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.

Several methods [1, 4, 11] have been proposed for mining frequent user query patterns over an XML query stream. XQSMiner is a one pass algorithm to find out frequent user query patterns from XML query
Mining Frequent User Query Patterns from XML Query Streams

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.

Definition 1 defines an XML query as a tree. Definition 2 defines an XML query rooted subtree. Definition 3 illustrates an XML query stream 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 = <T1, T2, ..., T100, T101, T102, T103, T104, T105, ..., T200>, where T1, T2, ..., T100, T101, T102, T103, T104, T105, ..., T200 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

XSCodes encodes the nodes of an XML tree in an xy coordinate system where x and y 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 represents 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 (xp, yp), then k’s coordinates are (xp + 1, yp + 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 (xi, yi). If m = 0 then k’s coordinates are (xi + 1, yi), otherwise, k’s coordinates are (xi + m, yi).

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

**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 xsodes 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 xsode is (0, 0). According to Rule 2, the nodes title, XML, author, john, jane, 2000, head1, origins, and head2 are the fc nodes of a node in the tree and their xsodes 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 section; are the nc nodes of a node in the tree and their xsodes are (2, 1), (4, 1), (5, 1), (4, 2), (7, 2), and (9, 3) respectively.

![Figure 3. The XML query stream XQS.](image)

![Figure 4. The xsodes of the nodes in Figure 1.](image)

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

**Lemma 1:** For any node f in an XML tree T, if f’s xsode is (x, y), then the value of y is equal to the level l of the node f in T.

**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.

- **Case 1:** Suppose that node f is the root node in T. According to Rule 1, the xsode 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. Since f is not the root node and with the level l, it has the ancestor nodes p0, p1, ..., p_l, where p_0 is f’s parent node, p_1 is p_0’s parent node, ..., and p_l is the root node. According to Rule 1, the xsode of p_0 is (0, 0). Thus, y_p0 is equal to 0. Also, according to Lemma 2, p_l’s xsode y_pl = y_p0 + 1. Thus, y_pl = y_p0 + 1 = 0 + 1 = 1. In consequence, p_l’s xsode y_p2 = y_pl + 1 = 1 + 1 = 2. Therefore, p_l’s xsode y_pl = l - 1. Since f is the child node of y_pl, f’s xsode y = y_pl + 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. According to Case 2, the fc’s xsode y_fc = l. In consequence, according to Rule 3, f’s xsode 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 f1 and f2 in an XML tree T, with the xsodes (x_1, y_1) and (x_2, y_2) respectively, if node f2 is a descendant node of f1, 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 f2 is a child node or not of f1.

- **Case 1:** Suppose that node f2 is a child node of f1. If f2 is the first child node of f1, according to Rule 2, the xsode (x_2, y_2) of f2 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 xsode of f1’s first child node f1 and f1 has m descendant nodes. Thus, if f2 is the first child node of f1, 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_2 = x_1 + 1, and y_2 = y_1 + 1 which result in x_2 > x_1 and y_2 > y_1. As a result, x_2 > x_1 and y_2 > y_1.

- **Case 2:** Suppose that node f2 is not a child node of f1 and has a parent node f_0 which is a child node of f1. According to Case 1, node f_0’s xsode x_0 > x_f1 and y_0 > y_f1. Also, since f_1’s xsode x_f1 > x_f0 and y_f1 > y_f0, they result x_f1 > x_f0 and y_f1 > y_f0.

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, w, ε, β, B_curren, t_i, a_i, count, error, sibling, s-count, 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 indicates the frequencies of t_i, error_i shows t_i’s frequencies which are not recorded in the system, sibling_i indicates the sibling relationship between nodes t_i and t_j, and s_count_j 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 width w = 1/ε.
query trees. The buckets are labeled with bucket IDs starting from 1. The current bucket ID is denoted as \( B_{\text{current}} \), whose value is \( \frac{n}{w} \). Also, the number of buckets in the main memory for the XML query stream is denoted as \( \beta \). 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 \( \text{count}_i > \beta \) and \( B_{\text{current}} > \beta \) in the system, the value of \( (B_{\text{current}} - \beta) \) are set into their variables \( \text{error}_i \). 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 \( T_1, T_2, \ldots, T_{99} \) have been processed by X-Path and result in the stored nodes \( t_i \) as shown in Figure 5. Suppose that the error parameter \( \varepsilon \) 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, \( \beta \) is equal to \( 100 \) \((\frac{100}{10} = 10)\), and \( B_{\text{current}} \) is equal to \( 10 \) \((\frac{90}{10} = 10)\). 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 \( t_i \). Therefore, the value 1 is added into the variables \( \text{count}_1 \) and \( \text{count}_5 \) of \( t_1 \) and \( t_5 \) respectively. Also, Lines 27-28 are executed and the new node \( t_5 \) is inserted between \( t_1 \) and \( t_5 \) since the leaf node \( \text{author} \) is a parent of \( t_5 \). Then, \( T_{101} \) is read and the value of \( B_{\text{current}} \) is changed to 11 (i.e., \( \frac{101}{10} = 11) \). Therefore, Lines 8-9 are executed since the values of variables \( \text{count}_1 \) and \( \text{count}_5 \) of \( t_1 \) and \( t_5 \) (i.e., values 26 and 86 respectively) are bigger than that of \( \beta \) (i.e., the value 10) and the value of \( B_{\text{current}} \) (i.e., the value 11) is bigger than that of \( \beta \). As a result, the variables \( \text{error}_1 \), \( \text{error}_3 \), \( \text{error}_4 \), and \( \text{error}_5 \) of \( t_1 \), \( t_3 \), \( t_4 \), and \( t_5 \) respectively are set to value 1 (i.e., \( B_{\text{current}} - \beta = 11 -10 \)). In consequence, \( T_{102} \) is read and Lines 20-24 are executed since the leaf nodes \( \text{title} \) and \( \text{allauthor} \) of \( T_{102} \) are the same as the nodes \( t_1 \) and \( t_4 \) respectively. Therefore, the value 1 is set into the variables \( \text{error}_1 \) and \( \text{error}_4 \) of \( t_1 \) and \( t_4 \) respectively. Then, \( T_{103} \) is read and Lines 21-25 are executed since \( T_{103} \)’s leaf node \( \text{title} \) is the ancestor of node \( t_1 \). Thus, the value of variable \( \text{error}_1 \) of \( t_3 \) is set by the value 1.

**Function X-Path** \((T, \varepsilon, \text{size})\)

**Input**: An XML query tree \( T \)

**Output**: nodes \( t_i \)

1. if there is no node stored in the system
2. store the nodes \( t_i \) as \( t_i \) and set variables \( \text{count}_1 \) with 1
3. for each leaf node \( t_i \) of \( T \)
4. compare the \( t_i \)’s xscodes with that of each \( t_i \)
5. if \( t_i \)’s xscodes is the same with that of \( t_i \) then
6. \( t_i \)’s \( \text{error} = B_{\text{current}} - \beta \)
7. all of \( t_i \)’s ancestor nodes \( a_i \), \( \text{error} = B_{\text{current}} - \beta \)
8. else
9. add 1 to \( \text{count}_1 \) of \( t_i \) and all of ancestor nodes \( a_i \)
10. else
11. if \( t_i \) is \( a_i \)’s ancestor and their xscodes do not
12. stratify Lemma 2, and \( t_i \) has no ancestor \( a_i \)
13. store the node \( l_i \) as a parent node \( p_i \) of \( t_i \)
14. set the value of \( \text{count}_1 \), of \( p_i \), is the sum of that of \( t_i \) with value 1
15. if \( (p_i \)’s \( \text{count}_1 ) > \beta \) and \( B_{\text{current}} > \beta \) then
16. \( p_i \)’s \( \text{error} = B_{\text{current}} - \beta \)
17. all of \( p_i \)’s ancestor nodes \( a_i \), \( \text{error} = B_{\text{current}} - \beta \)
18. else
19. find a \( a_i \) which is a child node \( l_i \)
20. set node \( l_i \) as a parent node \( p_i \) of \( a_i \)’s parent
21. if \( l_i \) is a descendant node of \( t_i \)
22. store \( l_i \), into a new created node \( c_i \)
23. set node \( c_i \) as a child node of \( t_i \)
24. add 1 to \( \text{count}_1 \), and all of \( c_i \)’s ancestors
25. relationship
26. store \( l_i \) into a new created node in system
27. end if
28. end for
29. end if
30. return nodes \( t_i \)

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

After reading \( T_{104} \), Lines 7-11 are executed and the values of variables \( \text{error}_1 \) and \( \text{error}_5 \) 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 \( \text{error}_1 \), \( \text{error}_3 \) and \( \text{error}_5 \) of \( t_1 \), \( t_2 \), and \( t_4 \) respectively are set by the value 1 and result in Figure 6.

Also, suppose that all of query trees \( T_1, T_2, \ldots, T_{200} \) 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 $t_I$ and $t_5$. Thus, sibling(i) is created and the variable $s$-count[15] is set to value 1. Then, $T_{100}$ is read and processed by Lines 3-8 and the variable $s$-count[15] between $t_I$ and $t_5$ is added by value 1 since it is the same as $T_{100}$. In consequence, $T_{100}$ is read and Lines 7-8 are executed since $T_5$’s leaf nodes title and allauthor are the ancestors of nodes $t_I$ and $t_5$ respectively. Thus, sibling[34] between nodes $t_3$ and $t_4$ is created. Also, the value of variable $s$-count[04] is set by the sum of value 1 and the value of $d_i$’s $s$-count[0]. 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.

```
9 set the variable $s$-count[1] between $t_i$ and $t_j$ with
10 the value which is the sum of the value of
11 variable $s$-count[1] of $d$ with $d_j$ and value 1
12 else
13 add 1 to the $s$-count[1] 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 $t_i$, $p_i$, $(c_i, c_v)$, $z$, 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, $(c_i, c_v)$ represents the xscode in $t_i$, $z$ 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 $z$. 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 $T_{11}$, $T_{22}$, ..., $T_{105}$ have processed and sets $\sigma = 0.3$. Also suppose that the error parameter $\epsilon$ is equal to 0.1 and the batch size is equal to 100. Therefore, a bucket has 10 \((1/\epsilon = 1/0.1)\) query trees, $\beta$ is equal to 10 \((105/10 = 10)\), and $B_{current}$ is equal to 10 \((105/10 = 10)\). 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[1] 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.

```
Function X-Subtree(T)
Input: An XML query tree $T$
Output: nodes $t_i$
1 for each pair of leaf nodes $t_i$ and $t_j$ of $T$
2 if the sibling relationship between $t_i$ and $t_j$ is not
3 stored
4 set the sibling(i) relationships between $t_i$ and $t_j$
5 if there is no variable $s$-count[1] of the descendant
6 nodes $d_i$ of $t_i$ and $t_j$
7 set the variable $s$-count[1] between $t_i$ and $t_j$ with 1
8 else
9 set the variable $s$-count[1] between $t_i$ and $t_j$ with
10 the value which is the sum of the value of
11 variable $s$-count[1] of $d$ with $d_j$ and value 1
12 else
13 add 1 to the $s$-count[1] variables between $t_i$ and $t_j$
14 end if
15 end if
16 end for
17 return nodes $t_i$
```
Mining Frequent User Query Patterns from XML Query Streams

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
1 Repeat
2 read a query tree T_i from XQS
3 X-Path(T_i, ε, size);
4 X-Subtree(T_i, ε, size);
5 if (t_i’s count + t_i’s error) is small than B_{current}
6 delete the node t_i
7 delete all of t_i’s ancestor nodes
8 end if
9 if a request issued from users
10 copy all information into temp_space
11 for each node t_i in temp_space
12 if the value of count, is small than σ * i
13 delete the node t_i
14 delete t_i’s sibling nodes
15 delete all of t_i’s descendant nodes
16 end if
17 end for
18 /* generate frequent subtrees from temp_space; */
19 for each node t_i with xsnode (c_i, c_j) in temp_space
20 while c_i > 0
21 add a path (p_j, t_j) into set fp
22 if t_j has the sibling node z_j
23 add the cross subtree c_{tz} into set fs
24 end if
25 set p_j is the parent of p_i
26 end while
27 delete t_i
28 delete t_i’s sibling nodes
29 end for
30 end if
31 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.

<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>20,000 ~ 90,000</td>
</tr>
<tr>
<td>ε</td>
<td>Maximum error in the system</td>
<td>0.008 ~ 0.2</td>
</tr>
<tr>
<td>σ</td>
<td>Minimum supports</td>
<td>1% ~ 20%</td>
</tr>
<tr>
<td>B</td>
<td>Size of batch in the system</td>
<td>2,000 ~ 8,000</td>
</tr>
</tbody>
</table>

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.
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.](image)

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


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.