Mining Frequent User Query Patterns from XML Query Streams

Tsui-Ping Chang
Department of Information Technology, Ling Tung University, Taiwan

Abstract: *An XML query stream is a massive and unbounded sequence of queries that are continuously generated at a fast speed from users over the Internet. Compared with traditional approaches of mining frequent user query patterns in static XML query databases, pattern mining in XML query streams is more challenging since several extra requirements need to be satisfied. In this paper, a mining algorithm is proposed to discover frequent user query patterns over an XML query stream. Unlike most of existing algorithms, the proposed algorithm works based on a novel encoding scheme. Through the scheme, only the leaf nodes of XML query trees are considered in the system and result in higher mining performance. The performance of the proposed algorithm is tested and analyzed through a series of experiments. These experiment results show that the XSM outperforms other algorithms in its execution time.*

Keywords: *Frequent XML query pattern, XML query stream mining, encoding scheme, database.*

Received May 25, 2012; accepted March 23, 2013; published online February 26, 2014

1. Introduction

XML query stream mining [1, 4, 11] has become a more and more attractive research area in recent years due to XML [8] has become the de facto standard language for information exchange over the Internet. An XML query stream is a massive and unbounded sequence of queries that are continuously generated from users to query XML data over the Internet. Due to this reason, as compared with traditional XML query pattern mining [5, 10, 12] in static databases, pattern mining in XML query streams has several requirements that need to be satisfied. For example, each user query in the XML query stream can be examined at most one time, that is, multiple scans of data source is infeasible.

An XML query stream [2, 3, 6, 13] comprises a continuous sequence of user queries which can be represented by multiple trees. Document data represented in XML comprise a sequence of possibly nested tags which can be expressed by a tree structure. Since XML data can be modeled as a tree, XML user queries are treated as trees. For example in Figures 1 and 2, they show an XML tree and its corresponding XML query tree. As an example of the XML tree depicted in Figure 1, an XML element enclosed within a pair of an opening tag and a closing tag is denoted by its tag name with a suffix number for distinguishing itself from other elements with the same tag name. Therefore, XML user queries (i. e., XPath [9]) typically specify patterns of selection predicates on multiple elements that have some specified tree-structured relationships. The primitive tree-structured relationships are parent-child and ancestor-descendant.

For example in Figure 2, it shows a query tree which is modeled by the XPath expression: Book [title = 'XML']/ allauthor/ author [.='john']. This expression matches author elements that: Have the string value "john"., are descendants of book elements that have a child title element whose value is "XML".

Mining frequent XML user query patterns may be used to enhance the query performance of XML streams. Frequent XML query patterns can be used to design an index mechanism or cache the results of these patterns to reduce the unnecessary computation and thus enhance the query performance. Using frequent XML query patterns, the features (i. e., contents and structures) of query results (i. e., the fragments of XML data in steams) are discovered and thus a suitable index mechanism can be designed. On the other hand, frequent XML query patterns can be used to support for storing a collection of XML data's fragments which are the answers of XML query patterns into a cache.

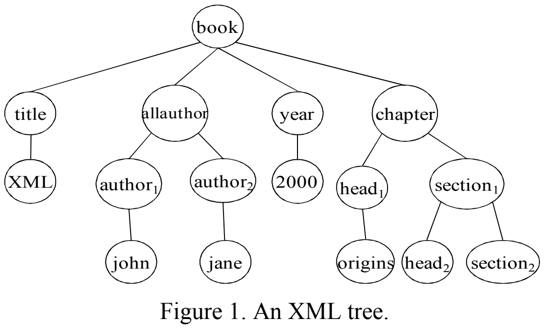

Figure 1. An XML tree.

Several methods [1, 4, 11] have been proposed for mining frequent user query patterns over an XML query stream. XQSMinerI is a one pass algorithm to find out frequent user query patterns from XML query streams. XQSMinerII provides some control over the error bound in the estimate. XQSMinerI treats the stream of XML query trees as a stream of batches. This is because it is very expensive to enumerate frequent query patterns for all of query trees in a stream. In addition, XQSMinerII adopts the framework [6] and operates with two parameters, support σ and error ε, and processes the XML query stream in batch. The incoming stream is conceptually divided into buckets and in every batch it processes a number of buckets of query trees, where the number of buckets is dependent on the size of buffer. The local mining result of every batch is integrated into a data structure DTS which is a set of entries of potential frequent query patterns.

In this paper, an efficient algorithm (namely XSM) is proposed to mine frequent query patterns from XML query streams. XSM utilizes the framework [6] for approximate frequency count of query patterns. In XSM, an encoding scheme (namely XSCode) to represent an XML tree with its corresponding user query trees is proposed. XSCode is more space-efficient since it preserves the structure information of an XML query tree by only recording the codes of its leaf nodes. Through these codes, the frequent query patterns are enumerated efficiently from an XML query stream in XSM.

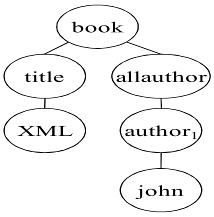

Figure 2. An XML query tree.

The rest of this paper is organized as follows: Section 2 illustrates a tree model of XML data. Section 3 describes the details of XSM. Section 4 shows the experimental results, and section 5 illustrates the conclusion and further work.

2. Preliminaries

- *Definition 1*: An XML query can be modeled as an unordered tree $T = <N, E>$, where N is the node set, and E is the edge set. Nodes $n \in N$ represent the elements, attributes, and string values in an XML query, and edges $e \in E$ represents the parent-child relationships denoted by "/".
- *Definition 2*: Given an XML query tree $T = <N, E>$ and an XML query rooted subtree $t_i = <N_i, E_i>$. t_i is considered to be the i^{th} rooted subtree of T iff there exists:

$$Root(t_i) = Root(T) \quad (1)$$

Where $Root(t_i)$ and $Root(T)$ are the functions which return the root nodes of t_i and T respectively.

$$N_i \subseteq N, E_i \subseteq E \quad (2)$$

- *Definition 3*: An XML query stream, $XQS = <T_1, T_2, ..., T_n>$, where n is the length of current stream, that is, the number of query trees seen so far.
- *Definition 4*: Given an XML query stream XQS, a minimum support value σ ranging from (0, 1], and an error parameter ε ranging from (0, 1], construct an algorithm to produce a set of frequent XML query trees t_i f along with their estimated frequencies on request. For any XML query tree t_i, let $count_{true}(t_i)$ denote the true frequency of t_i, and $count(t_i)$ denote the estimated frequency of t_i. The t_i produced has the following guarantees:

  1. For any t_i in the current stream, if $count_{true}(t_i) \geq \sigma \times n$, then we have $t_i \in f$.
  2. For any t_i that is considered to be frequent, we have $count_{true}(t_i) \geq (\sigma - \varepsilon) \times n$.
  3. For any t_i, its estimated frequency $count(t_i) \geq count_{true}(t_i) - \varepsilon \times n$.

Definition 1 defines an XML query as a tree. Definition 2 defines an XML query rooted subtree. Definition 3 illustrates an XML query stream XQS which contains multiple XML query trees. Each query tree in XQS represents a transaction associated with its transaction ID. For example in Figure 3, the XML query stream $XQS = <T_1, T_2, ..., T_{100}, T_{101}, T_{102}, T_{103}, T_{104}, T_{105}, ..., T_{200}>$, where $T_1, T_2, ..., T_{100}, T_{101}, T_{102}, T_{103}, T_{104}, T_{105}, ..., T_{200}$ are the user query trees and with their transaction IDs 1, 2, …, 100, 101, 102, ..., 200 respectively. Also, the length of XQS is equal to 200.

3. Mining Frequent Query Pattern Over XML Query Streams

3.1. XSCode

XSCode encodes the nodes of an XML tree in an xy coordinate system where xy are the coordinates of the two-dimensional space. The following symbols T, r, k, p, l, fc, and nc are used to represent the nodes in an XML tree. Symbol T represents an XML tree, r indicates the root node in T, k represents a node in T, 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 represent the child node of k expect the first child fc. The encoding rules are described for the nodes in an XML tree T and listed as follows:

1. For an XML tree T, 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, 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, if k is the nc node of its parent node p and its left sibling node l has m descendant nodes with the coordinates (x_l, y_l). If $m = 0$ then k's coordinates are $(x_l + 1, y_l)$, otherwise, k's coordinates are $(x_l + m, y_l)$.

Note that, hereafter; the coordinates of an XML node based on XSCode are namely *xscode*.

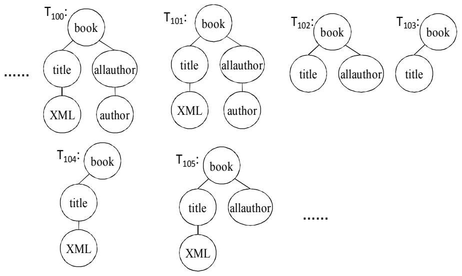

Figure 3. The XML query stream XQS.

- *Example 1*: Consider the XML tree in Figure 1. Suppose that all of nodes in the tree are encoded by the rules of XSCode. The *xscodes* of these nodes are shown in Figure 4. According to Rule 1, the root node *book* in the XML tree in Figure 1 is set on the origin and its *xscode* 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 *xscodes* 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 *xscodes* are (2, 1), (4, 1), (5, 1), (4, 2), (7, 2), and (9, 3) respectively.

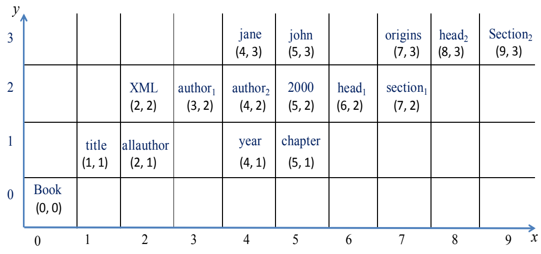

Figure 4. The xscodes of the nodes in Figure 1.

Derived from the XSCode encoding rules, Lemmas 1 and 2 show the features of *xscodes* of an XML tree.

- Lemma 1: For any node f in an XML tree T_i, if f's *xscode* 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 *xscode* of f is (0, 0). Thus, the value of y is equal to 0. Also, since f is the root node, f's level l 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 *xscode* of p_0 is (0, 0). Thus, y_{p0} is equal to 0. Also, according to Lemma 2, p_1's *xscode* $y_{p1} = y_{p0} + 1$. Thus, $y_{p1} = y_{p0} + 1 = 0 + 1 = 1$. In consequence, p_2's *xscode* $y_{p2} = y_{p1} + 1 = 1 + 1 = 2$. Therefore, p_{l-1}'s *xscode* $y_{pl-1} = l - 1$. Since f is the child node of y_{p-1}, f's *xscode* $y = y_{pl-1} + 1 = l - 1 + 1 = l$. 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 *xscode* $y_{fc} = l$. In consequence, according to Rule 3, f's *xscode* 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 2*: For any two nodes f_1 and f_2 in an XML tree T_i with the *xscodes* (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.
- *Proof*: We prove 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 *xscode* (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 *xscode* 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 *xscode* $x_{fa} > x_{f1}$ and $y_{fa} > y_{f1}$. Also, since f_2's *xscode* $x_{f2} > x_{fa}$ 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.

3.2. Functions X-Path and X-Subtree

In algorithm XSM, the path and subtree information of query trees in XML query streams are firstly considered by functions X-Path and X-Subtree. Symbols T, l_i, w, ε, β, $B_{current}$, t_i, a_i, $count_i$, $error_i$, $sibling_{ij}$, $s\text{-}count_{ij}$, and d_i are used in X-Path and X-Subtree. Symbol T represents an XML query tree, l_i indicates a leaf node of T, t_i represents the nodes which are stored in the system, a_i indicates an ancestor node of t_i, $count_i$ indicates the frequencies of t_i, $error_i$ shows t_i's frequencies which are not recorded in the system, $sibling_{ij}$ indicates the sibling relationship between nodes t_i and t_j, $s\text{-}count_{ij}$ shows the frequencies of the sibling relationship between t_i and t_j, and d_i shows a descendant node of t_i. On the other hand, symbol ε denotes an error parameter. The incoming XML query stream is conceptually divided into buckets of $w=\left\lceil \frac{1}{\varepsilon} \right\rceil$

query trees. The buckets are labeled with bucket IDs starting from 1. The current bucket ID is denoted as $B_{current}$, whose value is $\lceil \frac{n}{w} \rceil$. Also, the number of buckets in the main memory for the XML query stream is denoted as β. In function X-Path, the leaf nodes of XML query trees are concerned to record the path information of an XML query tree. If no node stored in the system, the leaf nodes of an XML user query tree are stored; otherwise, their *xscodes* are compared with those of nodes t_i. In addition, if $count_i > \beta$ and $B_{current} > \beta$ in the system, the value of ($B_{current}$ - β) are set into their variables $error_i$. In function X-Subtree, 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.

- *Example 2*: Suppose that all of the query trees T_1, T_2, …, and T_{200} in an XML query stream in Figure 3 are sequential read and processed by function X-Path. Also, suppose that query trees T_1, T_2, …, T_{99} have been processed by X-Path and result in the stored nodes t_i as shown in Figure 5. Suppose that the error parameter ε is equal to 0.1 and the batch size is equal to 100. Therefore, a bucket has 10 (1/ε=1/0.1) query trees, β is equal to 10 ($\lceil \frac{100}{10} \rceil$=10), and $B_{current}$ is equal to 10 ($\lceil \frac{99}{10} \rceil$=10). Firstly, T_{100} is read and Lines 7-11 are executed since the leaf node *XML* of T_{100} is the same as the node t_1. Therefore, the value 1 is added into the variables $count_1$ and $count_3$ of t_1 and t_3 respectively. Also, Lines 27-28 are executed and the new node t_5 is inserted between t_4 and t_2 since the leaf node *author* is a parent of t_2. Then, T_{101} is read and the value of $B_{current}$ is changed to 11 (i. e., $\lceil \frac{101}{10} \rceil$=11). Therefore, Lines 8-9 are executed since the values of variables $conut_1$ and $count_5$ of t_1 and t_5 (i. e., values 26 and 86 respectively) are bigger than that of β (i. e., the value 10) and the value of $B_{current}$ (i. e., the value 11) is bigger than that of β. As a result, the variables $error_1$, $error_3$, $error_4$, and $error_5$ of t_1, t_3, t_4, and t_5 respectively are set to value 1 (i.e., $B_{current}$ - β = 11 -10). In consequence, T_{102} is read and Lines 20-24 are executed since the leaf nodes *title* and *allauthor* of T_{102} are the same as the nodes t_3 and t_4 respectively. Therefore, the value 1 is set into the variables $error_3$ and $error_4$ of t_3 and t_4 respectively. Then, T_{103} is read and Lines 21-25 are executed since T_{103}'s leaf node *title* is the ancestor of node t_1. Thus, the value of variable $error_3$ of t_3 is set by the value 1.

```
Function X-Path (T, ε, size)
Input: An XML query tree T
Output: nodes $t_i$
1  if there is no node stored in the system
2    store the nodes $l_i$ as $t_i$ and set variables $count_i$ with 1
3  else
4    for each leaf node $l_i$ of T
5      compare the $l_i$'s xscode with that of each $t_i$
6      if $l_i$'s xscode is the same with that of $t_i$ then
7        if (($t_i$'s $count_i$) ≥β) and ($B_{current}$>β) then
8          $t_i$'s $error_i$ =$B_{current}$ – β
9          all of $t_i$'s ancestor nodes $a_i$'s $error_i$= $B_{current}$ – β
10       else
11         add 1 to $count_i$ of $t_i$ and all of ancestor nodes $a_i$
12     else
13       if $l_i$ is $t_i$'s ancestor and their xscodes do not
14       stratify Lemma 2, and $t_i$ has no ancestor $a_i$
15         store the node $l_i$ as a parent node $p_i$ of $t_i$
16         set the value of $count_i$ of $p_i$ is the sum of that of
17         $t_i$ with value 1
18       if (($p_i$'s $count_i$) ≥β) and ($B_{current}$>β) then
19         $p_i$'s $error_i$ =$B_{current}$ – β
20         all of $p_i$'s ancestor nodes $a_i$'s $error_i$= $B_{current}$ – β
21          if $l_i$ is an ancestor node of $t_i$ and $t_i$ has an
22          ancestor node $a_i$ which is the same as $l_i$
23         if (($t_i$'s $count_i$) ≥β) and ($B_{current}$>β) then
24          $t_i$'s $error_i$ =$B_{current}$ – β
25          all of $t_i$'s ancestor nodes $a_i$'s $error_i$= $B_{current}$ – β
26           else
27             add value 1 to $count_i$ of $a_i$ and all of $a_i$'s
28             ancestor nodes
29       if $l_i$ is an ancestor node of $t_i$ and all of $t_i$'s
30       ancestor nodes $a_i$ are different from $l_i$
31         find a $a_i$ which is a child node of $l_i$
32         set node $l_i$ as a parent node $p_i$ of $a_i$'s parent
33       if $l_i$ is a descendant node of $t_i$
34         store node $l_i$ into a new created node $c_i$
35         set node $c_i$ as a child node of $t_i$
36         add 1 to $count_i$ and all of $c_i$'s ancestors
37       if $l_i$ and $t_i$ have no ancestor-descendant
38       relationship
39         store $l_i$ into a new created node in system
40       end if
41     end for
42   end if
43   return nodes $t_i$
```

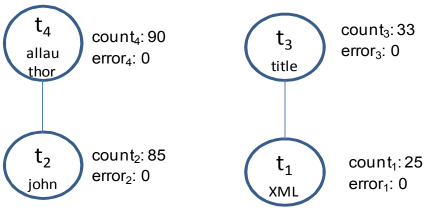

Figure 5. The nodes t_i after processing T_1, T_2,.., T_{99} in Figure 3 by X-Path.

After reading T_{104}, Lines 7-11 are executed and the values of variables $error_1$ and $error_3$ of t_1 and t_3 respectively are added by value 1 since the leaf node *XML* of T_{104} is the same as t_1. Finally, T_{105} is read and Lines 7-8 and 21-25 are executed. The values of variables $error_1$, $error_3$ and $error_4$ of t_1, t_3, and t_4 respectively are set by the value 1 and result in Figure 6.

Also, suppose that all of query trees T_1, T_2, …, and T_{200} in Figure 3 are sequential read and processed by function X-Subtree. Suppose that the stored nodes t_i is shown in Figure 6 before executing X-Subtree. Firstly, T_{100} is read and Lines 3-8 in function X-Subtree are executed since the relationship between the leaf nodes *XML* and *author* are not recorded in their corresponding nodes t_1 and t_5. Thus, $sibling_{15}$ is created and the variable $s\text{-}count_{15}$ is set to value 1. Then, T_{101} is read and processed by Lines 3-8 and the variable $s\text{-}count_{15}$ between t_1 and t_5 is added by value 1 since it is the same as T_{100}. In consequence, T_{102} is read and Lines 7-8 are executed since T_2's leaf nodes *title* and *allauthor* are the ancestors of nodes t_1 and t_5 respectively. Thus, $sibling_{34}$ between nodes t_3 and t_4 is created. Also, the value of variable $s\text{-}count_{34}$ is set by the sum of value 1 and the value of d_i's $s\text{-}count_{ij}$. This is shown in Figure 7. In addition, T_{103} and T_{104} is read and not to be processed since it has no a pair of leaf nodes. Finally, T_{105} is read and then Lines 7-8 are executed and results in Figure 8.

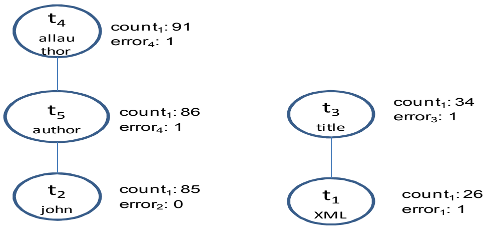

Figure 6. Nodes t_i for Figure 3 after executing X-Path.

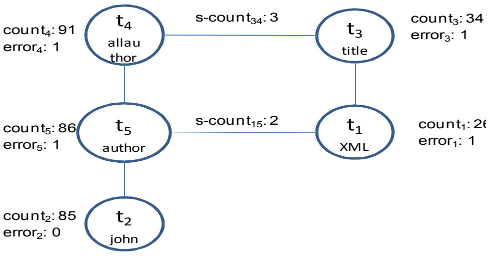

Figure 7. Nodes t_i for T_{100}, T_{101}, and T_{102} in Figure 3 after executing X-Subtree.

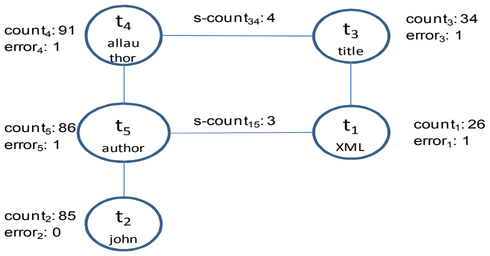

Figure 8. Nodes t_i for Figure 3 after executing X-Subtree.

```
Function X-Subtree(T)
Input: An XML query tree T
Output: nodes $t_i$
1  for each pair of leaf nods $l_i$ and $l_j$ of T
2    if the sibling relationship between $l_i$ and $l_j$ is not
3    stored
4      set the $sibling_{ij}$ relationships between $t_i$ and $t_j$
5      if there is no variable $s\text{-}count_{ij}$ of the descendant
6      nodes $d_i$ of $t_i$ and $t_j$
7        set the variable $s\text{-}count_{ij}$ between $t_i$ and $t_j$ with 1
8      else
9        set the variable $s\text{-}count_{ij}$ between $t_i$ and $t_j$ with
10       the value which is the sum of the value of
11       variable $s\text{-}count_{ij}$ of $d_i$ with $d_j$ and value 1
12   else
13     add 1 to the $s\text{-}count_{ij}$ variables between $t_i$ and $t_j$
14   end if
15 end if
16 end for
17 return nodes $t_i$
```

3.3. XSM

The following symbols t_i, p_i, (c_x, c_y), z_i, *temp_x*, *ct*, *fp*, and *fs* are used in algorithm XSM. t_i indicates the node which is stored in the system, p_i represents t_i's parent node, (c_x, c_y) represents the *xscode* in t_i, z_i indicates the sibling node of t_i, and *temp_space* represents a temp space in the system. Symbol *ctz* indicates a subtree of nodes t_i and z_i. In addition, symbol *fp* indicates a set of frequent paths, while *fs* shows a set of frequent subtrees.

Example 3: Suppose that *XQS* has the query trees T_1, T_2, …, and T_{200} as shown in Figure 3 and a user issues an request when the query tree T_1, T_2, .., T_{105} have processed and sets $\sigma = 0.3$. Also suppose that the error parameter ε is equal to 0.1 and the batch size is equal to 100. Therefore, a bucket has 10 ($1/\varepsilon=1/0.1$) query trees, β is equal to 10 ($\left\lceil\frac{100}{10}\right\rceil=10$), and $B_{current}$ is equal to 10 ($\left\lceil\frac{99}{10}\right\rceil=10$). Firstly, after executing Lines 3-5, the information of the XML query stream is shown in Figure 8. Then, Figure 9 and 10 show the results after executing algorithm XSM. Figure 9 shows the results after executing Lines 5-12. In Figure 9, the values of $count_i$ of t_i are bigger than 31.5 ($\sigma * i = 0.3 * 105 = 31.5$). Finally, figure 10 presents sets *fp* and *fs* after executing Lines 14-25.

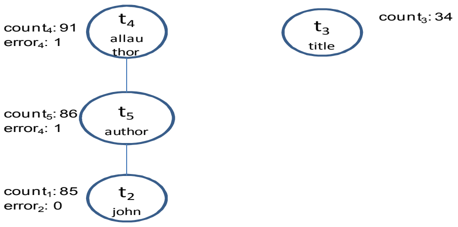

Figure 9. The frequent query patterns for Figure 3 after executing XSM.

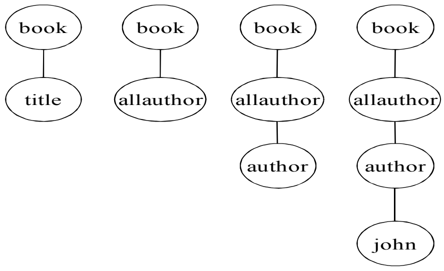

Figure 10. The frequent query patterns for Figure 3 after executing XSM.

```
Algorithm 1: XSM (XQS, ε, size, σ)
 Input: The XML query stream XQS; The value of error
 parameter ε; The size of batch size; Specified minimum
 support σ;
Output: A set of frequent query patterns from XQS
Repeat
read a query tree $T_i$ from XQS
X-Path($T_i$, ε, size);
X-Subtree ($T_i$, ε, size);
  if ($t_i$'s $count_i$ + $t_i$'s $error_i$) is small than $B_{current}$
    delete the node $t_i$
    delete all of $t_i$'s ancestor nodes
  end if
if a request issued from users
    copy all information into temp_space
    for each node $t_i$ in temp_space
    if the value of $count_i$ is small than $σ*i$
        delete the node $t_i$
        delete $t_i$'s sibling nodes
      delete all of $t_i$'s descendant nodes
    end if
   end for
/* generate frequent subtrees from temp_space; */
for each node $t_i$ with xscode ($c_x$, $c_y$) in temp_space
    while $c_x > 0$
      add a path ($p_i$, $t_i$) into set fp
      if $t_i$ has the sibling node $z_i$
          add the cross subtree ctz into set fs
      end if
      set $p_i$ is the parent of $p_i$
    end while
    delete $t_i$
    delete $t_i$'s sibling nodes
   end for
  end if
Until not eof(XQS)
```

3.4. Comparison

One reason confirms that XSM may outperform XQSMinerI and XQSMinerII. XQSMinerI and XQSMinerII construct a Dynamic Transaction summary Structure (DTS) that summarizes the query patterns seen so far and keeps track of the transaction ID of each query pattern. Then, through tree-join process (i.e., constructing data structure *ECTree*), the single branch candidate subtrees are merged to produce the frequent query patterns. As a result, in XQSMinerI and XQSMinerII, more XML query trees are processed on DTS and thus cost a lot of time to produce frequent XML query patterns. In contrast, XSM encodes the path and subtree information in an XML query tree's leaf node and results in a few nodes are recorded to be tested. Therefore, the mining performance is enhanced.

4. Experimental Results and Analyses

In this section we are to appraise our algorithm XSM. Two experiments in total have been conducted to evaluate the performance of XSM. The two experiments were carried out on the platform of personal computer with P8 2.67 GHz dual core CPU and about 4 GB of available physical memory space. The operating system is Windows 7, and the programs of the algorithm are implemented in C++ (and complied by Dev-C++).

To simulate an XML query stream, we employ the XMARK.DTD [7] as the DTDs to generate the query trees. First of all, we translate the DTDs into DTD trees. Secondly, we generate a query tree database containing 90000 different queries. Finally, we randomly select a number of queries (ranging from 10000 to 90000) from the previous step to form an XML query stream. In consequence, in the two experiments, parameters and their settings are listed in Table 1. The parameter n denotes the number of XML query trees in an XML query stream. Parameter ε illustrates the maximum error in the system to mine the frequent user query patterns. Parameter σ represents the value of minimum support in the system, and the parameter B denotes the batch size in the system.

Table 1. Simulation parameters and settings.

Parameters	Descriptions	Settings
n	Number of XML query trees	20,000 ~ 90,000
ε	Maximum error in the system	0.008 ~ 0.2
σ	Minimum supports	1%~20%
B	Size of batch in the system	2,000 ~ 8,000

The first experiment, with results shown in Figure 11, observes the execution time (Y-axis) of XSM for different support levels under different batch size (X-axis). The specified maximum error ε is set to 0.1 in Figure 11. From Figure 11, we can infer that for XSM, the execution time taken decreases drastically as the batch size increases. The reduction in time taken is even more pronounced when the batch size is 3000. The reason is that for small batch sizes, the *xsnodes* in the system are deleted frequently in the system. In addition, using the higher support as the threshold to prune away infrequent query patterns results in higher performance and further reduces the execution time taken. As a result, it is important to choose an appropriate batch size for an XML query stream mining algorithm. The performance will deteriorate significantly if the batch size is too small.

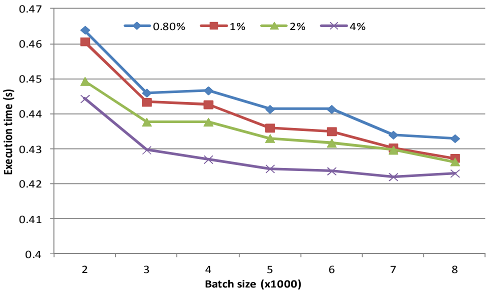

Figure 11. The execution time with varying batch size.

The second experiment, with results shown in Figure 12, observes the execution time (Y-axis) of XSM for different support levels under different maximum error (X-axis). The specified number of queries in the batch B is set to 4,000 in Figure 12. From Figure 12, we can infer that for XSM, the execution time taken decreases drastically as the maximum error increases. The reduction in time taken is even more pronounced when the maximum error is 0.05. The reason is that, the number of query trees is deleted for high maximum error more than that for low maximum error and results in less query trees' information are preserved in the system.

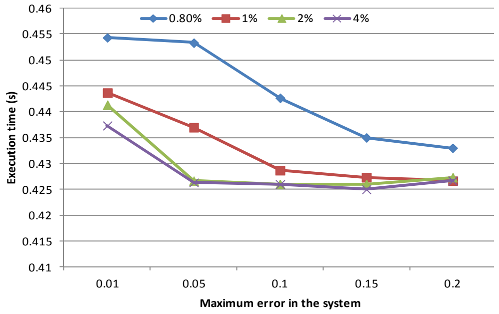

Figure 12. The execution time with varying maximum error.

5. Conclusions

In this paper, an efficient mining algorithm XSM is proposed to discover frequent XML query patterns from XML query streams. Unlike the existing algorithms, a new idea by encoding XML query trees (i. e., XSCode) is proposed and thus preserves the path and subtree information of query trees. With this idea, it became obvious that XSM is not capable of maintaining all of the user queries and thus takes less execution time and memory space to produce the frequent XML query patterns. Our future work includes expanding XML query patterns with repeating-siblings, since XSM cannot mine the frequent XML query patterns with sibling repetitions from an XML query stream.

Reference

[1] Asai T., Abe K., Kawasoe S., Arimura H., and Sakamoto H., "Online Algorithms for Mining Semi-structured Data Stream," *in Proceedings of the International Conference on Data Mining*, Maebashi, Japan , pp. 27-34, 2002.

[2] Auasu A., Babcock B., Babu S., McAlister J., and Widom J., "Characterizing Memory Requirements for Queries over Continuous Data Streams," *in Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, New York, USA, pp. 221 - 232, 2002.

[3] Babcock B., Datar M., and Motwani R., "Sampling from a Moving Window over Streaming Data," *in Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia, USA, pp. 633 - 634, 2002.

[4] Basci D. and Misra S., "Entropy as a Measure of Quality of XML Schema Document," *the International Arab Journal of Information Technology*, vol. 8, no. 1, pp. 75 - 83, 2011.

[5] Gu M., Hwang J., and Ryu K., "Frequent XML Query Pattern Mining based on FP-TRee," *in Proceedings of the 18th International Conference on Database and Expert Systems Applications*, Regensburg, Germany, pp. 555-559, 2007.

[6] Manku G. and Motwani R., "Approximate Frequency Counts over Data Streams," *in Proceedings of the 28th International Conference on Very Large Data Base*, Hong Kong, China, pp. 346 - 357, 2002.

[7] XMARK.DTD., available at: http://monetdb.cwi.nl/xml, 2014.

[8] XML., available at: http://www.w3.org/XML, 2013.

[9] XPath., available at: http://www.w3.org/TR/2007/REC-xpath20-20070123, 2014

[10] Yang L., Lee M., and Hsu W., "Efficient Mining of XML Query Patterns for Caching," *in Proceedings of the 29th International Conference on Very Large Data Bases*, Berlin, Germany, pp. 69 - 80, 2003.

[11] Yang L., Lee M., and Hsu W., "Finding Hot Query Patterns over an XQuery Stream," *the VLDB Journal*, vol. 13, no. 4, pp. 318 - 332, 2004.

[12] Yang L., Lee M., Hsu W., Huang D., and Wong L., "Efficient Mining of Frequent XML Query Patterns with Repeating-Siblings," *Information and Software Technology*, vol. 50, no. 5, pp. 375 - 389, 2008.

[13] Zhu Y. and Shasha D., "StatStream: Statistical Monitoring of Thousands of Data Streams," *in Proceeding of the 28th Conference on Very Large Data Bases*, Hong Kong, China, pp. 346 - 357, 2002.

Tsui-Ping Chang received her PhD degree in Computer Science and Engineering from National Chung Hsing University in 2009. Since 2000, she was the faculty member of the Department of Business Administration, Kao Yuan University. In 2009, she joined the Department of Information Technology, Ling Tung University. Her research interests include XML database systems, XML data mining, and object-oriented systems.