) Root(t_{ij}) = Root(T_i), where Root(t_{ij}) and 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 $D = \{T_1, T_2, \ldots, T_n\}$, where $T_1, T_2, \ldots, T_n$ 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_{ij} such that for each t_{ij} in *S*, sup(t_{ij}) $\geqq m$ holds, where sup(t_{ij}) is the equation: the number of t_{ij} / the number of XML query trees in *D*.

Definition 1 defines an XML query as a tree. For example, Figure 1 shows an XML query tree T_i of the query to retrieve the *author* elements that have the string value "john" and are descendants of *book* elements that have a child *title* element whose value is "XML".

Definition 2 defines an XML query rooted subtree. It shows the rooted subtrees t_{ij} of the query tree T_i . These rooted subtrees have the same root as the T_i and their edges belong to those of T_i. Note that, in this paper, a rooted subtree t_{ij} with *k* edge is called a *k*-edge t_{ij}. As a result, subtrees (a) and (b) are *1*-edge subtrees, (c), (d), and (e) are *2*-edge subtrees, and (f) is a *3*-edge subtree.

Definition 3 illustrates an XML tree database *D* which contains multiple XML query trees. Each query tree in database *D* represents a transaction associated with its transaction *ID*. For example, in Figure 2, the database $D = <T_1, T_2, T_3, T_4, T_5>$, where T_1, T_2, T_3, T_4, and T_5 are the query trees and with their transaction *ID*s 1, 2, 3, 4, and 5 respectively. In addition, Definition 4 defines the frequent XML query pattern mining problem in this paper.

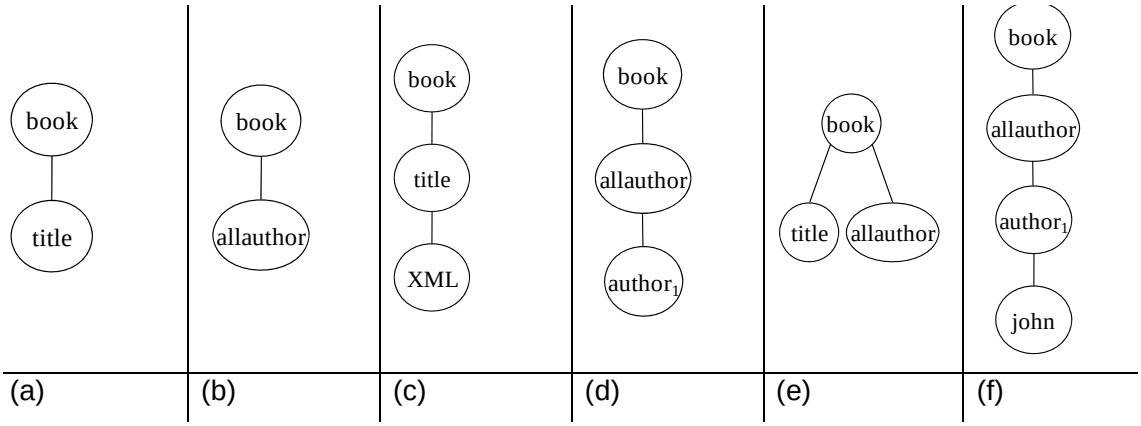


Figure 1. The rooted subtrees of the XML query tree.

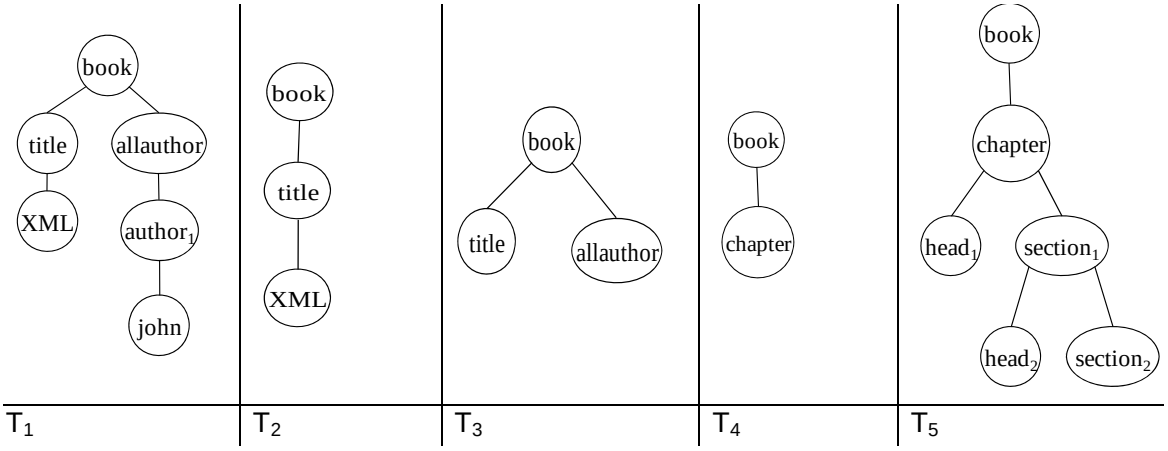


Figure 2. The XML query trees in the database *D*.

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 ebX^{2}Miner 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* expect 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 *fc* 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 *nc* node of its parent node p and its left sibling node l has m descendant nodes with the coordinates (xl, yl), then k's coordinates are (xl+m, yl).

Note that, for simplify, hereafter, the coordinates of a

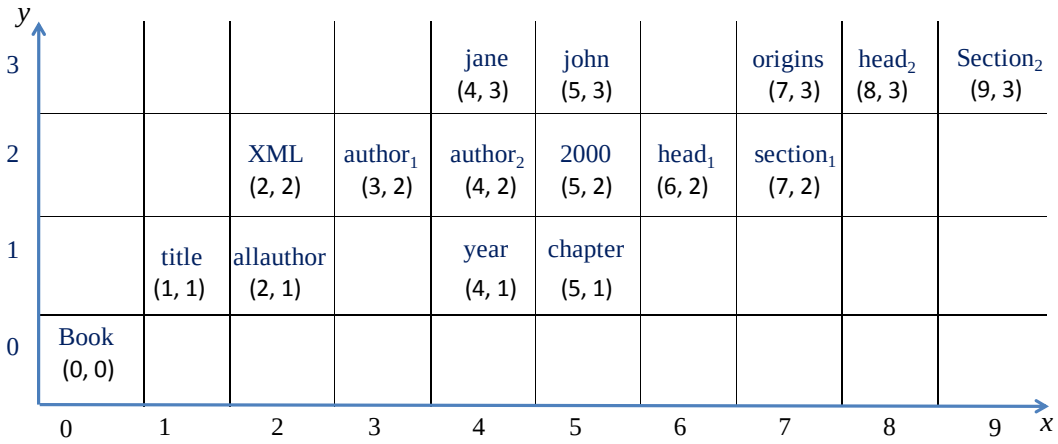


Figure 3. The *xcodes* of the nodes in the XML tree in Figure 2(a).

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*, $author_1$, *john*, *jane*, *2000*, $head_1$, *origins*, and $head_2$ 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*, $author_2$, $section_1$, and $section_2$ 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 f_1 and f_2 in an XML tree T_i with the *xcodes* (x_1, y_1) and (x_2, y_2) respectively, if node f_2 is a child node of f_1, then $y_2 = y_1 + 1$.

Proof: 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_s+m, y_s), where (x_s, y_s) is the *xcode* of f_1's first child node f_s and f_s has m descendant nodes. Thus, if f_2 is the first child node of f_1, $y_2 = y_1+1$. In addition, since $y_2 = y_s$ and $y_s = y_1 + 1$ which result in $y_2 = y_s = y_1 + 1$. As a result, $y_2 = y_1 + 1$.

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

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 T_i.

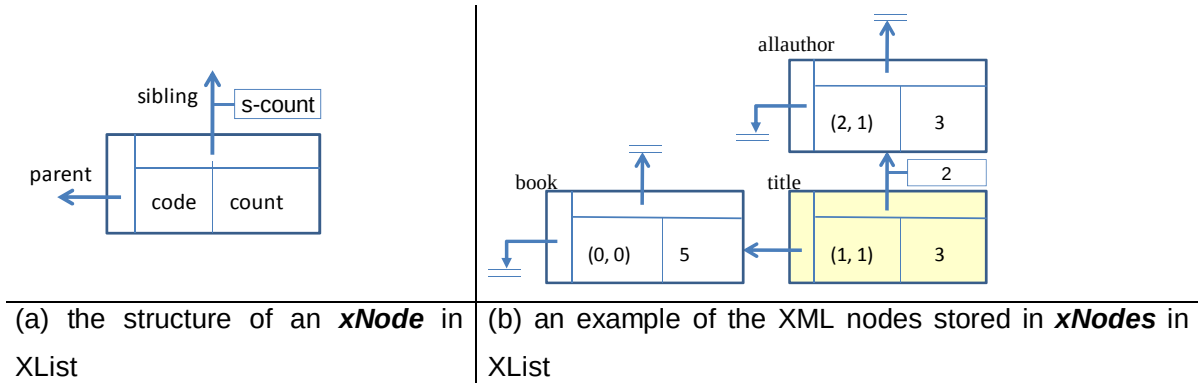
Case 1: Suppose that node *f* is the root node in T_i. According to Rule (1), the *xcode* of *f* is (0, 0). Thus, the value of y is equal to 0. Also, since *f* is the root node, *f*'s level/ is equal to 0. As a result, the value of y is equal to that of l.

Case 2: Suppose that *f* is the *fc* node in T_i. Since *f* is not the root node and with the level l, it has the ancestor nodes $p_0, p_1, .., p_{l-1}$, where p_{l-1} is *f*'s parent node, p_{l-2} is p_{l-1}'s parent node,…, and p_0 is the root node. According to Rule (1), the *xcode* of p_0 is (0, 0). Thus, y_{p0} is equal to 0. Also, according to Lemma 2, p_1's *xcode* $y_{p1} = y_{p0}+1$. Thus, $y_{p1} = y_{p0} + 1 = 0 + 1 = 1$. In consequence, p_2's *xcode* $y_{p2} = y_{p1} + 1 = 1 + 1 = 2$. Therefore, p_{l-1}'s *xcode* $y_{pl-1} = l$-1. Since *f* is the child node of y_{p-1}, *f*'s *xcode* $y = y_{pl-1} + 1 = l - 1 + 1 = l$. 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 T_i. According to *Case 2*, the *fc*'s *xcode* $y_{fc} = l$. In consequence, according to Rule (3), *f*'s *xcode* y is equal to y_{fc}. As a result, $y = y_{fc} = l$ and the value of y is equal to that of *f*'s level l.

Based on *Case 1*, *Case2*, and *Case 3*, we thus prove this lemma.

Lemma 3: For any two nodes f_1 and f_2 in an XML tree T_i with the ***xcodes*** (x_1, y_1) and (x_2, y_2) respectively, if node f_2 is a descendant node of f_1, then both of the values of x_2 and y_2 are bigger than those of x_1 and y_1 respectively.



(a) the structure of an ***xNode*** in XList

(b) an example of the XML nodes stored in ***xNodes*** in XList

Figure 4. The structures and contents of *xNodes* in XList.

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_s+m, y_s), where (x_s, y_s) is the ***xcode*** of f_1's first child node f_s and f_s 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 $x_2 = x_s + m$, $y_2 = y_s$, $x_s = x_1 + 1$, and $y_s = y_1 + 1$ which result in $x_2 >= x_s > x_1$ and $y_2 > y_s > y_1$. 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_a which is a child node of f_1. According to *Case 1*, node f_a's ***xcode*** $x_{fa} > x_{f1}$ and $y_{fa} > y_{f1}$. Also, since f_2's ***xcode*** $x_{f2} > x_{fa}$ and $y_{f2} > y_{fa}$, they result $x_{f2} > x_{f1}$ and $y_{f2} > y_{f1}$.

Based on *Case 1* and *Case2*, we thus prove this lemma.

Lemma 4: For any node *f* in an XML tree T_i, 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_i.

Case 1: Suppose that node *f* is the root node in T_i. 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_i, where p_n is *f*'s parent node, p_{n-1} is p_n'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_1'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_1's *xcodes* are bigger than 0. In consequence, according to Rules (2) or (3), the values of the *xcodes* in p_2, p_3, …, p_n are thus bigger than 0. Since, according to Rule (2), the values in *f*'s *xcode* are the sum of those of p_n's *xcode* with 1. As a result, the values in *f*'s *xcodes* are bigger than 0.

Case 3*:** Suppose that *f* is the *nc* node and thus has a sibling node *fc* in T_i. 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 XML node to be stored in an *xNode* of XList. Variable *code* is used to store an XML node's *xcode*, and variable *count* is used to store the number of occurrences of the XML node of a user query tree in a database. Also, two pointers *parent* and *sibling* are used to link the XML node's parent and sibling nodes respectively. Furthermore, the *sibling* pointer has a variable *s-count* to record the number of occurrences of the relationships between two XML nodes. For example, the *title* node is shown in the query trees T_1, T_2, and T_3 in the database *D*. Through the XCode scheme, the *xcode* of the *title* node is (1, 1) and it can be stored in an *xNode* of XList; the *title*

node's parent and sibling nodes are the *book* and *allauthor* nodes and linked by its *parent* and *sibling* pointers respectively. The ***xcodes*** of nodes *book* and *allauthor* are (0, 0) and (2, 1) respectively, while the numbers of occurrences of those nodes are 5 and 3 respectively. In addition, the *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 ***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 *xNode* exists in XList, these leaf nodes are stored in the new created *xNodes* of XList; otherwise, their *xcodes* are compared with the variables *code* of the existing *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 *sibling* pointers of *xNodes* are used; otherwise, the number of their occurrences is recorded in the existing variables *s-count*. The following symbols T_i, l_i, (l_x, l_y), t_i, 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, l_i indicates a leaf node of T_i, and (x_l, y_l) denotes the *xcode* of l_i. On the other hand, for the data structure XList, symbol n_i represents a new created *xNode*, t_i represents the *xNodes* which are not lined by any *parent* pointer of an *xNode*, a_i indicates an ancestor node of t_i , and d_i shows a descendant node of an *xNode*.

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

For example, suppose that all of the query trees T_1, T_2, …, and 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 *xNode* in XList. Thus, the leaf nodes *XML* and *john* of T_1 are stored in the new *xNodes* n_1 and n_2 of XList. Then, T_2 is read and Line 10 is executed since the leaf node *XML* of T_2 is the same as the *xNode* n_1. Therefore, the value 1 is added into the variable *count* of n_1 and results. In consequence, T_3 is read and Lines 13-15 are executed since T_3's leaf nodes *title* and *allauthor* are the ancestors of *xNodes* n_1 and n_2 respectively. Thus, two new *xNodes* n_3 and n_4 are created to store the two leaf nodes and *xNodes* n_1 and n_2's parent pointers are linked to n_3 and n_4 respectively. Also, the values of variables *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 *count* of n_1 and n_2, respectively. After reading T_4, Lines 2-5 are executed and the new *xNode* n_5 is thus created for T_4's leaf node *chapter*. Finally, T_5 is read and Lines 24-25 are executed. The new *xNodes* n_6, n_7, and n_8 are created for T_5's leaf node $head_1$, $head_2$, and $section_2$. Also, the parent pointers of n_6 , n_7, and n_8 is linked to n_5.

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

For example, suppose that all of query trees T_1, T_2, …, and 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 *XML* and *john* are not recorded in their corresponding *xNodes* n_1 and n_2. Thus, the *sibling* pointer of n_1 is linked to n_2 and the variable *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_2's leaf nodes *title* and *allauthor* are the ancestors of *xNodes* n_1 and n_2 respectively. Thus, the *sibling* pointer between *xNodes* n_3 and n_4 are created. Also, the value of variable *s-count* is set by the sum of value 1 and the value of d_i's *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: ebX^{2}Miner

This subsection provides an overview of the ebX^{2}Miner algorithm to mine frequent XML query patterns from anXML query tree database for ebXML applications. The ebX^{2}Miner 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 ebX^{2}Miner algorithm. The following symbols n_i, t_i, p_i, (c_x, c_y), z_i, *temp_n*, *ct*, *fp*, and *fs* are used to describe the ebX^{2}Miner algorithm. In XList,

Algorithm *XL-Path* (T_i)

Input: An XML query tree T_i

Output: XList

```
if there is no xnode in XList then
create the new xnodes n1, n2,…, ni for all of Ti's leaf nodes l1, l2,…, li respectively
store the xcodes of nodes l1, l2,…, li into the variables code of xnodes n1, n2,…, ni
respectively
set the variables count of xnodes n1, n2,…, ni with the value 1
else
for each leaf node li of Ti
compare the li's xcode (lx, ly) with the variable code of each ti in XList
if xcode (lx, ly) is the same with ti's code then
add value 1 to the count variables of ti and all of ti's ancestor nodes ai
  else
if li is the ancestor node of ti and ti has no ancestor node ai
    store the node li into a new created xnode ni
    link the parent pointer of ti to ni
    set the value of variable count of ni is the sum of that of ti with 1
if li is an ancestor of ti and ti has an ancestor ai which is the same as li
    add value 1 to the variable count of ai and all of ai's ancestor nodes
if li is an ancestor of ti and all of ti's ancestor ai are different from li
   find the xnode ai which is a descendant node of li
   store node li into a new created xnode ni
   link the parent pointer of ni to ai's parent pointer
   link the parent pointer of ai to ni
 if li is a descendant node of ti
   store node li into a new created xnode ni
   link ni's parent pointer to ti
   add value 1 to the count variables of ni and all of ni's ancestor nodes
  if li and ti have no ancestor-descendant relationship
   store node li into a new created xnode ni in XList
 end if
end for
 end if
return XList
```

Figure 5. Algorithm XL-path.

symbol ni illustrates an xNode, ti indicates the node which has no descendant node, pi represents ti's parent node, (cx, cy) represents the code in ti, zi indicates the sibling node of ti, and temp_n represents a temp xNode. Symbol ctz indicates a cross subtree of nodes ti and zi. In addition, symbol fp indicates a set frequent path, while fs

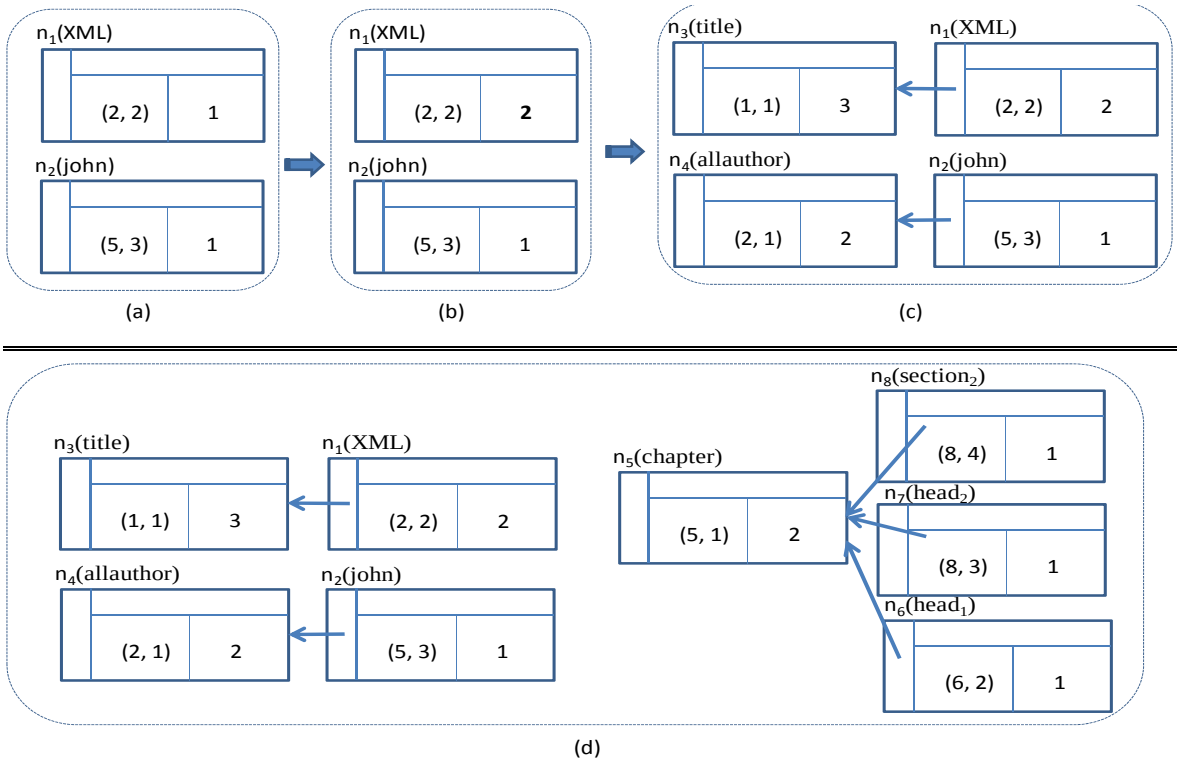


Figure 6. The XList for the XML query trees in Figure 4 after executing the XL-Path algorithm.

Algorithm *XL-Subtree*(T_i)

Input: An XML query tree T_i

Output: XList

```
1     for each pair of leaf nods 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.

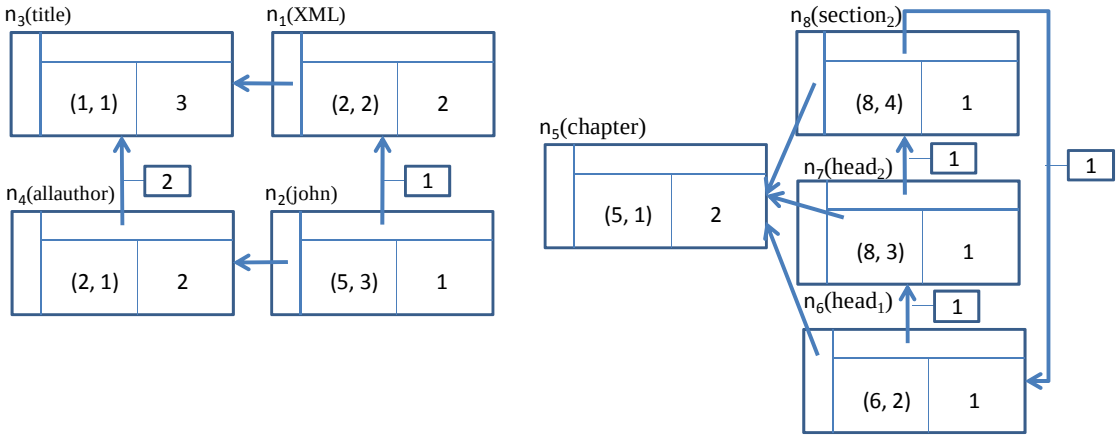


Figure 8. The XList for the XML query trees in Figure 4 after executing the XL-Subtree algorithm.

```
Algorithm ebX²Miner(D, m)
Input: A set of query trees in D; specified minimum su
Output: A set of frequent query subtrees
1  /*scan the database D to construct XList;*/
2        for each T_i in the database D
3         XList = XL-Path(T_i);
4         XList = XL-Subtree(T_i);
5        end-for;
6  /* remove the infrequent nodes from XList; */
7  for each xnode n_i in XList
8   if the value of count is small than m
9    delete the xnode n_i
10   delete n_i's sibling pointer
11   delete all of n's descendant nodes
12  end if
13  end for
14  /* generate the frequent subtrees from XList; */
15  for each xnode t_i with xcode (c_x, c_y) in XList
16   while c_x > 0
17    add a path (p_i, t_i) into set fp
18    if t_i has the sibling node z_i
19      add the cross subtree ctz into set fs
20    end if
21    set temp_n is p_i
22    set p_i is the parent of p_i
23   end while
24   delete t_i
25   delete t_i's sibling pointer
26  end for
```

Figure 9. Algorithm ebX²Miner.

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 ebX²Miner with other algorithms, including the VBUXMiner (Bei et al., 2009), XQPMiner, XQPMinerTID, and 2PXMiner (Yang et al., 2008) algorithms.

Comparing with VBUXMiner

ebX²Miner 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, ebX²Miner considers the characteristic of ebXML applications and thus encodes the nodes of XML user query trees. As a result, the path and

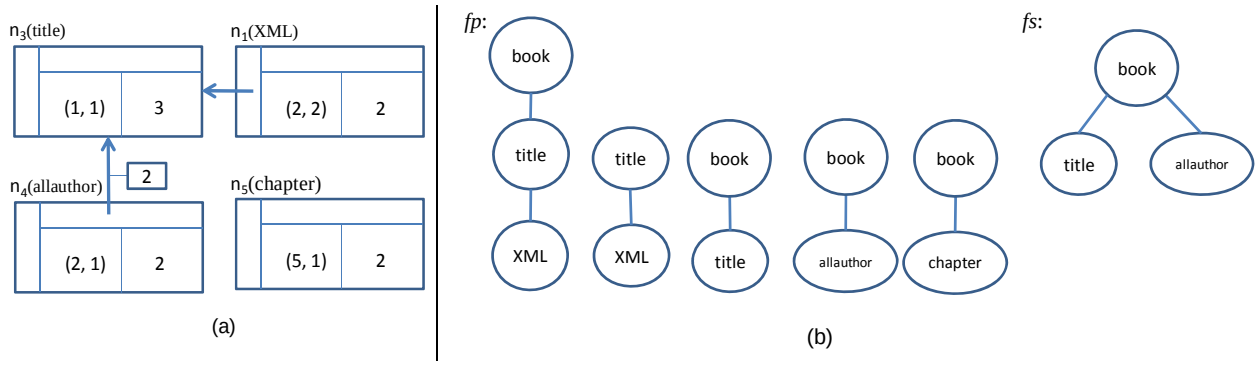


Figure 10. The frequent query patterns for the XML query trees.

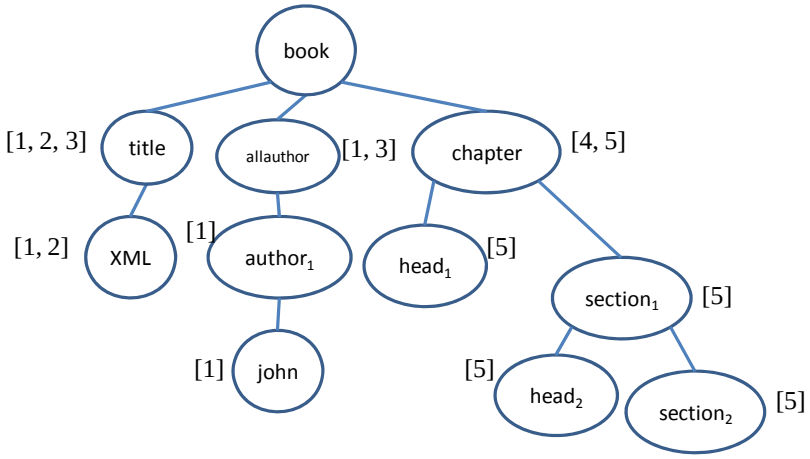


Figure 11. The CGTG tree of the query trees in database *D*.

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 ebX^{2}Miner. Therefore, the tree joining process and database scans cannot be used in ebX^{2}Miner for generating frequent XML query trees.

Comparing with XQPMiner, XQPMinerTID, and 2PXMiner

One reason confirms that ebX^{2}Miner 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, ebX^{2}Miner 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 ebX^{2}Miner 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.

Parameters	Descriptions	Settings
n	Number of XML query trees	10000 ~ 50000
S	Minimum supports	3%~8%

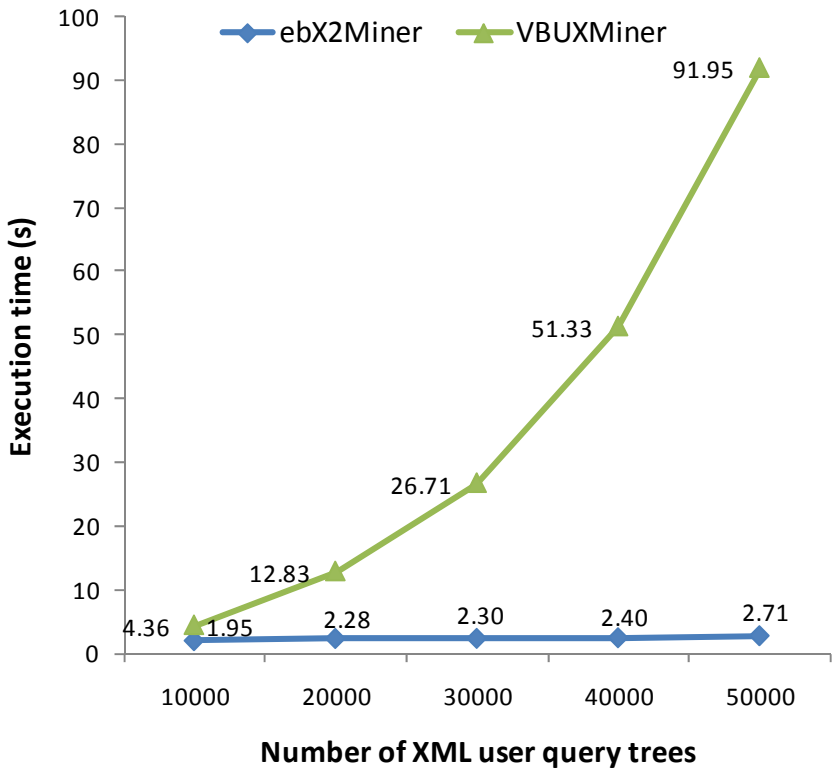


Figure 12. The execution time with varying number of XML query trees.

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 ebX^{2}Miner and VBUXMiner is measured by their created nodes in XList and CGTG tree respectively. The specified minimum support *s* is set to be 5%. ebX^{2}Miner outperforms VBUXMiner on the execution time. Both curves for VBUXMiner and ebX^{2}Miner increase as the number of XML query trees increases. Obviously, ebX^{2}Miner 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 ebX^{2}Miner. 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 ebX^{2}Miner in XList are less than those from VBUXMinr 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 ebX^{2}Miner and VBUXMiner under different minimum supports (X-axis). The specified number of XML query trees is set to 30000. ebX^{2}Miner outperforms VBUXMiner on the execution time. Both curves for VBUXMiner and ebX^{2}Miner 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 ebX^{2}Miner and VBUXMiner are produced from XList and CGTG tree

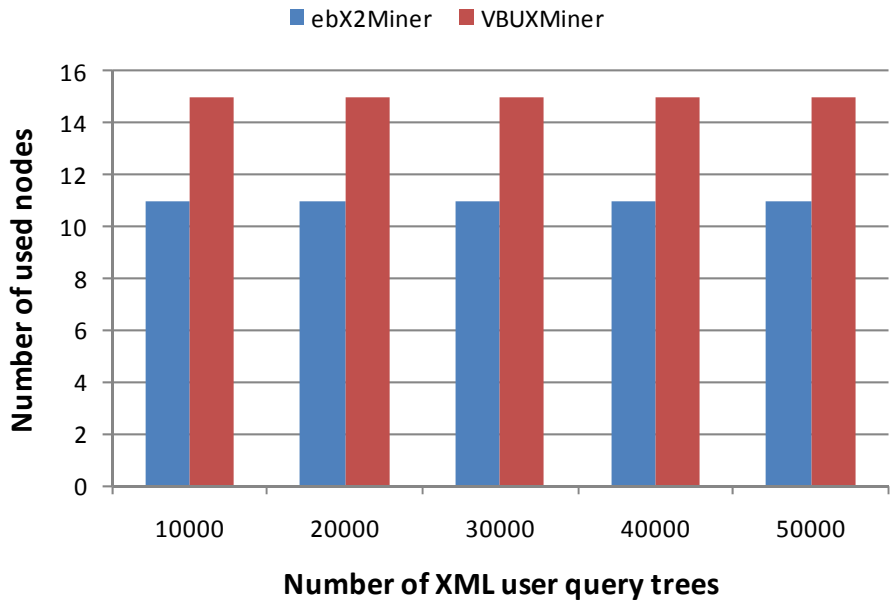


Figure 13. The enumerated nodes with varying number of XML query trees.

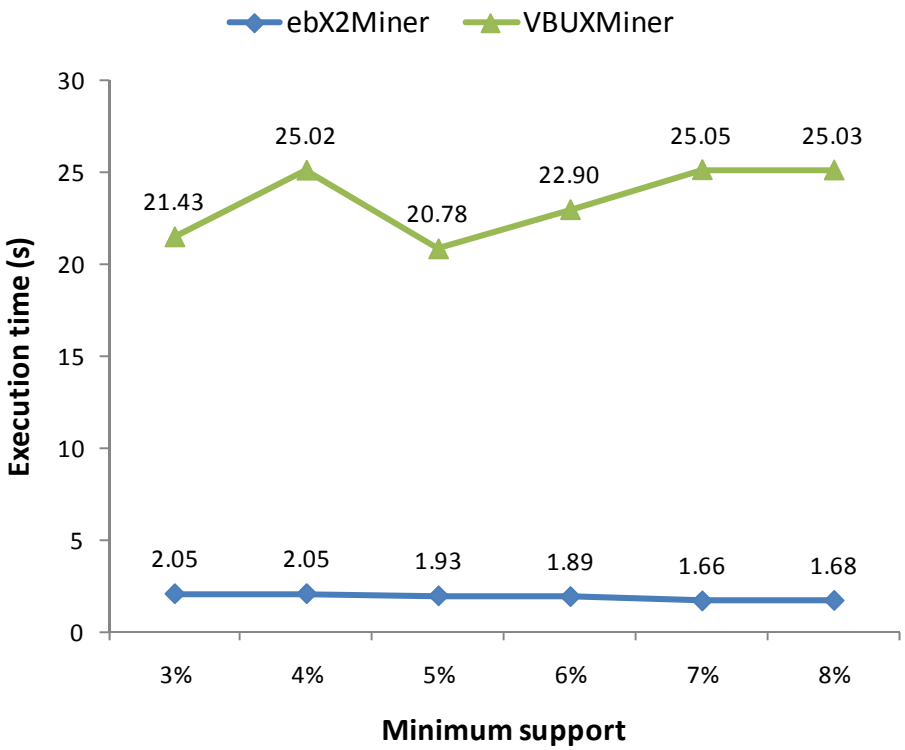


Figure 14. The execution time with varying minimum supports.

respectively. The execution time of ebX^{2}Miner 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 ebX^{2}Miner has higher mining performance than

VBUXMiner. This is because by XCode and XList schemes, the path and subtree information are preserve in the leaf nodes of query trees and result in less space and time cost in the ebX^{2}Miner.

Conclusion

This paper presents an efficient mining algorithm ebX^{2}Miner 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 ebX^{2}Miner 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 ebX^{2}Miner cannot mine the frequent XML query patterns with sibling repetitions.

Conflict of Interests

The author has not declared any conflict of interests.

REFERENCES

Bei Y, Chen G, Dong J, Chen K (2008). Buttom-up Mining of XML Query Patterns to Improve XML Querying. J. Zhejiang University. 9(6):744-757.

Bei Y, Chen G, Shou L, Li X, Dong J (2009). Buttom-up Discovery of Frequent Rooted Unordered Subtrees. Information Sci. 179(1-2):70-88.

Bio BM ebXML: An Electronic Business Scenario. (2003). Available from: http://www.developer.com/xml/article.php/2234201.

Boag SXQ (2010). Available from: http://www.w3.org/XML/Query.

Chen L, Bhowmick SS, Chia LT (2006). FRACTURE-Mining: Mining Frequently and Concurrently Mutating Structures from Historical XML Documents. Data Knowl. Eng. 59(2):517-524.

Clark JXML Path Language (XPath) 2.0 (1999). Available from: http://www.w3.org/TR/2007/REC-xpath20-20070123/

Cunningham LA (2005). Language, Deals and Standards: The Future of XML Contracts. Washington University Law Review.

Green PF, Rosemann M, Indulska M (2005). Ontological Evaluation of Enterprise Systems Interoperability Using ebXML. IEEE transactions on knowledge and data engineering. 17(5):713-725.

Gu MS, Hwang JH, Ryu KH (2007). Frequent XML Query Pattern Mining based on FP-TRee. Proceedings of the 18th International Conference on Database and Expert Systems Applications. pp.555-559.

Kim H (2002). Conceptual Modeling and Specification Generation for B2B Business Processes based on ebXML. SIGMOD Record. 31(1):37-42.

Kwon J, Rao P, Moon BSL (2008). Value-based Predicate Filtering of XML Documents. Data Knowl. Eng. 67(1):51-73.

Lu J, Ling TW, Chan CY, Chen T (2005). From Region Encoding to Extended Dewey: on Efficient Processing of XML Twig Pattern Matching. Proceedings of the 31st International Conference on Very Large Databases pp.193-204

Moberg DO (2007). Available from: http://www.oasis-open.org.

Raj A, PS K (2007). Branch Sequencing Based XML Message Broker Architecture. IEEE 23rd International Conference on Data Engineering (ICDE) pp.656-665.

XML. Available from: http://www.w3.org/XML.

Yang LH, Lee ML, Hsu W, Acharya S (2003). Mining Frequent Query Patterns from XML Queries. In Proceedings of the Eight International Conference on Database Systems for Advanced Applications (DASFAA) pp.75-87.

Yang LH, Lee ML, Hsu W, Huang D, Wong L (2008). Efficient Mining of Frequent XML Query Patterns with Repeating-siblings. Inform. Software Technol. 50(5):375-389.