AN EFFICIENT XML DATABASE MINING WITHOUT CANDIDATE GENERATION: AN FREQUENT PATTERN SPLIT APPROACH

Chin-Feng Lee and Tsung-Hsien Shen
Department of Information Management
Chaoyang University of Technology
No. 168, Jifong E. Rd., Wufong Township,
Taichung County 41349, Taiwan (R.O.C.)
Email: lcf@cyut.edu.tw

ABSTRACT
The popularity of XML results in producing large numbers of XML documents. Therefore, to develop an approach of association rule mining on native XML databases is an important research. The FP-growth based on an FP-tree algorithm performs more efficiently than other methods of association rules mining, but it cannot be applied to native XML databases. Hence, we adaptive an improving FP-tree algorithm called Frequent Pattern Split method, simply FP-split, for fast association rule mining from native XML databases. We show that the FP-split method is time-efficient for mining association rules from native XML databases by experiments with various parameters, such as various minimum supports, different number of items, and large amount of data. In addition, we also implement a lot of experiments to show that our proposed method performs better than FP-tree construction algorithm in transaction database.

Keywords: data mining, association rule, XML, DTD, XML schema, XML database.

1 INTRODUCTION

Due to the extensive application of XML (eXtensible Markup Language) technology by various corporations in different fields, an enormous number of XML documents have been created [3][7][13]. It becomes imperative to enhance data mining among the ever-growing native XML databases and uncover the hidden, unpredictable and unknown information. Therefore, scholars began to propose mining techniques for XML documents in recent years, but most of them were based on XQuery for the mining of text [2][15][16][17]. In studies on the mining of association rules by XQuery, efficiency of the mining technique was not particularly emphasized. Some researchers even adopted earlier Apriori Algorithm, along with XQuery, for mining of XML documents [15][16]. In studies of [15][16], tag was not treated as a mining object; thus it's been overlooked that tag might carry more important information or rules. Besides, when there's no association rule to be generated from the character data, the tag describing the character data itself might be sufficient to serve as an association rule. In this study, our research goals are:

(1) Develop a mining technique for native XML databases

(2) Design an efficient mining technique:
A revised FP-tree Algorithm, FP-split Algorithm, is proposed to mine association rules in XML documents. When applied to conventional transaction databases, FP-split Algorithm performs better efficiency than FP-tree.

(3) View tag as the object of data mining
In native XML databases, not only character data but tags are targets of data mining. Any association rule will be extracted, whether it is between tag and tag, tag and character data, or character data and character data.

(4) Extract association rules with complete information
In the process of extracting, association rules in XML documents are fully described, whether they are generated from character data or tags.

By analyzing DTD or XML Schema in XML documents, this research developed a quick mining technique, FP-split Algorithm, for the mining of association rules in native XML databases. The mining was aimed to disclose all the possible, concealed and complete information behind character data and tags. Scanning the database for only once, FP-split Algorithm can find all the frequent itemsets without generating any candidate itemsets. Verified by various experimental parameters, FP-split Algorithm is shown to be highly efficient.

2 LITERATURE REVIEWS

2.1 DTD and XML Schema

DTD and XML Schema can be used to define the structure of XML documents, as well as tag names, tag attributes, number of tag occurrence and the content model of tags. Table 1 lists the numbers of tag occurrences allowed in an XML document as defined by DTD. Four symbols are used, "*", "+", "?" and blank. Tag names and tag attributes are defined by "ELEMENT" and "ATTLIST".

Table 1: Frequency of element occurrence

Symbol	# of occurrence
?	0 or 1
*	0 or more
+	1 or more
blank	1

XML schema was proposed by Microsoft, and its definition method is more complicated. It can define tag names, tag attributes, numbers of tag occurrences and content models of tags. Moreover, it can also define the type of character data.

2.2 Native XML Database

Currently, there are two ways of storing and managing XML documents. One is to process and transform the XML documents to save them in a relational database and restore them to the original XML document [3][12] The other is to save the XML documents in a native XML database. As the first approach is more complicated and unnatural, a number of companies began to develop native XML databases, e.g. X-Hive/DB by X-Hive [15][16][17], Tamino/DB by Software AG, Ipedo XML Database by Ipedo and Apache Xindice by Apache.

Most native XML databases contain the architecture of Collection that is capable of storing multiple XML documents with the same XML Schema or DTD. Therefore, the mining of association rules will be conducted on the character data and tags of multiple XML documents within the same architecture.

2.3 Association Rule

The definition of association rules was as follows [1]: Let $I= \{i_1, i_2, \ldots, i_n\}$ be a set of items in transaction databases. Let $D= \{T_1, T_2, \ldots, T_m\}$ be a set of transactions. Each transaction T contains a set of items in I. An association rule means an implication of the *XÆY* where $X \subset I$, $Y \subset I$, and $X \cap Y= \emptyset$. The rule *XÆY* must satisfy two criteria that are both the support and the confidence. A *support s*% is the percentage of the associated transactions $X \cup Y$ in the transaction database. The confidence *c*% means that the percentages of transactions in the transaction database containing X also containing Y. If both the support and the confidence of the rule *XÆY* are greater than user specified minimum support and minimum confidence, the rule *XÆY* is strong.

2.4 FP-tree and FP-growth Algorithm

The FP-tree and FP-growth algorithms were proposed for improving the efficiency of association rule mining [6]. The FP-growth algorithm based on an FP-tree to generate frequent itemsets without candidate itemsets generation and the FP-tree construction algorithm scans the database only twice. Hence, the FP-tree and FP-growth Algorithm save a lot of I/O time to enhance mining efficiency.

FP-tree construction algorithm:

Step 1. Scan database to generate frequent items.

Step 2. Store the set of frequent items into a list labeled as "*L*" and is sorted by their supports.

Step 3. Construct an FP-tree in the following two sub steps: The First is to create a root labeled with "null". The second one is to scan database a second time. The items in each transaction are processed in *L* order and a branch of tree is created for each transaction. If a new branch shares a common prefix with the existing pattern for some transactions, the count of each node along the common prefix is incremented by one and node for the items following the prefix are created and linked accordingly.

FP-growth algorithm:

Step 1. Find conditional pattern base of a frequent itemset with length $k \geqq 1$ from FP-tree.

Step 2. Construct a conditional FP-tree on the conditional pattern base.

Step 3. Exploit the conditional FP-tree for generating frequent itemsets with length k+1.

Before constructing the FP-tree, a root node and a header table will be created. The header table consists of two columns; one is "Item" to list sorted frequent items and the other is "Link", recording each item's starting point in the FP-tree. While constructing the FP-tree, each item in the transaction log has to be put in order. If an item already exists on a node of a collative path, a count of the node is cumulated. Otherwise, a new node will be added on the path. Afterwards, FP-growth Algorithm based on the FP-tree can generate frequent itemsets. A bottom-up approach is performed by the FP-growth Algorithm

2.5 Data mining on the development and application of XML

As XML is widely used in various areas, a number of XML documents were created. Several scholars applied the technology of data mining to

XML documents to find meaningful information. Currently, issues concerning mining of XML documents include finding the DTD structure of XML documents [11], analyzing similarities between XML documents [10], mining of frequent query pattern of XML [19], mining of association rules of character data in XML documents through XQuery [2][15][17], and mining of association between character data and tags in XMLs documents [9].

XQuery was used in most of the studies as the mining technique. However, it was neglected that tags might also contain important information. Therefore, Lee et al. [9] proposed a technique capable of mining association rules in character data, as well as tags. Nevertheless, the technique Lee proposed could only mine shortly association rules instead of complex association rules. Assume that an association rule exists in this XML document "customers who buy milk will buy bread as well". The shortly association rule will express this rule as "milkÆbread", while the complex associate rule will state in the manner of "<chocolate>milk</chocolate>Æ<strawberry>bread </strawberry>". By comparison, we can see that the complex association rule can explicitly present the more complete information of the association rule. Therefore, it is paramount for this paper to develop association rules that handle both character data and tags and consist of complete information.

3 METHODOLOGY

We propose a revised FP-tree Algorithm, called FP-split Algorithm, for mining in native XML databases. Since FP-split Algorithm is originated from FP-tree Algorithm, it can be used in transaction databases as well. Besides, its mining efficiency is better than FP-tree Algorithm. The methodology is divided into 5 major steps, and the research procedure is shown in Figure 1.

First of all, DTD or XML Schema in one collection architecture of the native XML database is parsed. Based on the occurrence count of each element and the element's content model, element names are listed in three sets, i.e., *TS*, *TM* and *TO*. Next, by matching element names in the three sets with tag names in the XML document, equivalence classes of both tags and character data can be obtained. Meanwhile, support of each item is also computed. After equivalence classes of item are created, items under support threshold are filtered. Then, the equivalence classes of all frequent items are converted into nodes, which in turn are matched and partitioned with the concept of intersection and difference to build an FP-split tree. Finally, all the frequent itemsets of the FP-split tree are mined to create association rules. Let *XDB* be a native XML database, and let $XDB=\{C_1, C_2, \ldots, C_N\}$, indicating that *XDB* consists of *N* architectures of collection. Each collection *Ci* contains a set of XML documents constrained by DTD or XML Schema.

Let $X=\{X_1, X_2, \ldots, X_m\}$, in which X represents the set of multiple XML documents. Also, let $X_i = TG \cup CD$, in which $TG=\{t_1, t_2, \ldots, t_e\}$. *TG* is the collective name of all the elements in the structure of DTD or XML Schema, i.e. the tag name in XML documents. *CD* represents the character data in XML documents. The method and steps of mining association rules are described as follows.

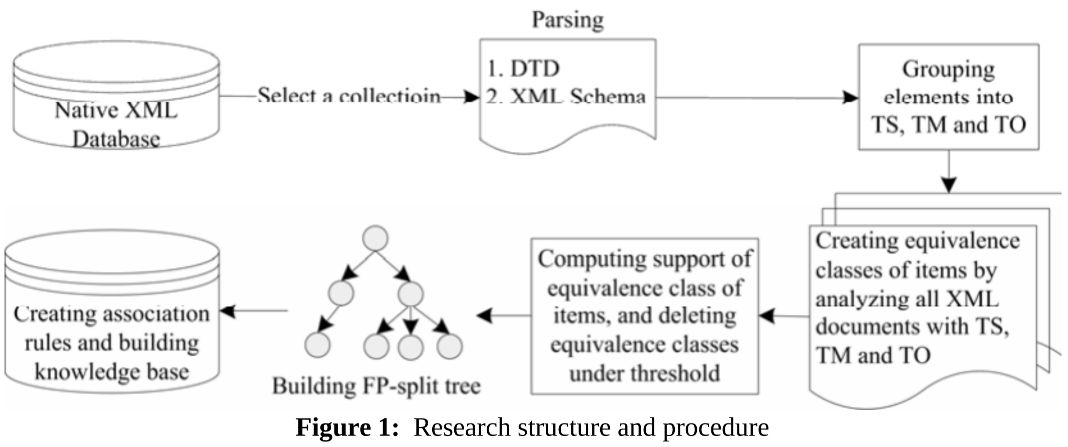

Parsing
Native XML Database
Select a collectioin
1. DTD
2. XML Schema
Grouping elements into TS, TM and TO
Creating equivalence classes of items by analyzing all XML documents with TS, TM and TO
Computing support of equivalence class of items, and deleting equivalence classes under threshold
Building FP-split tree
Creating association rules and building knowledge base

Figure 1: Research structure and procedure

Step 1. Analyzing DTD or XML Schema

Given each collection C_i, definitions of the XML document structure provided by DTD or XML Schema are parsed. Therefore, information can be obtained, including tag names which may occur in the XML document, tag's occurrence count, and the content model of the tag. Definition 1 below explains the minimum occurrence count of an element in the XML document and its content model.

[Definition 1] Minimum occurrence count of an element in the XML document and its content model

Let $f^0(t)=Z^+ \cup \{0\}$, in which $t \in TG$, Z^+ is a positive integer, and $f^0(t)$ indicates the minimum occurrence count of element *t*.

Let $f^C(t)=\{0, 1\}$, in which $t \in TG$, and $f^C(t)$ is the content model of element *t*. When $f^C(t)=0$, the content model of element *t* is sub-element. If $f^C(t)=1$, the content model of element *t* is character data.

When parsing the structure of DTD or XML Schema, elements are grouped into three sets, *TS*, *TM* and *TO*, according to the number of element occurrence and whether the content model is sub-element or character data. The element names listed in the three sets are the main objects for parsing XML documents. The following is definitions of *TS*, *TM* and *TO*.

[Definition 2] TS (Super Tag Set)

Let $TS=\{ts_1, ts_2, \ldots, ts_p\}$, in which the least occurrence count of element ts_i in DTD or XML Schema is labeled as zero. Besides, its content model is sub-element. Ts_i is the tag in XML documents, and may contain sub-tags.

[Definition 3] TM (Mandatory Tag Set)

Let $TM=\{tm_1, tm_2, \ldots, tm_q\}$, in which the least occurrence count of element tm_i in the structure of DTD or XML Schema is labeled as once. Its content model is character data, which means that tm_i is a tag to describe character data.

[Definition 4] TO (Optional Tag Set)

Let $TO=\{to_1, to_2, \ldots, to_r\}$, in which the least occurrence count of element to_i in the structure of DTD or XML Schema is labeled as zero. Its content model is character data, which means that to_i is a tag to describe character data in XML documents.

According to Definitions 2, 3 and 4, we have $TS \cap TM \cap TO=\emptyset$. Characteristics of the three sets are as shown in Table 2.

Table 2: Characteristics of *TM*, *TS*, and *TO*

	TS	***TM***	***TO***
The least occurrence count of element	0	1	0
Content model of element	0	1	1

From these three sets, there isn't any element name with least occurrence count of once, nor the content model of sub-element. Since this type of element name is bound to appear in every XML document without recording any character data, as a mining target, it will certainly become a frequent item associated with other elements or character data. Therefore, it is an unnecessary target for mining. Considering such a situation, evident association rules will be filtered in advance, and only unpredictable information or hidden knowledge will be mined to decrease the number of producing unimportant rules. Algorithm 1 functions as the technique for parsing of DTD or XML Schema and creating the sets, *TS*, *TM* and *TO*.

Algorithm 1：Generation of *TS*, *TM* and *TO*

```
/* Let TG be a set of elements in DTD or XML schema,
  $t \in TG$ */
Input: Given a DTD or XML Schema
Output: Three sets $TS$, $TM$ and $TO$
Begin: For each t in $TG$
  If ( $f^0(t) == 0$ and $f^C(t) == 0$ ) $TS$ Å $\{t\}$
  Else if ( $f^0(t) == 1$ and $f^C(t) == 1$ ) $TM$ Å $\{t\}$
  Else if ( $f^0(t) == 0$ and $f^C(t) == 1$ ) $TO$ Å $\{t\}$
End Algorithm 1
```

Step 2. Creating equivalence classes of items

By utilizing *TS*, *TM* and *TO* formerly obtained by Algorithm 1, every tag name in the XML document is parsed to build the equivalence class of tag or character data.

[Definition 5] Item

Let item ι be represented as $x_1(x_2(\ldots(x_s)))$, in which x_i and x_j are either tag name or character data in XML documents, and $i < j$. Therefore, x_i is the ascendant tag of x_j, that is to say, x_j is character data and sub-tag of x_i.

If x_j is a tag, an equivalence class of tag can be created. If x_j is character data, an equivalence class of character data can be created.

[Definition 6] Equivalence Class of Item

Let equivalence class of item be $EC_\iota=\{i|$XML document X_i where item ι occurs$\}$.

By using *TS*, *TM* and *TO* set, tag names in every XML document are parsed to create equivalence classes of tag or character data. The parsing methods fall into three cases. Let X_i be the i^{th} document.

Case 1: Creating an equivalence class of tag

If (t is also in TS) Then $EC_t=EC_t \cup \{i\}$.

When the tag name in an XML document is the same as the element name of the *TS* set, record the tag name and the XML document number. They are the equivalence class of tag.

Case 2: Creating an equivalence class of character data

If (x is also in TM) Then
Find the child of x (say y) and generate item ι such that $\iota \equiv x(y)$, $EC_\iota=EC_\iota \cup \{i\}$.

If the tag name in an XML document is the same as the element name of the *TM* set, record the character data and the number of the XML document. These are the equivalence class of character data.

Case 3: Creating an equivalence class of tag and character data

If (t is also in TO) Then
Find the child of x (say y) and generate item ι such that $\iota \equiv x(y)$,
$EC_\iota=EC_\iota \cup \{i\}$ and $EC_t=EC_t \cup \{i\}$.

If the tag name in an XML document is the same as the element name of the *TO* set, not only the tag name and XML document number should be recorded, but the character data and the XML document number. Therefore, the equivalence class of tag and character data will be created simultaneously. Table 3 lists *TS*, *TM*, and *TO* can create equivalence class set.

Table 3: Generation equivalence classes

	Equivalence class of tag	Equivalence class of character data
Tag belongs to *TS*	V	
Tag belongs to *TM*		V
Tag belongs to *TO*	V	V

Step 3. Computing support

Support of each item ι is the number of XML documents contained in the equivalence class of item. Let $|EC_\iota|$ be support of the equivalence class of item. After support of each item ι is calculated, those lower than the minimum support are deleted and those that cross the threshold are reserved as frequent items. Finally, items are sorted according to support in descending order.

Step 4. Constructing FP-split tree

To facility tree traversal, a header table is built in advanced so that each item can point to its first occurrence in the FP-split tree. There are two entries for each item in the header table. The first entry is to store frequent items and the second one is used to link the associated items in the FP-split tree.

There are five entries in a node structure of FP-split tree that are *Content*, *List*, *Count*, *Link_sibling* and *Link_child*. The *Conten*t entry is to store frequent item ι. The *List* entry is store EC_ι. The *Count* entry is to record the support of item ι, that is to say, $|EC_\iota|$. The *Link_sibling* entry is a pointer, as it is mainly used for the connection of the nodes with the same item in the entry of *Conten*t. The *Link_child* entry is also a pointer for the connection of child nodes.

There are four rules for constructing FP-split tree, where *p* stands for a specific node in the FP-split tree. Let *n* be a new node about to be added into the FP-split tree. Each time to add a new node *n* into the FP-split tree, all four rules should be taken into consideration.

Rule 1: *p* is root
If (*p* is *root* and *p.Link_child* == null)
Then *p.Link_child* Å *n*
Else Compare (*p.Link_child.List*, *n.List*)
Rule 2: *n.List* ⊆ *p.List*
If (*n.List* ⊆ *p.List* and *p.Link_child* == null) Then *p.Link_child* Å *n*
Else *Compare* (*p.Link_child.List*, *n.List*)
Rule 3: *n.List* ∩ *p.List*=Ø
If (*n.List* ∩ *p.List* == Ø and *p.Link_sibling* == null)
Then *q.Link_child* Å *n* // *q* denotes parent of *p*
Else *Compare* (*q.Link_child.List*, *n.List*)

Rule 4: *n.List* ∩ *p.List* ≠ Ø 且 *n.List* ≠ *p.List*
If (*p.List* ∩ *n.List* ≠ Ø and *n.List* - *p.List* ≠ Ø) Then
Generate a new node n2
    n_2*.Content* = *n.Content*
    n_2*.List* = *n.List* − *p.List*
    n.List = *n.List* ∩ *p.List*
    n.Link_sibling Å n_2*.Link_sibling*

When the *List* of node *n* resembles partially that of node *p*, that is to say *n.List* ∩ *p.List* ≠ Ø and *n.List* ≠ *p.List*, node *n* would be split into two nodes, that is to say node n_1 and node n_2. The item stored in the entry *Content* of node n_1 is the same to the item stored in the entry *Content* of node n_2. That is n_1*.Content*= n_2*.Content*=*n.Content*. The *List* of node n_1 resembles partially that of node *p* by the operation of intersection, as shown in Eq. (1). The *List* of node n_2 is different from that of node *n* and node *p*, the difference operation will be taken as shown in Eq. (2). After splitting, the *Link_sibling* entry of node n_2 will be connected to the node connected by *Link_sibling* of node *n* and then *Link_sibling* of node n_1 is immediately connected to node n_2, thus retaining the connection.

$$n_1.List = n.List \cap p.List, \quad (1)$$
$$n_2.List = n.List - p.List. \quad (2)$$

Next, node n_1 will first, by following the definition of Rule 2, decide whether to become a child node of node *p*, or whether to compare to child nodes of node *p*. Node n_2 will, by following the definition of Rule 3, decide whether to become a sibling node of node *p* or whether to compare to sibling nodes of node *p*.

Step 5. Mining association rules

After a complete FP-split tree is constructed, the mining of the FP-tree is performed to create all frequent itemsets.

Phase 1: Creating frequent itemsets

As TD-FP-growth Algorithm proposed by Wang et al. [17] only applies to conventional transaction databases, Algorithm 2 is developed as an Adaptive TD-FP-growth Algorithm to implement in native XML databases so that the frequent itemsets in FP-split tree can be mined. Let α be an itemset, and ζ be the path from root node to the node which contains item α (α is conditional pattern-base). Let $\beta \in \zeta$, and ζ be a sequential path, $\zeta=\{\beta_1, \beta_2, \ldots, \beta_n\}$, in which β_1 is child node under root node, and β_n is parent node of the node which contains item α.

[Definition 7] Super Items

Let R_α be the super item of item α, $\alpha=x_1(x_2(\ldots(x_n)))$, so $R_\alpha=\{x_1(x_2(\ldots(x_n(x_{n+1})))), x_1(x_2(\ldots(x_n(x_{n+1}(x_{n+2}))))),\ldots,x_1(x_2(\ldots(x_n(x_{n+j}(x_{n+j+1})))))\}$, in which $j\in \mathrm{N}\cup\{0\}$.

[Definition 8] Trivial Items

Let item $\alpha=x_1(x_2(\ldots(x_n)))$, and its segment component set be $S_\alpha=\{x_1, x_1(x_2), \ldots, x_1(x_2(\ldots(x_{n-1})))\}$. S_α

consists of n-1 segment components of item α. For item α, there are two cases of trivial items:

Case 1: item $\beta_i \in S_\alpha$. For item α, item β_i is a trivial item.

Case 2: item $\beta_i \in R_\alpha$. For item α, item β_i is a trivial item.

[Property]

For item α, if item β_i is a trivial item, the frequent itemset $\alpha \cup \beta_i$ may create a trivial association rule, i.e. αÆβ_i or β_iÆα. Therefore, the frequent itemset $\alpha \cup \beta_i$ should be avoided creating trivial association rules.

Algorithm 2 modifies the TP-FP-growth Algorithm proposed by Han *et al.* [17]; therefore, it applies to semi-structured XML documents.

```
Algorithm 2: Adaptive TD-FP-growth
Input: a FP-split tree
Output: frequent patterns
Begin   Mine-tree ($L$, $H$)
   { For each entry $\alpha$ in $H$
If ( $H(\alpha)$ >= minsup ) and ( $\alpha$ not exist ($S_L$ and $R_L$) )
     output Al
     create a new header table $H_\alpha$ by call function
         Build-subtable ($\alpha$)
     mine-tree ($\alpha L$, $H_\alpha$)                    }
 Build-subtable ($\alpha$)
  {For each node $u$ on the Link of $\alpha$
        walk up the path from $u$ once do
if encounter a $J$_node $v$
then    link $v$ into the Link of $J$ in $H_\alpha$
           count($v$)=count($v$)+count($u$)
           $H_\alpha(J)$= $H_\alpha(J)$+count($u$)  }
End Algorithm 2
```

Phase 2: Creating association rules

Only when confidence crosses the threshold determined by the user can the rule be established.

4 AN ELABORATION FOR THE PROPOASED APPROACH

This section uses an example to illustrate how XML documents in a native XML database are analyzed and transformed into a FP-split tree, as well as how association rules with complete information are mined. DTD in Figure 2 is applied to define the structure of XML documents. Based on the two definition methods, six XML documents are derived as seem in Figure 3.

Step 1. Parsing DTD or XML Schema

Algorithm 1 is utilized to parse a DTD in Figure 8 and acquire information about defined element names, number of element occurrence and the content model of elements. A set of element names *TG*={transaction log, customer data, transaction item, identifier, gender, food, supply, electrical appliance, snack, bread, drink, bath, cleanser, other, Audio, air conditioning}. The minimum occurrence count of each element is f^0(transaction log)=1, f^0(customer data)=1, f^0(transaction item)=1, f^0(identifier)=1, …, and f^0(air conditioning)=0. The content model of each element is f^C(transaction log)=0, f^C(customer data)=0, f^C(transaction item)=0, f^C(identifier)=,…, and f^C(air conditioning)=1. Results are shown in Table 4. All elements are categorized into the sets of *TS*, *TM* and *TO*.

Elements with content model of sub-element and least occurrence count of zero are listed in *TS* set, *TS*={food, supply, electrical appliance}. Elements with content model of character data and least occurrence count of 1 are listed in *TM* set, *TM*={identifier, gender}. Elements with content model of character data and least occurrence count of zero are listed in *TO* set, *TO*={snack, bread, drink, bath, cleanser, other, audio, air conditioning}. This categorization is summarized in Table 5. In DTD, elements of “transaction log”, “customer data” and “transaction item” are sub-elements in terms of content model, and their least occurrence count is one. It indicates that these three elements are bound to occur in every XML document. In addition, they are not used to describe character data. Therefore, it is trivial to mine these elements.

Step 2. Creating equivalence classes of items

After DTD or XML Schema is parsed, element names in *TS*, *TM* and *TO* are matched with tag names in each document to establish equivalence classes of items as described in Algorithm 2.

Take the six XML documents in Figure 9 for instance. In the first XML document, we can find that {food, supply} $\in$ TS. Based on Case 1 of creating equivalence classes, equivalence classes of tag are created, i.e. EC_{food} ={1} and EC_{supply} ={1}. {identifier, gender} $\in$ TM.

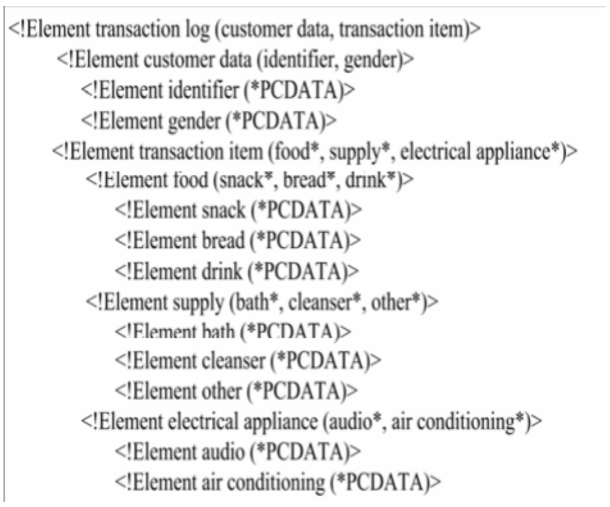
According to Case 2, equivalence classes of character data can be created—$EC_{\text{identifier (A001)}}$ ={1} and $EC_{\text{gender(male)}}$ ={1}. {snack, drink, other} $\in$ *TO*.

According to Case 3, not only equivalence classes of tag but also equivalence classes of character data can be created, i.e. $EC_{\text{food(snack)}}$ ={1}, $EC_{\text{food(drink)}}$ ={1} and $EC_{\text{suppl y(other)}}$ ={1}, and the equivalence classes of character data these tags describe, $EC_{\text{food(snack(puff))}}$ ={1}, $EC_{\text{food(drink(beer))}}$ ={1}, $EC_{\text{suppl y(other(diaper))}}$ ={1} and $EC_{\text{suppl y(other(feeding bottle))}}$ ={1}. The set of equivalence classes in the all XML documents are listed in Table 6.

Step 3. Support of equivalence classes

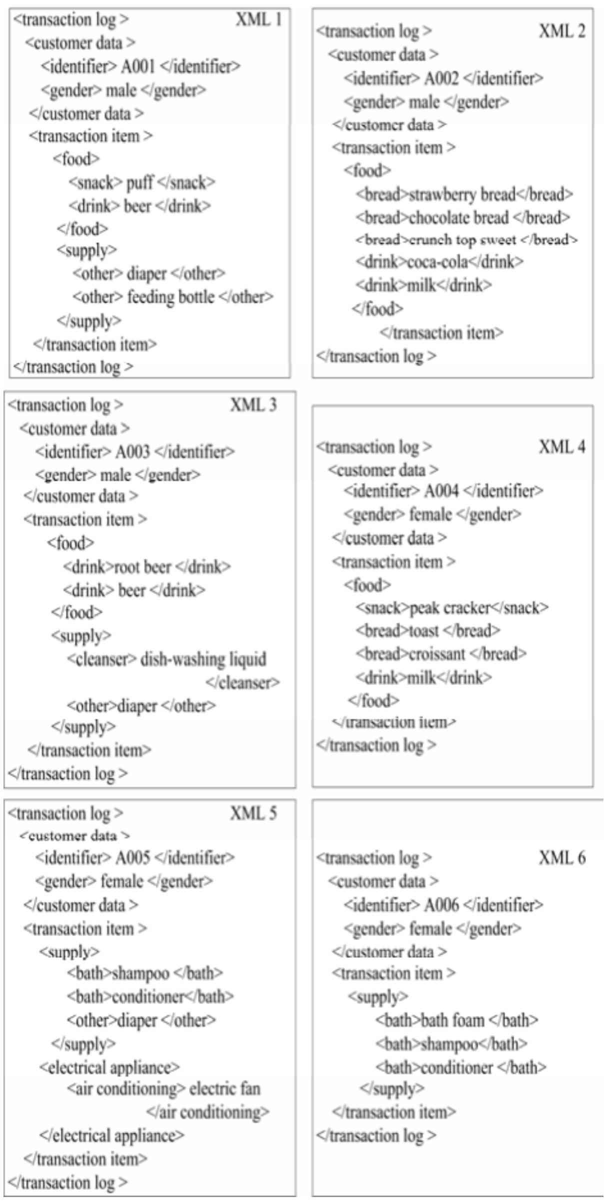
When creating each EC_ι, its support can be computed at the same time to find frequent items. Support of each item ι is the number of XML documents contained in the equivalence class of item. Therefore, $|EC_{\text{identifier(A001)}}|$=1, $|EC_{\text{gender(male)}}|$=3, $|EC_{\text{food}}|$=4, $|EC_{\text{food(snack)}}|$=2, $|EC_{\text{food(drink)}}|$=4,…, $|EC_{\text{supply(bath(bath foam))}}|$=1.

We determine that the minimum support is 2. EC_ι under the minimum support is deleted, and EC_ι over the threshold is regarded as a frequent item.



```
<!Element transaction log (customer data, transaction item)>
    <!Element customer data (identifier, gender)>
        <!Element identifier (*PCDATA)>
        <!Element gender (*PCDATA)>
    <!Element transaction item (food*, supply*, electrical appliance*)>
        <!Element food (snack*, bread*, drink*)>
            <!Element snack (*PCDATA)>
            <!Element bread (*PCDATA)>
            <!Element drink (*PCDATA)>
        <!Element supply (bath*, cleanser*, other*)>
            <!Element bath (*PCDATA)>
            <!Element cleanser (*PCDATA)>
            <!Element other (*PCDATA)>
        <!Element electrical appliance (audio*, air conditioning*)>
            <!Element audio (*PCDATA)>
            <!Element air conditioning (*PCDATA)>
```

Figure 2: DTD of transaction data



```
XML 1
<transaction log >
  <customer data >
    <identifier> A001 </identifier>
    <gender> male </gender>
  </customer data >
  <transaction item >
    <food>
      <snack> puff </snack>
      <drink> beer </drink>
    </food>
    <supply>
      <other> diaper </other>
      <other> feeding bottle </other>
    </supply>
  </transaction item>
</transaction log >
```

```
XML 2
<transaction log >
  <customer data >
    <identifier> A002 </identifier>
    <gender> male </gender>
  </customer data >
  <transaction item >
    <food>
      <bread>strawberry bread</bread>
      <bread>chocolate bread </bread>
      <bread>crunch top sweet </bread>
      <drink>coca-cola</drink>
      <drink>milk</drink>
    </food>
      </transaction item>
</transaction log >
```

```
XML 3
<transaction log >
  <customer data >
    <identifier> A003 </identifier>
    <gender> male </gender>
  </customer data >
  <transaction item >
    <food>
      <drink>root beer </drink>
      <drink> beer </drink>
    </food>
    <supply>
      <cleanser> dish-washing liquid
                               </cleanser>
      <other>diaper </other>
    </supply>
  </transaction item>
</transaction log >
```

```
XML 4
<transaction log >
  <customer data >
    <identifier> A004 </identifier>
    <gender> female </gender>
  </customer data >
  <transaction item >
    <food>
      <snack>peak cracker</snack>
      <bread>toast </bread>
      <bread>croissant </bread>
      <drink>milk</drink>
    </food>
  </transaction item>
</transaction log >
```

```
XML 5
<transaction log >
  <customer data >
    <identifier> A005 </identifier>
    <gender> female </gender>
  </customer data >
  <transaction item >
    <supply>
      <bath>shampoo </bath>
      <bath>conditioner</bath>
      <other>diaper </other>
    </supply>
    <electrical appliance>
      <air conditioning> electric fan
                      </air conditioning>
    </electrical appliance>
  </transaction item>
</transaction log >
```

```
XML 6
<transaction log >
  <customer data >
    <identifier> A006 </identifier>
    <gender> female </gender>
  </customer data >
  <transaction item >
    <supply>
      <bath>bath foam </bath>
      <bath>shampoo</bath>
      <bath>conditioner </bath>
    </supply>
  </transaction item>
</transaction log >
```

Figure 3: Content of the six documents

Table 4: Information of element

TG	f^{θ}	f^{C}	*TG*	f^{θ}	f^{C}
transaction log	1	0	snack	0	1
customer data	1	0	bread	0	1
transaction item	1	0	drink	0	1
identifier	1	1	bath	0	1
gender	1	1	cleanser	0	1
food	0	0	other	0	1
supply	0	0	audio	0	1
electrical appliance	0	0	air conditioning	0	1

Table 5: Element name in the set of TS, TM, and To

	TS	TM	TO
Element name	food, supply, electrical appliance	identifier, gender	snack, bread, drink, bath, cleanser, other, audio, air conditioning

Table 6: The set of equivalence classes in all documents

Item	EC	Item	EC
identifier(A001)	1	food(drink(root beer))	3
gender(male)	1, 2, 3	supply(cleanser)	3
food	1, 2, 3, 4	supply(cleanser(dish-washing liquid))	3
food(snack)	1, 4	identifier(A004)	4
food(drink)	1, 2, 3, 4	gender(female)	4, 5, 6
food(snack(puff))	1	food(snack(peak cracker))	4
food(drink(beer))	1, 3	food(bread(toast))	4
supply	1, 3, 5, 6	food(bread(croissant))	4
supply(other)	1, 3	identifier(A005)	5
supply(other(diaper))	1, 3	supply(bath)	5, 6
supply(other(feeding bottle))	1	supply(bath(shampoo))	5, 6
identifier(A002)	2	supply(bath(conditioner))	5, 6
food(bread)	2, 4	electrical appliance	5
food(bread(strawberry bread))	2	electrical appliance(air conditioning)	5
food(bread(chocolate bread))	2	electrical appliance(air conditioning(electric fan))	5
food(bread(crunch top sweet roll))	2	identifier(A006)	6
food(drink(milk))	2, 4	supply(bath(bath foam))	6
identifier(A003)	3		

Step 4. FP-split Algorithm

After frequent items are created, they are orderly positioned in a FP-split tree. First a simulated root node has to be created. A header table has to be created to note down the position of each frequent item in the FP-split tree.

Then, the first frequent item EC_{food} is transformed into a node (N_1). Since there is no child node under root node in the beginning, according to Rule 1 of FP-split tree construction, N_1 is placed below root node directly and become its child node. At the same time, the column "Link" of the item "food" in the header table is linked to N_1.

Next, the second frequent item $EC_{\text{food(drink)}}$ is transformed into a node (N_2), which is then matched with N_1. Due to the reasons that the *List* content of N_1 completely include that of N_2, and that N_1 does not have any child root, based on the definition of Rule 2, N_2 is designated as the child root of N_2. Moreover, the "Link" column of "food(drink)" item in the header table is linked to N_2.

Similarly, EC_{supply} is transformed into a node (N_3), and matched with the nodes in the FP-split tree. According to the definition of Rule 4, the *List* content of N_3 is only partly the same with that of N_1, so N_3 has to be partitioned to create a node N_4. After the partition, the column "Link_sibling" of N_3 will be linked to N_4 immediately to preserve their relation, as shown in Figure 4. The content of *List* of N_3 is modified as {1, 3} according to Equation (3), and the *List* content of N_4 is modified as {5, 6} based on Equation (4).

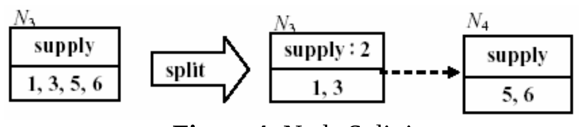
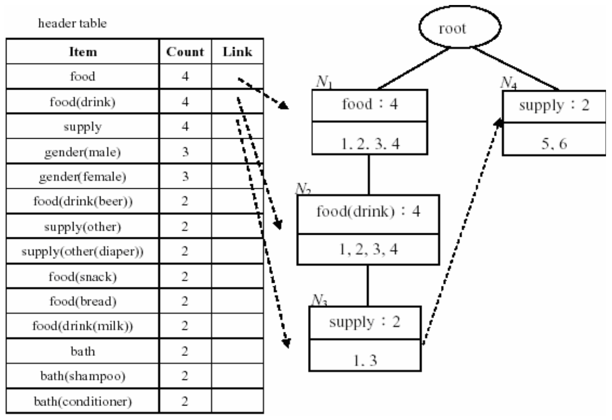


Figure 4: Node Splitting

After partitioning, it can be shown clearly that the *List* content of node N_1 completely includes that of N_3. Since N_1 has a child node N_2, based on Rule 2, N_3 has to be matched with N_2. The *List* content of node N_2 completely includes that of N_3, and N_2 does not have any child node. Therefore, according to Rule 2 again, N_3 becomes the child node of N_2.

The *List* content of node N_4 is completely different from that of N_1. In addition, N_1 does not have any sibling node. Based on Rule 3, N_4 is designated as the sibling node of N_1, as illustrated in Figure 5.

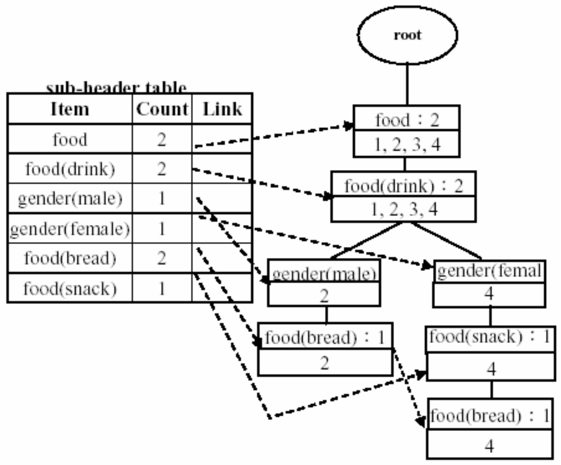


Item	Count	Link
food	4	
food(drink)	4	
supply	4	
gender(male)	3	
gender(female)	3	
food(drink(beer))	2	
supply(other)	2	
supply(other(diaper))	2	
food(snack)	2	
food(bread)	2	
food(drink(milk))	2	
bath	2	
bath(shampoo)	2	
bath(conditioner)	2	

Figure 5: Insert EC_{supply} into FP-split tree

Step 5. Mining association rules

After a complete FP-split tree is constructed, adaptive TD-FP-growth Algorithm is applied to mine the FP-split tree and create association rules. Figure 13 is an illustration of how adaptive TD-FP-growth Algorithm deals with each item in support order to create the frequent pattern associated with a given item. To created all frequent itemsets, the mining begins with item "food", and then "food(drink)", "supply" "gender(male)", etc. Take item α=food(drink(milk)) for example. From column "Link" in the header table, two paths related to item "food(drink(milk))" can be found—ζ_1=<food, food(drink), gender(male), food(bread)> and ζ_2=<food, food(drink), gender(female), food(snack), food(bread)>. At the same time, a sub-header table for "food(drink(milk))" is created to store all the items and count values in both paths. Values of *Count* and *Link_sibling* of the nodes in the FP-split tree are also adjusted, indicated in Figure 6.



Item	Count	Link
food	2	
food(drink)	2	
gender(male)	1	
gender(female)	1	
food(bread)	2	
food(snack)	1	

Figure 6: The mining of paths related to "food(drink(milk))" and the sub-header table

Next, items which do not cross the threshold in the sub-header table are deleted, and trivial items belonging to "food(drink(milk))" are filtered out as well. Only item "food(bread)" is left, and a frequent itemset with a length of 2 is created, i.e., "food(bread)$\cup$food(drink(milk))." Similarly, a sub-header table for the itemset with the length of 2 is established, values of *Count* and *Link_sibling* of nodes are adjusted to search for itemsets with the length of 3. The process is repeated until the itemset with the length of *k* is created.

5 EXPERIMENTAL RESULTS

In this section, an experimental program is designed to assess the efficiency of FP-split Algorithm proposed in this study in terms of mining of association rules. The experiment will be investigated in two parts. In the first part, a comparison will be made between FP-split Algorithm and FP-tree Algorithm. In the second part, the efficiency of FP-split Algorithm in mining native XML databases will be discussed.

5.1 Experiment introduction

The test data used in the first part of the experiment is from Assoc.gen provided by IBM Almaden Research Center [8]. Assoc.gen is a synthetic data generator, and its source code can be downloaded from IBM website. In the second part of the experiment, we write a program to synthesize an XML document. Parameters in Table 7 are used to establish experimental data with various properties.

Table 7: Parameter description

Parameter code	Description
T	Average transaction size
D	Transaction numbers
$\overline{T}$	Average number of item in XML document
$\overline{D}$	XML document numbers
N	Total number of item in DB
I	Average maximal frequent itemset size
Note: k is as 1000	

5.2 Experimental analysis of FP-split Algorithm and FP-tree Algorithm

FP-tree Algorithm can only be applied in conventional transaction databases, and is not feasible for native XML databases. FP-split algorithm, can be applied in both types of database. In this section, the efficiency of FP-split Algorithm and FP-tree Algorithm will be contrasted in a conventional transaction database.

The synthetic data generator Assoc.gen will be used to create several groups of transaction data. Various parameters are designed as well to prove that FP-split Algorithm is superior to FP-tree Algorithm proposed by Han et al. [6] in terms of execution duration.

5.2.1 Comparative analysis with various minimum supports

Figure 7 and Figure 8 exhibit comparisons with the same parameter settings T20.I10.D100k.N1k. The threshold of minimum support is varied to assess the performance of FP-split and FP-tree Algorithms.

The mining time indicated in Figure 7 includes time spent on scanning the database, constructing the tree and creating frequent itemsets. With the raising of the threshold of minimum support, the execution time is decreased for both TD-FP-growth based on FP-split tree and TD-FP-growth based on FP-tree. When the threshold is raised, the number of frequent items is decreased, therefore reducing the time required to construct the tree and create frequent itemsets (see Figure 7). The reason is that the tree construction with FP-split Algorithm is more timesaving than FP-tree Algorithm (see Figure 8).

In Figure 8, we have the experiment comparison by T20.I10.D100k.N1k. The efficiency of FP-split and FP-tree construction algorithms can be evaluated by adjusting the value of minimum supports. The run time in Figure 8 includes the time spent in scanning the database and constructing tree. When the minimum support is set to be 4%, FP-split algorithm saves as many as four times in run time than that of FP-tree construction algorithm. When the value of minimum support goes up to 10%, the difference between the two algorithms is double.

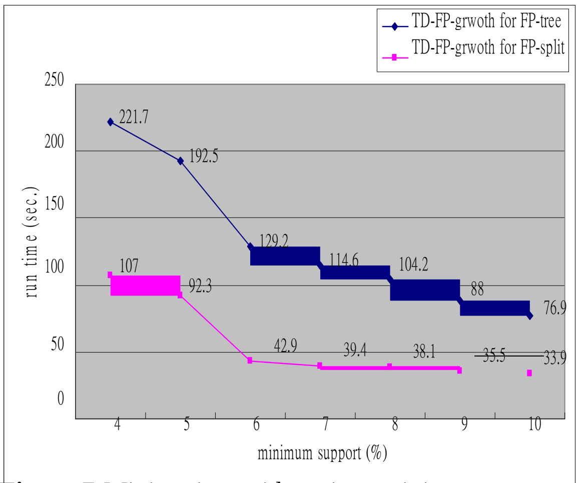


Figure 7: Mining time with various minimum supports

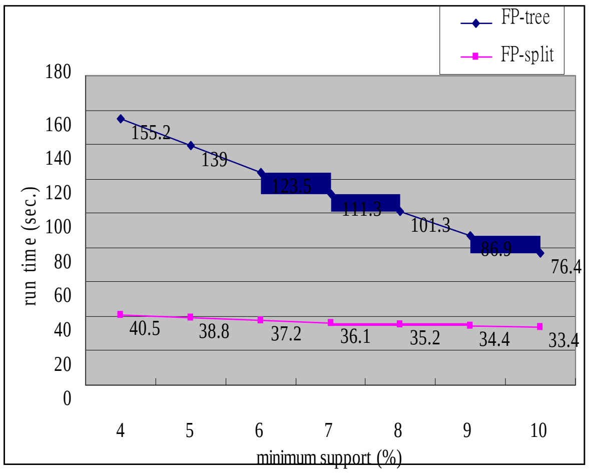


Figure 8: The run time with various minimum supports

5.2.2 Comparative analysis with various average transaction size

In Figure 9, we have the efficiency evaluation when the data parameters are set to be I10.D100k.N1k and the minimum support is set to be 7% by setting different average transaction items. The figure shows that out proposed FP-split algorithm is superior to FP-tree construction algorithm. There are three reasons such that our proposed method outperformed. The first reason is the more the average transaction size is, the longer time FP-tree construction algorithm takes to execute. This is because the longer the transaction size is, the more time it takes to scan. The second one is that a longer average transaction size in the database will generate more frequent itemsets. Accordingly, more time is spent in repeatedly search header table for maintaining links. The last one is that FP-split method doesn’t filter non-frequent items by checking the transaction record, and nor does it reorder those frequent items in each transaction record.

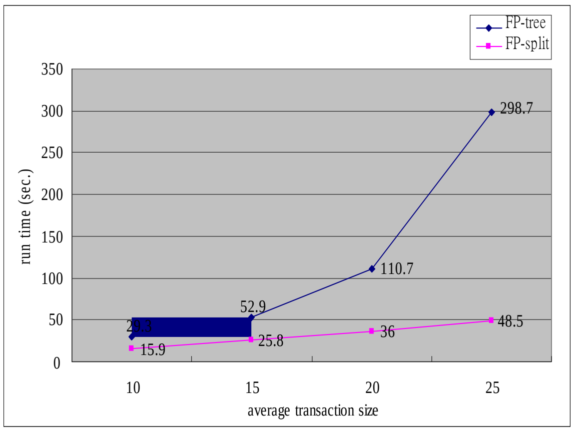


Figure 9: The run time with various average transaction size

5.2.3 Comparative analysis with various transaction numbers

In Figure 10, by adjusting different parameters of transaction numbers, we have the efficiency evaluation when the data parameters are set to be T20.I10.N1k and the minimum support is 7%. This figure shows that the more transaction records is, the more time it takes to execute. Meanwhile, the time difference between FP-split algorithm and FP-tree construction algorithm increases from tens of seconds spent for 100,000 transactions to hundreds of seconds for 500,000 transactions.

As FP-split algorithm scans the database only once but not twice as the FP-tree construction algorithm does, FP-split algorithm saves more time. In the event of large number of transaction records, the I/O cost remains much less compared with that of FP-tree construction algorithm.

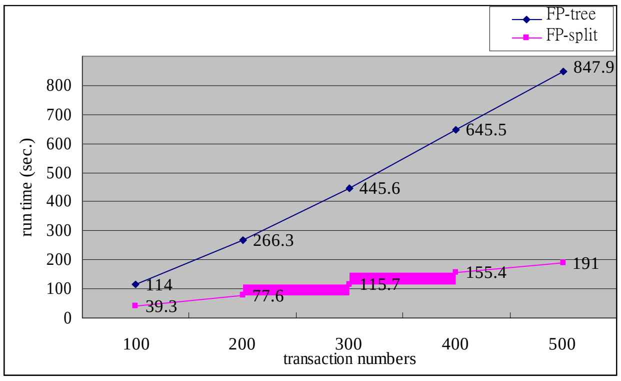


Figure 10: The run time with various transaction numbers

5.2.4 Comparative analysis with various item numbers

In Figure 11 the setting of data parameter is T20.I10.D100k, and the minimum support is 1%. Item numbers are varied to evaluate the efficiency of tree construction. Figure 11 shows that, regardless of item number, FP-split Algorithm is more efficient than FP-tree Algorithm in constructing the tree.

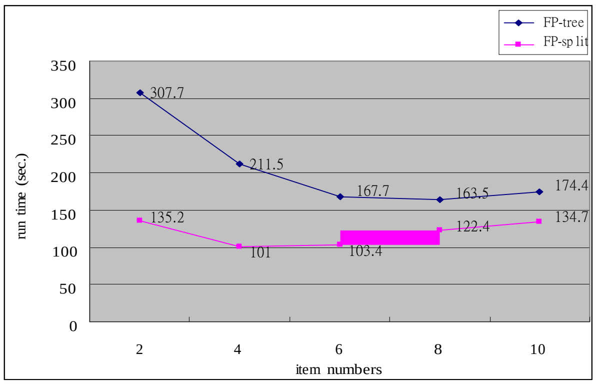


Figure 11: The run time with various item numbers

5.2.5 Assessment of memory utility rate

Figure 12 illustrates the utility rate of memory with the data parameter set at T20.I10.N1k, the minimum support being 7% and the quantity of transaction being varied. FP-split Algorithm is different from FP-tree Algorithm in that it has a "List" column in the node structure. The Link column marks down the transaction record in which each item occurs, so we can search for trees created by different transactions, and calculate how much memory is occupied by the List columns of all nodes.

Since FP-split Algorithm is written with the Java program, an integer takes up a space of 4Bytes. From Figure 12, it can be discovered that when the quantity of transaction increases, FP-split Algorithm occupies more memory space than FP-tree. Even so, the high time cost FP-tree Algorithm bears is greatly improved in FP-split Algorithm.

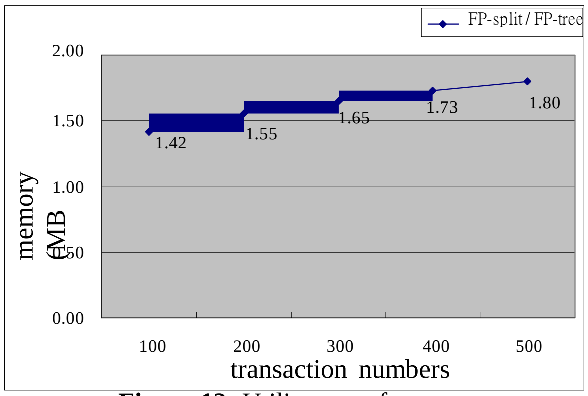


Figure 12: Utility rate of memory

5.2.6 Comparisons of FP-split Algorithm and FP-tree Algorithm

FP-split Algorithm is superior to FP-tree Algorithm in the execution efficiency of tree construction for three reasons. Their differences are summarized in Table 8.

1. FP-split Algorithm scans the database only once, while FP-tree Algorithm has to scan twice.
2. With FP-split Algorithm, only candidate items with a length of 1 need to be sorted and filtered. With FP-tree Algorithm, not only candidate items with a length of 1 but all items in every transaction log needs to be sorted and filtered.

3. Whenever a node is added to the tree, FP-split Algorithm is not required to repeat the search of Link between the header table and the node. However, with FP-tree Algorithm, the search has to be repeated to reserve the relation among nodes.

Table 8: Comparison of FP-split Algorithm with FP-tree Algorithm

	FP-tree Algorithm	**FP-split Algorithm**
Frequency of scanning database	2	1
Frequency of sorting data and filtering out non-frequent items	m+1	1

5.3 Experimental analysis of mining native XML databases

In this section, XML documents with various parameter settings are used to assess the mining efficiency of native XML databases by FP-split algorithm. In addition, comparisons are also made concerning the mining of native XML databases and conventional transaction databases with FP-split Algorithm.

5.3.1 Comparative analysis with different settings of XML document number, average transaction length, and support

The curves in Figure 13 and Figure 14 represent three sets of data parameter, D10k.T25.N1k, D30k.T25.N1k and D50k.T25.N1k, and D10k.T10.N1k, D10k.T20.N1k and D10k.T30.N1k. The analysis is based on different numbers of XML document, average transaction lengths and minimum supports.

Figures 13 and 14 indicate that increased document number and transaction length result in prolonged mining time. This is because the time spent on I/O and tree construction expands, nodes grow in quantity, and therefore the time to create frequent itemsets is affected.

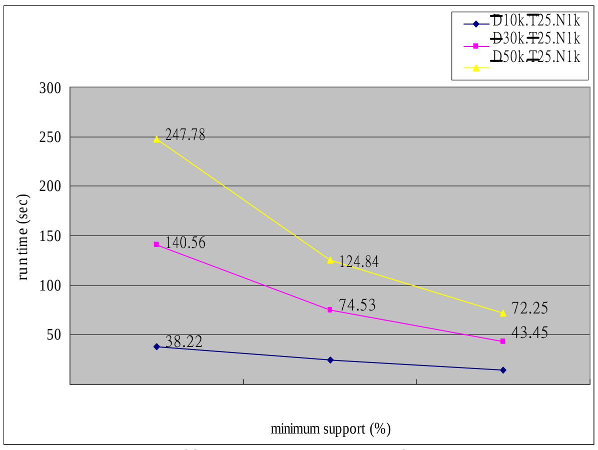


Figure 13: Different numbers of documents and minimum supports

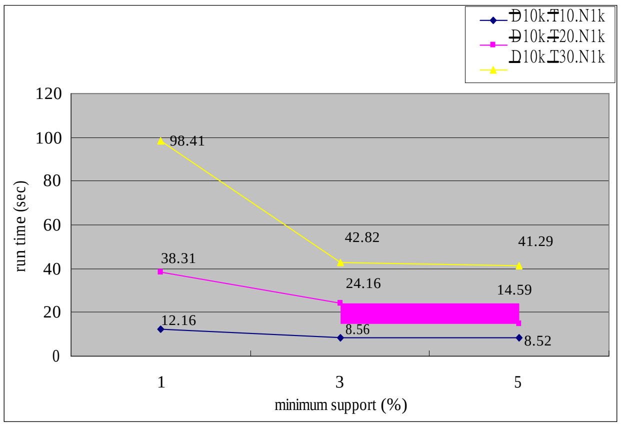


Figure 14: Different average transaction lengths and minimum supports

5.3.2 Comparative analysis on the mining of native XML databases and transaction databases

In this section, we estimate the execution time to exploit association rules from native XML databases and transaction databases. In addition, the exceution time of mining association rules from transaction databases also contains transfering XML document into transaction databases.

Figure 15 illustrates the assessment of mining efficiency when the data parameter is set to be T20.N1k and $\overline{T}$ 20.N1k, the minimum support is 3%, and the number of XML documents and transaction records are varied. The mining time is a total duration of database scanning, tree construction, generation of frequent itemsets and transfer phase. Figure 16 shows the individual time span.

In Figure 15, the curve "FP-split on XML" indicates the mining result of the native XML database with FP-split algorithm, and the curve "FP-split on TDB" indicates the mining result of the transaction database with FP-split algorithm. The comparison of these two curves show that, with the same data parameter, it spends more time mining the transaction database than mining the native XML database, because mining association rules from the transaction database is extra spending much time on transfer XML document into the transaction database.

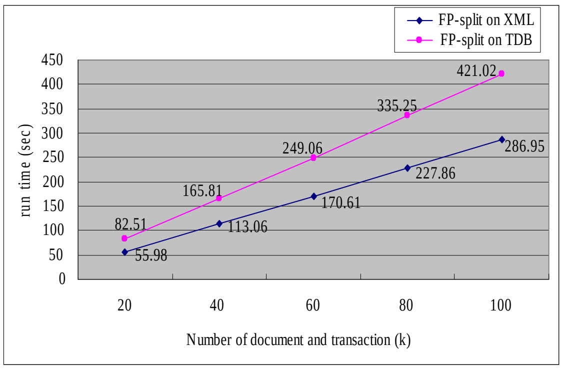


Figure 15: Mining of native XML databases and transaction databases

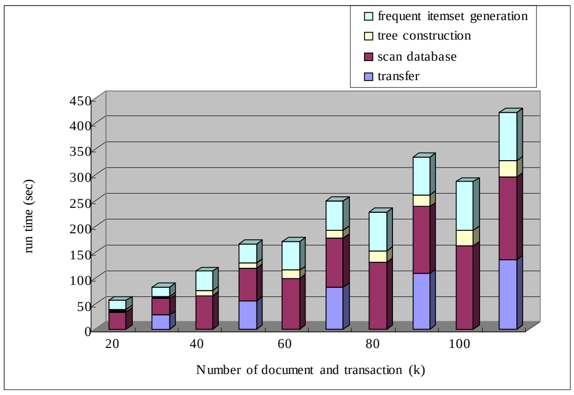


Figure 16: The individual time span

6 CONCLUSIONS

In this paper, we proposed a fast algorithm called Frequent Pattern Split method for extracting complete information from native XML databases. The FP-split method can easily and efficiently aid users to exploit association rules from large number of XML documents of the same structure by parsing DTD and XML schema without needing to understand both the structure of XML documents and their corresponding syntax. In addition, our proposed method can mine multi-level association rules between character data and tags.

Finally, we show that our proposed method is a fast association rule mining approach by experiment with various parameters. The FP-split method cannot only be applied to native XML databases but also efficiently applied to transaction databases for mining association rules.

7 REFERENCES

[1] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules between Sets of Items in Large Databases," Proceedings of the ACM SIGMOD Conference on Management of Data, pp. 207-216 (1993).

[2] D. Braga, A. Campi, M. Klemettinen, and P. Lanzi, "Mining Association Rules from XML Data," Proceedings of the 4th International Conference on Data Warehousing and Knowledge Discovery (2002).

[3] S. Chan, T. Dillon, and A. Siu, "Applying a Mediator Architecture Employing XML to Retailing Inventory Control," The Journal of Systems and Software, Vol.60, pp. 239-248 (2002).

[4] J. Fong, H. K. Wong, and Z. Cheng, "Converting Relational Database into XML Documents with DOM," Information and Software Technology, Vol. 45, pp. 335-355 (2003).

[5] J. Han and M. Kamber, "Data Mining: Concepts and Techniques," CA: Morgan Kaufmann Publishers (2001).

[6] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," In Data Mining and Knowledge Discovery, Vol. 8, pp. 53-87 (2004).

[7] H. Ishikawa and M. Ohta, "A Decentralized XML Database Approach to Electronic Commerce" Proceedings of the 5th International Symposium on Autonomous Decentralized Systems, pp. 153-160 (2001).

[8] IBM Almaden Research Center, Quest Synthetic Data Generation, http://www.almaden.ibm.com/software/quest/Resources/datasets/syndata.html (2005).

[9] C. F. Lee, S.W. Changchien, and W.T. Wang, "Association Rules Mining for Native XML Database," Department of Information Management, Chaoyang University of Technology, CYUT-IM-TR-2003-011 (2003).

[10] J. W. Lee, K. Lee, and W. Kim, "Preparations for Semantics-Based XML Mining," Proceedings of the IEEE International Conference on Data Mining, pp. 345-352(2001).

[11] C. H. Moh, E. P. Lim, and W. K. Ng, "DTD-Miner: A Tool for Mining DTD from XML Documents," Proceedings of the 2nd International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems, pp. 144-151 (2000).

[12] J. Shanmugasundaram, E. Shekita, R. Barr, M. Carey, B. Lindsay, H. Pirahesh, and B. Reinwald, "Efficiently Publishing Relational Data as XML Documents," Proceedings of the 26th International Conference on Very Large Databases, pp. 65-76(2000).

[13] M. Ströbel, "An XML Schema Representation for the Communication Design of Electronic Negotiations," Computer Networks, Vol.39, pp. 661-680(2002).

[14] D. Suciu, "On Database Theory and XML," ACM SIGMOD Special Section on Advanced XML Data Processing, Vol. 30, Issue 3, pp. 39-45(2001).

[15] J. W. Wan and G. Dobbie (2003), "Extracting Association Rules from XML Documents Using XQuery," Proceedings of the 5th ACM International Workshop on Web information and Data Management, pp. 94-97.

[16] J. W. Wan and G. Dobbie (2004), "Mining Association Rules from XML Data Using XQuery," Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation, Vol. 32, pp.169-174.

[17] K. Wang, L. Tang, J. Han, and J. Liu (2002), "Top Down FP-Growth for Association Rule Mining," Proceedings of the 6th Pacific-Asia Conference on Advances in Knowledge Discovery and Data Mining, pp. 334-340.

[18] Q. Wei and G. Chen (1999), "Mining Generalized Association Rules with Fuzzy Taxonomic Structures," Proceedings of the 4th America Fuzzy Information Processing Society, pp. 477-481.

[19] L. H. Yang, M. L. Lee, W. Hsu, and S. Acharya (2003), "Mining Frequent Query Patterns from XML Queries," Proceedings of the 8th International Conference on Database Systems for Advanced Applications, pp. 355-362.