Received March 3, 2021, accepted March 21, 2021, date of publication April 8, 2021, date of current version April 16, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3071777

Incremental Association Rule Mining With a Fast Incremental Updating Frequent Pattern Growth Algorithm

WANNASIRI THURACHON AND WORAPOJ KREESURADEJ
Faculty of Information Technology, King Mongkut's Institute of Technology Ladkrabang, Bangkok 10520, Thailand
Corresponding author: Worapoj Kreesuradej (worapoj@it.kmitl.ac.th)

This work was supported by the Faculty of Information Technology, King Mongkut's Institute of Technology Ladkrabang, Thailand.

ABSTRACT One of the most challenging tasks in association rule mining is that when a new incremental database is added to an original database, some existing frequent itemsets may become infrequent itemsets and vice versa. As a result, some previous association rules may become invalid and some new association rules may emerge. We designed a new, more efficient approach for incremental association rule mining using a Fast Incremental Updating Frequent Pattern growth algorithm (FIUFP-Growth), a new Incremental Conditional Pattern tree (ICP-tree), and a compact sub-tree suitable for incremental mining of frequent itemsets. This algorithm retrieves previous frequent itemsets that have already been mined from the original database and their support counts then use them to efficiently mine frequent itemsets from the updated database and ICP-tree, reducing the number of rescans of the original database. Our algorithm reduced usages of resource and time for unnecessary sub-tree construction compared to individual FP-Growth, FUFP-tree maintenance, Pre-FUFP, and FCFPIM algorithms. From the results, at 3% minimum support threshold, the average execution time for pattern growth mining of our algorithm performs 46% faster than FP-Growth, FUFP-tree, Pre-FUFP, and FCFPIM. This approach to incremental association rule mining and our experimental findings may directly benefit designers and developers of computer business intelligence methods.

INDEX TERMS Association rule mining, data mining, FP-tree, FP-growth, FPISC-tree, frequent itemset mining, incremental association rule mining.

I. INTRODUCTION

Association rule mining [1] is a well-known and widely used technique in data mining [2]; it has been used to mine patterns or relationships between sets of data in a large database. It has been widely applied in medicine, education, and business [3]–[7]. In general, association rule mining consists of two major sub-tasks [8]: first, frequent itemset generation, finding frequent itemsets, that satisfy a minimum support threshold, and second, association rule generation, from the derived frequent itemsets, that satisfy a minimum confidence threshold in the form of $A \Rightarrow B$. Most researchers have focused on improving the efficiency of frequent itemset mining, because it usually requires considerable resources and compute time.

During the past decade, many researchers have developed algorithms for finding frequent itemset. A conventional algorithm for finding frequent itemset is the Apriori algorithm [10]: it generates many candidate itemsets and requires multiple database scans, resulting in significant wasted resources and compute time. The FP-Growth algorithm [11] has been used to solve this problem, without candidate itemsets generation, reducing the number of database scans. Therefore, the FP-Growth algorithm is more efficient than Apriori. Nevertheless, it still wastes some resources and compute time for constructing multiple sub-trees. The Eclat algorithm [17] used a vertical data format for frequent itemset mining: it processes transaction data in a vertical data format, item-TID i.e. {item: set of TIDs}, where item is an item name, and TID refers to a set of transaction numbers for transactions containing the item.

Frequent itemset mining algorithms can be classified into three groups [9]: 1) Apriori-based algorithms that

The associate editor coordinating the review of this manuscript and approving it for publication was Junchi Yan.

generate candidate item for finding frequent itemset [10], 2) FP-Growth-based algorithms that construct frequent pattern tree (FP-tree) and find frequent itemsets from the FP-tree [11]–[16], and 3) algorithms that use vertical data format [17]. Still, there are other new algorithms such as Partition algorithm for association rule mining that partitions database transactions into clusters [36].

A main challenging problem in association rule mining had always been the problem that existing frequent itemsets and association rules might become invalid, as new transaction data or an incremental database was added to the original database, and some new frequent itemsets and association rules might have to be generated. The initial solution, for solving such a problem, is to mine all frequent itemsets and generate association rules from the full updated database, i.e., the incremental database plus the original database. However, this approach is inefficient and wastes resources and compute time. To deal with this problem, several algorithms have been developed, which may divide into majorly two categories [18]: Apriori-based algorithms [19]–[25] and FP-Growth-based algorithms [26]–[30]. Here, we focused on FP-Growth-based algorithms. The key problem of incremental association rule mining based on FP-Growth-based algorithms [31] is that the current FP-tree cannot be directly applied to an incremental database. Thus, FP-Growth-based algorithms generally focus on modifying FP-tree structure to avoid the overhead from database scans. Using one of these algorithms is more effective than rescanning the full updated database. However, this approach still wastes resources and compute time in constructing multiple conditional pattern trees or sub-trees, because it still uses the traditional FP-Growth algorithm to mine frequent itemsets from the updated FP-tree.

To improve the efficiency of the incremental frequent itemset mining, we developed a new approach for incremental association rule mining, a fast incremental updating frequent pattern growth algorithm (FIUFP-Growth), and designed a new type of sub-tree, incremental conditional pattern tree (ICP-tree), suitable for incremental frequent itemsets mining as well. The main idea was to retrieve previously discovered frequent itemsets from the original database and use them in mining all frequent itemsets from the updated FPISC-tree: this reduced the number of scans of the conditional pattern bases in the original path, when they were unnecessary and improved the efficiency in updating frequent itemsets, by reducing the construction of conditional pattern trees or sub-trees. It also reduced the sub-tree sizes, by constructing them only from conditional pattern bases of the new path. We discuss ICP-tree in detail in Section IV.

This paper is organized as follows: Section II briefly reviews related work; Section III introduces the background of incremental association rule mining; Section IV introduces our approach and shows an example that describes our approach; Section V presents and discusses experimental results; Section VI concludes; Section VII acknowledgment.

II. RELATED WORK

Association rule mining was first introduced by Agrawal *et al.* [1] in 1993. They presented an efficient method for discovery of significant relationships, between items in retail transactions. A year later, a well-accepted and simple algorithm called Apriori algorithm [10] was proposed for finding frequent itemsets and association rules. This algorithm generated and tested candidate itemsets, level by level, whether they were frequent or infrequent itemsets. However, a limitation to this idea was the generation of many candidate itemsets, which required multiple database scans, considerable storage space and compute time.

To avoid this problem, FP-Growth algorithm [11] and a compact data structure - a frequent pattern tree or FP-tree - that collected all frequent items from a transaction, was developed by Han *et al.* in 2000. With this technique, a database needed to be scanned only twice—the first time, for finding frequent itemsets, and, the second time, for constructing the FP-tree. The algorithm recursively constructed FP-trees to find all frequent itemsets, without generating candidate itemsets. However, in finding frequent itemsets, the algorithm needed to construct multiple conditional pattern trees, which still required significant resources and compute time.

In a dynamic database, when an incremental database was added to the original database, some previous association rules became invalid and some new association rules were generated. The basic and simple approach to solving this problem was to rescan the whole updated database to rediscover all frequent itemsets and association rules. However, this is time-consuming and inefficient and several efficient algorithms for dealing with the incremental association rule mining problem have been described, for example, Apriori-based algorithms, that include FUP [19], NBd [20], pre-large [21], probability-based algorithms [22]–[25].

The first algorithm that handled new transactions and updated association rules effectively was Cheung *et al.*'s Fast Update Algorithm (FUP) [19]: it focused on reducing unnecessary database scans, by partitioning itemsets, calculated from the incremental database, into four cases, based on previously found frequent itemsets or infrequent itemsets, from the original database. The algorithm needed to rescan the original database, only in one necessary case, saving some resources and time needed for the three unnecessary cases. The algorithm handled the incremental database that has been added more efficiently than the basic approaches.

A negative border algorithm (NBd), based on the FUP algorithm, reduced the number of original database scans. This algorithm stored both frequent and infrequent itemsets (border itemsets) of the original database, as well as their support counts, but used space to do so and many border itemsets could use significant storage space [20].

To deal with this problem, Hong *et al.*'s 'Pre-large' algorithm reduced the number of borders itemsets stored, and lower and upper support thresholds were introduced. As the algorithm runs, the itemsets with support counts, between the thresholds, were stored. These itemsets were called 'pre-large itemsets', i.e. infrequent itemsets that were expected to become frequent itemsets after the database was updated. The itemsets with support counts below the lower support threshold were not stored. Thus, the border itemset count was reduced [21].

FP-Growth is more efficient than Apriori algorithm [11], and many researchers have used FP-Growth-based algorithms for better management of frequent itemsets search in a dynamic database. Several previous works have used FP-Growth-based algorithms [26]–[30], [37]. We developed our new algorithm, starting with an FP-Growth-based algorithm.

Koh and Shieh's Adjusted FP-tree for Incremental Mining algorithm (AFPIM) [26] handled the incremental association rule mining problem, when new transactions were inserted into the original database and some existing transactions in the original database were deleted or modified. An updated FP-tree structure was computed by adjusting a previously constructed FP-tree structure from the original database, using the concepts of Pre-large algorithms. It does not need to rescan the original database in most cases. Updated frequent itemsets are mined from an updated FP-tree by applying the original FP-Growth algorithm.

Hong *et al*. developed a Fast Updated FP-tree (FUFP-tree) algorithm to effectively manage new transactions and improve the efficiency of construction and adjustment of the FP-tree structure, by reducing the number of rescans of the original database, after an incremental database was added. Their FUFP-tree added a bi-directional link for handling node insertion and deletion from the tree. The FUFP-tree algorithm improved the FUFP-tree structure, after an incremental database was added, similarly to the FUP algorithm [27].

A year later, Hong *et al*. demonstrated an improved FUFP-tree structure, that efficiently handled mining of incremental association rules, when transactions were deleted from the database: it reduced execution time, when transactions were deleted. After the FUFP-tree was updated, the algorithm continued to find all frequent itemsets from the updated FUFP tree by using the original FP-Growth algorithm [28].

Lin *et al*.'s Pre-FUFP maintenance algorithm [29] modified the construction and modification of the FUFP-tree based on the 'pre-large' concept: it defined upper and lower support thresholds. As a result, if the number of incremental transactions was less than a safety number, f, of new transactions, the original database did not need to be rescanned. The safety number was defined:

$$f = \left\lfloor \frac{(S_u - S_l)\, d}{1 - S_u} \right\rfloor \tag{1}$$

where S_u and S_l are the upper and lower support thresholds; and d is transaction count in the original database [29].

Sun *et al.* [37] proposed a solution for incremental frequent itemsets mining using a Full Compression Frequent Pattern Tree (FCFP-Tree) and a related algorithm called FCFPIM. The main advantage of FCFP-Tree is that it stores both frequent and infrequent items in compressed form to avoid repeated scans of the original. The original database will not be scanned to improve the structure of the new FCFP-tree when an incremental database is added. In addition, this algorithm works well in cases where the minimum support value is low because the size of the generated tree is no larger than that of the FP-tree that store the frequent itemset reposit. The main disadvantage that the strength of FCFPIM of which FCFP-tree stores both frequent and infrequent items is also its Achilles heels—the process of creating an FCFP-tree from a large original database will take a long time and use more memory than the processes of any other algorithms [ref]. However, incremental database updates will be much faster than a complete scan, so users will benefit from the most current data available anytime.

In addition, another Incremental association rule are in case of updating and deleting transactions in the original database [38]–[40].

Even though FP-Growth-based algorithms focus on updating the FP-tree structure which is better than rescanning the whole updated database, they still waste resources and time, for constructing multiple sub-trees, because they used the original FP-Growth algorithm to find frequent itemsets from the updated FP-tree. Those algorithms efficiently mined incremental association rules, by only adjusting the FP-tree structure, but they still used basic FP-Growth to mine all frequent itemsets from the updated FP-tree.

We introduced a new approach, our Fast Incremental Updating Frequent Pattern Growth Algorithm (FIUFP-Growth) for efficiently mining frequent itemsets from an updated FPISC-tree that we have described in [32]. This approach includes an introduction of a new incremental conditional pattern tree (ICP tree) to store and represent frequent itemsets along with their support counts in the updated and incremental databases, enabling more efficient handling of incremental frequent itemsets when new transactions are added. For solving an incremental association mining problem, FIUFP-Growth not only improves FP-tree structure but also improves the mining process of frequent itemsets from the updated FP-tree.

III. BACKGROUND

In this section, we briefly present the FUFP-tree maintenance algorithm [27], developed to deal with problems in incremental association rule mining based on the FP-Growth-based algorithm.

A. FUFP-TREE MAINTENANCE ALGORITHM

The FUFP-tree maintenance algorithm used an FUFP-tree to handle new transactions: the bi-directional link (mentioned already in Section II) made it easier to remove or delete items from the FUFP-tree. When constructing an FUFP-tree, the database was scanned only twice: The first scan found frequent items and sorted them in descending order, according to their support count, then created a header table according

TABLE 1. The original database.

TID	Item	Ordered frequent items
1	a,b,c,d,f	b,a,c,d,f
2	b,c,f	b,c,f
3	a,b,d,e	b,a,d,e
4	a,b,e	b,a,e
5	a,b,c,d	b,a,c,d
6	a,b,d,e	b,a,d,e
7	b,c	b,c
8	a,b,d	b,a,d
9	a,b,c	b,a,c
10	a,f	a,f

TABLE 2. The incremental database.

TID	Item
1	a,b,c,d,e
2	b,c,d,e
3	a,b,c,e,d
4	a,e,d
5	a,b,c,e,d

to the sorted frequent patterns. The second scan constructed an FUFP-tree from transactions, that only had items that matched frequent items in the header table. The items were sorted in descending order of support value, then a node was added to represent each item in the FUFP-tree to complete each transaction. As an example, Table 1 presents an original database, and Table 2 presents an incremental database. Fig. 1 presents an FUFP-tree constructed from the original database.

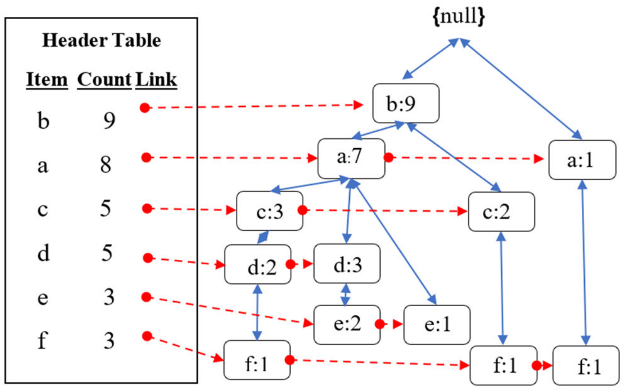

FIGURE 1. FUFP-tree constructed from an original database.

When a new database was added, as shown in Table 2, the FUFP-tree maintenance algorithm found items and their support count from new transactions. Then, it divided the items into four cases, based on FUP concept [19].

Case 1: The item is a frequent item in both the original and the incremental database. In this case, the item always becomes a frequent item in the updated database, and its support count in the updated database can be computed by adding the support counts in the original and the incremental database. Then, the support count is updated in the header table and FUFP-tree.

*Case 2:*The item is a frequent item in the original database but an infrequent item in the incremental database. The item may or may not become a frequent item in the updated database, but its support count in the updated database can be computed as described in Case 1. When the item becomes a frequent item in the updated database, its support count is updated in the header table and FUFP-tree. On the other hand, if the item becomes an infrequent item in the updated database, the item is removed from the header table and its nodes are removed from the FUFP-tree.

Case 3: The item is an infrequent item in the original database but a frequent item in the incremental database. In this case, a scan of the original database is needed to find its support count in the original database, and the support count in the updated database is computed by adding this support count to the count in the incremental database. When the item becomes a frequent item in the updated database, it is placed at the end of the header table and its nodes are added to the leaf node of a path in the FUFP-tree. On the other hand, when the item becomes an infrequent item in the updated database, it does not affect the header table or FUFP-tree.

Case 4: The item is an infrequent item in both original and incremental databases. In this case, the item remains an infrequent in the updated database. Thus, it does not affect the header table or the FUFP-tree.

For all cases, the algorithm updated the support count for each item in the updated database and compared it to the minimum support threshold. If the item is an infrequent item in both original and updated databases and if its support count in the updated database is less than the threshold, the FUFP-tree and header table were not modified. However, if the item was a frequent item in the original database, but became infrequent item in the updated database, the item in the header table and its nodes in the FUFP-tree were deleted. Moreover, if the support count for the item in the updated database exceeded the threshold, some modifications were made to the FUFP-tree and header table. If the item was a frequent item in the original database and remained a frequent item in the updated database, its support count in the FUFP-tree and header table was updated. In addition, if the item was an infrequent in the original database but became frequent in the updated database, the item and its support count were added to the header table and its nodes were added to the FUFP-tree. The updated FUFP-tree is shown as Fig. 2.

FUFP-tree algorithm utilizes the procedural steps in the embedded FP-growth algorithm to mine all frequent patterns. Therefore, multiple sub-trees are still needed to be created and cycled through which require a lot of computational time.

B. FREQUENT ITEMSET MINING BY FP-GROWTH

A frequent pattern growth mining [11] mines all frequent itemsets directly from a derived FP-tree without candidate itemset generation. It uses a divide-and-conquer strategy to mine frequent itemsets, in which the conditional pattern base

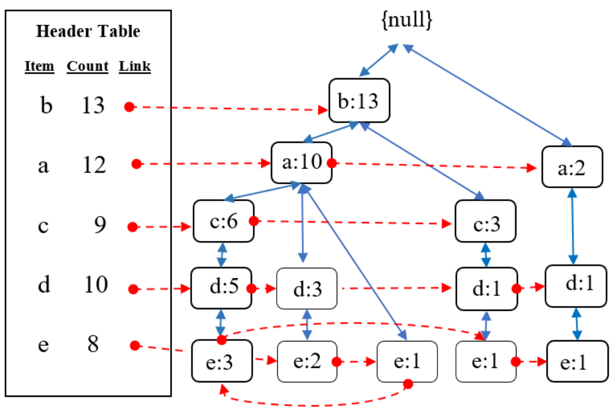

FIGURE 2. Updated FUFP-tree.

TABLE 3. Frequent itemsets mined with FP-growth algorithm.

Item	Frequent itemset
'*e*'	('*e*'): 8, ('*de*'): 7, ('*bde*'): 6, ('*abde*'): 5, ('*ade*'): 6, ('*be*'): 7, ('*abe*'): 6, ('*ae*'): 7
'*d*'	('*d*'): 10, ('*cd*'): 6, ('*bcd*'): 6, ('*acd*'): 5, ('*abcd*'): 5, ('*bd*'): 9, ('*abd*'):8, ('*ad*'):9
'*c*'	('*c*'): 9, ('*bc*'): 9, ('*ac*'): 6, ('*bac*'): 6
'*a*'	('*a*'): 12, ('*ba*'): 10
'*b*'	('*b*'): 13

and conditional pattern tree, of each item, are generated level by level. From a conditional pattern base, a conditional pattern tree or sub-tree is derived. Then, the sub-tree is scanned to generate frequent itemsets, associated with an item in the header table, and the frequent itemsets are extracted. The algorithm recursively creates additional conditional pattern bases of sub-items from this sub-tree. In each recursive call, a new conditional pattern base and sub-tree are constructed and more frequent itemsets are mined by concatenating an item in the header table to the lists of frequent itemsets generated from the sub-tree.

We introduce an example to illustrate the steps in frequent itemset mining from an updated FUFP-tree, in Fig. 2. We assume that the minimum support threshold was set at 30%, then the support count in the original database is 3, while that in the updated database is 5.

The header table initially contains '*b*', '*a*', '*c*', '*d*' and '*e*'. The algorithm started to mine the frequent itemsets by considering the item at the bottom of the header table first, i.e. item '*e*'. Finally, nine conditional pattern trees or sub-trees were constructed for mining frequent itemsets in this example. The conditional pattern tree (sub-tree) for item '*e*' is shown in Fig. 3, for item '*d*' in Fig. 4. and for items '*c*' and '*a*' in Fig. 5. All frequent itemsets from the updated database are shown in Table 3.

IV. OUR APPROACH

In this section, we introduce our new approach for improving the efficiency of frequent itemset mining called Fast Incremental-Updating Frequent Pattern growth algorithm (FIUFP-Growth). We also present a new FPISC-tree [32], a more efficient data structure for incremental association rules mining. Our algorithm is based on an FP-Growth algorithm. Improvements to FP-Growth-based algorithms by others changed the FP-tree structure, but they still used the FP-growth algorithm to discover all frequent itemsets from the updated FP-tree. Hence, they still wasted resources and time for constructing multiple sub-trees. In contrast, our new algorithm, not only improves the FP-tree structure, but also efficiently mines all frequent itemsets from the updated FP-tree. Moreover, we also designed a new conditional pattern tree (or sub-tree) to represent only frequent items from the incremental database, along with their support counts, called incremental conditional pattern tree (ICP-tree). As a result, there are fewer ICP-tree nodes than in the original and the incremental databases combined.

We describe our previously described FPISC-tree [32] in sub-section A, then the new FIUFP-Growth algorithm in sub-section B.

A. FPISC-TREE

As described previously [32], in a conventional FP-tree, each node has three key attributes: item name, support count, and node link. We added an incremental count so that each node of an FPISC-tree has four main attributes: item name, support count, node link, and incremental count. This count is the support count of an item that appears in the incremental database. In the FPISC-tree construction step, this attribute was initialized to 0, but in updating the FPISC-tree, this count increased if the item in the node appears in the new transaction, otherwise it was not changed. It was also useful for identifying whether a node is from the incremental database, the original database or both. This attribute enables the algorithm to separate conditional pattern bases into two groups: 1) conditional pattern bases, derived from the original database, and 2) conditional pattern bases, derived from the incremental database. In addition, the algorithm reduced the number of rescans of conditional pattern bases from the FPISC-tree. As shown in Fig. 6, the FPISC-tree was constructed from the original database, by scanning the original database only twice. Subsequently, when the incremental database was added to the original database, the FPISC-tree was updated. The incremental count of each item in the node increased, if the item appeared in the incremental database as well- see Fig. 7.

B. OUR FAST INCREMENTAL UPDATING FP-GROWTH ALGORITHM

Here, we describe an FIUFP-Growth algorithm for mining frequent itemsets from an updated FPISC-tree. We used previously discovered frequent itemsets and their support counts, based on the FUP concept, from the original database, in frequent itemsets mining, from the updated FPISC-tree. Furthermore, our algorithm constructs a conditional pattern tree or sub-tree from only the conditional pattern base of the incremental database. Unlike the FP-Growth algorithm, that first

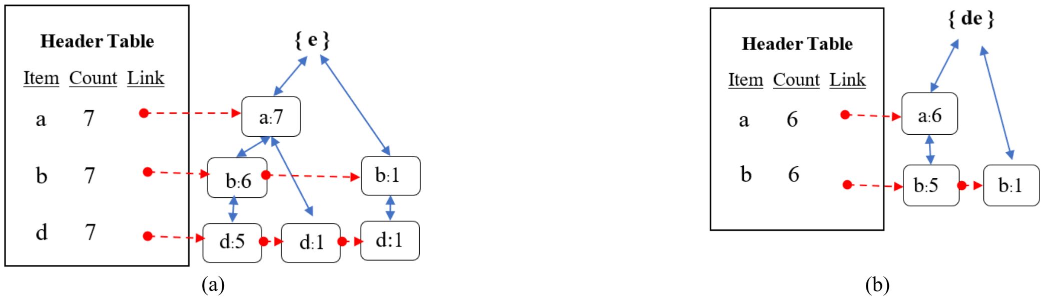

FIGURE 3. Conditional pattern tree (sub-tree) for item 'e.'

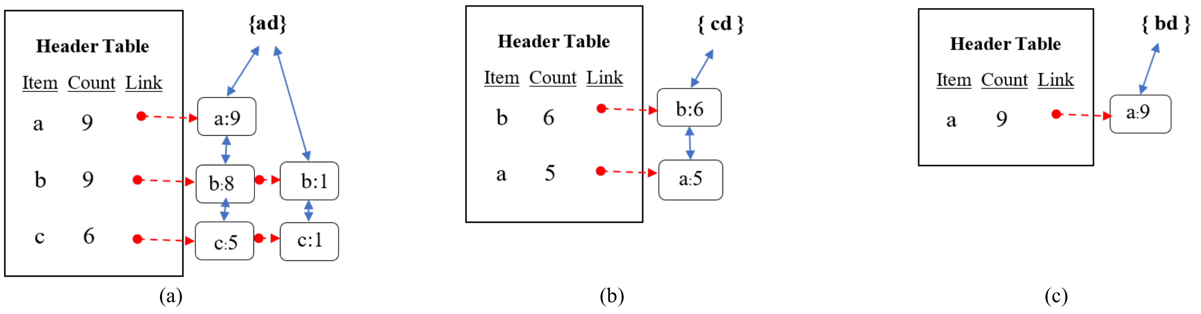

FIGURE 4. Conditional pattern tree (sub-tree) for item 'd.'

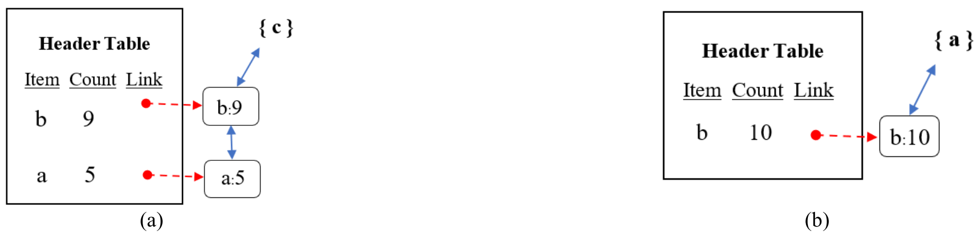

FIGURE 5. Conditional pattern trees (sub-trees) for item 'c' and 'a.'

constructs a sub-tree then generates frequent patterns, when the $(k + 1)$th item was added to an existing k item itemset, mining associated with the considered item, the conditional pattern base for the incremental database, that has already been constructed was used to construct the sub-tree, avoiding the need to construct unnecessary conditional pattern trees or sub-trees. The symbols used are defined below.

1) NOTATION

DB	original database
$db+$	incremental database
$\|DB\|$	number of transactions in DB
$\|db+\|$	number of transactions in $db+$
UDB	updated database, i.e. $\|DB\|+\|db+\|$
min_sup	minimum support threshold
$newFIS$	set of new, emerged frequent items
$originalFIS$	set of original frequent itemsets in the original database
$condNewPath$	set of prefix paths or the conditional pattern based of the incremental database in the FPISC-tree, where $addedCount \geq updatedCount$
$condOrgPath$	set of prefix paths or the conditional pattern based of the original database in the

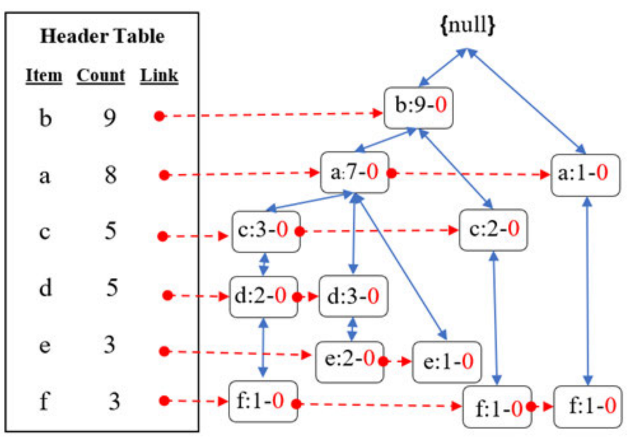

FIGURE 6. FPISC-tree constructed from the original database.

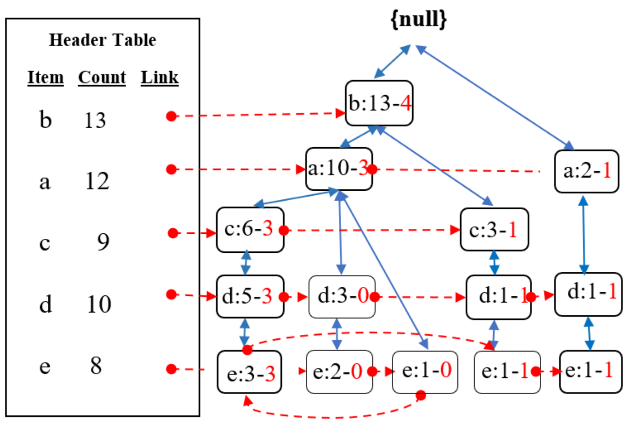

FIGURE 7. Updated FPISC-tree.

	FPISC-tree where *addedCount* $= 0 \vee$ *updatedCount* $>$ *addedCount*
updatedFIS	set of frequent itemsets in *UDB*
frequentList	set of items that are frequent items
infrequentList	set of infrequent items
fList	set of items that become frequent items after the database was updated.
root_item	set of items that contains the items and its sub-items in each iteration of frequent pattern mining
rescanOriginal	set of items left over from the step in case#1 and processed in case#3
β_m	pattern associated with itemset, *m*
β_{mDB}	previously discovered frequent pattern associated with itemset, *m,* in the original database

2) FIUFP-GROWTH ALGORITHM (FAST INCREMENTAL UPDATING FREQUENT PATTERN GROWTH)

FIUFP-Growth can be split into two algorithms: 1) the main algorithm in Algorithm 1; and 2) an incremental pattern growth algorithm in Algorithm 2. FIUFP-Growth starts to mine frequent itemsets from an updated FPISC-tree, as shown in Fig. 7, considering only the items derived from *condNewPath* for the incremental database, and then separates them into four cases as in the FUP algorithm.

Algorithm 1 Main Algorithm

```
Input: updated FPISC-tree, header_table, newFIS, min_sup, originalFIS.
Output: updatedFIS.
Step 1:
1)      for each item i in header_table
Step 2:
2)          if item i in newFIS then
3)              newFIS = call FP-Growth algorithm
              (updated FPISC-tree, header_table, i, min_sup)
4)              updatedFIS = updatecdFIS ∪ newFIS
5)          else
Step 3:
6)              k = 2
7)              insert item i into root_item
8)              generate condNewPath and condOrgPath
Step 4:
9)              for each item l in condNewPath
10                  if $l.supCount >= \lceil |db^+| * min\_sup \rceil$
11)                     insert l into freqeuntList
12)                 else
13)                     insert l into infrequentList
14)             end for
Step 5:
15)             newFIS = call Incremental Pattern Growth
              Algorithm (freqeuntList, infrequentList,
              root_item, originalFIS, fList, k, condNewPath,
              condOriginalPath, sub_item)
16)             updatedFIS=updatedFIS ∪ nesFIS
17)             root_item.remove('i')
18)     end for
19) return updatedFIS
```

The steps of the main algorithm are some as pseudo code in Algorithm 1, are expanded below.

Step 1: In line (1) in Fig. 8, for each item *i*, starting from the bottom to the top of the header table, check whether the item *i* is an emerged frequent item and appears in the set of *newFrequentItem* or not.

Step 2: if the item *i* is in *newFrequentItem*, then in line (2)-(5), all frequent items associated with the item *i* are mined out by the traditional FP-Growth algorithm; else go to Step 3-11.

Step 3: In line (6)-(7), the algorithm first inserts the item *i* into a set of *root_item* and set the initial value *k* = 2, then in line (8), it generates a conditional pattern base for the original database (*condOrgPath*) and the incremental database (*condNewPath*) by following the node-link of item *i*. A *condNewPath* is generated when *addedCount* equal to *count* or *addedCount* > 0 and

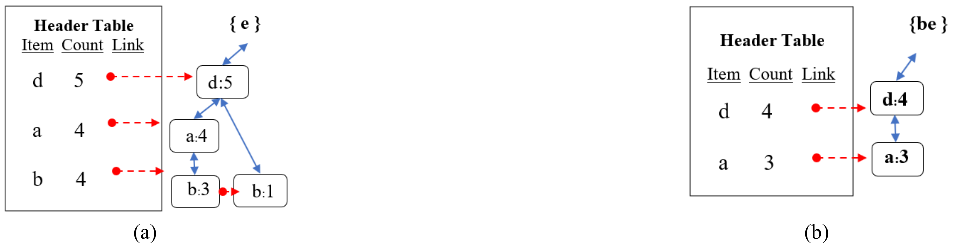

FIGURE 8. Incremental conditional pattern tree (ICP-tree) constructed for item 'e.'

Algorithm 2 Incremental Pattern Growth Algorithm

```
Input: frequentList, infrequentList, root_item,
       originalFIS, fList, k, condNewPath, condOrgPath,
       sub_item
Output: newFIS
Step 6:
1)     //Case#1:
2)     for each item m in frequentList
Step 7:
3)         generate new pattern $\beta_m = (m \cup root\_item)$
4)         if $\beta_m == \beta_{mDB} | \beta_{mDB} \in originalFIS$
5)           $\beta_m.supCount = \beta_{mDB}.supCount + m.supCount$
6)           if $\beta_m.supCount \geq \lceil |UDB| * min\_sup \rceil$ then
7)               $newFIS = newFIS \cup \beta_m$
8)               remove $\beta_{mDB}$ from originalFIS
9)               insert item m into fList
10)          else
11)              insert item m into rescanOriginal
12)    end for
Step 8:
13)    //Case#2:
14)    for each item n in infrequentList
15)        generate new pattern $\beta_n = (n \cup root\_item)$
16)        if $\beta_n == \beta_{nDB} | \beta_{nDB} \in originalFIS$
17)            $\beta_n.supCount = \beta_{nDB}.supCount + n.supCount$
18)            if $\beta_n.supCount \geq \lceil |UDB| * min\_sup \rceil$ then
19)                $newFIS = newFIS \cup \beta_n$
20)                remove $\beta_{nDB}$ from originalFIS 21)
insert item n into fList
22)    end for
```

Algorithm 2 *(Continued.)* Incremental Pattern Growth Algorithm

```
Step 9:
23)    //Case#3:
24)    for each item p in rescanOriginal
25)        $p.supCount_{DB} = count\ item\ p\ in\ \{T_{OP} | T_{OP} \in condOrgPath\}$
26)        $p.supCount_{UDB} = p.supCount_{DB} + p.supCount$
27)        if $p.supCount_{UDB} \geq \lceil |UDB| * min\_sup \rceil$ then
Step 10:
28)            generate new pattern $\beta_p = (p \cup root\_item)$
29)            $newFIS = newFIS \cup \beta_p$
30)            insert item p into fList
31)    end for
Step 11:
32)  if |fList| > 1 then
33)      ks = k + 1
Step 12:
34)      ICP_Tree, sub_header_table, sub_fList = call
         FP_treeConstruction (root_item,ks, kfList, condNewPath)
Step 13:
35)      for each item 'i' in sub_header_table
36)          sub_orgCondPath = condOrgPath
37)          insert item 'i' into sub_item and root_item
38)          generate sub_condNewPath from ICP_Tree
39)          generate sub_freqeuntList and sub_infrequentList
40)          insert item 'i' into root_item
41)          newFIS=call Incremental Pattern Growth
             Algorithm(sub_frequentList, sub_infrequentList, root_item, originalFIS,
             ks, sub_condNewPath, sub_orgCondPath, sub_item)
42)          α.remove('i')
43)          remove 'i' from sub_item and root_item
44)      end for
Step 14:
45)  return newFIS
```

addedCount is equal to *count* or *addedCount* > 0 and *condOrgPath* is generated when *addedCount* ≥ 0.

Step 4: In line (9)-(14), the algorithm counts the frequency of occurrences of each item *l* only in *condNewPath* and check whether each item *l.supCount* is greater than or equal to the support threshold in the incremental database. If so, it inserts the item *l* and its *supCount* into a set of *freqeuntList*. Otherwise, it inserts the item into a set of *infrequentList*.

Step 5: In line (15)-(18), the algorithm starts to mine frequent itemsets by recursively calling incremental pattern growth algorithm.

The procedural steps of the FIUFP-Growth algorithm shown in Algorithm 2 are as follows.

Step 6: For case#1, in line (1)-(2) the algorithm first generates a pattern β_m by joining each item m in the *freqeuntList* with all items in *root_item*.

Step 7: In line (3), the algorithm checks whether or not the pattern β_m is a frequent pattern (β_{mDB}) in the set of *originalFIS*. If so, go to sub-step 7-1; otherwise, go to sub-step 7-2.

sub-step 7-1: in line (5)-(9), the algorithm updates the support count of frequent pattern (β_{mDB}) by adding $\beta_m.supCount$ to $\beta_{mDB}.supCount$, then it verifies if $\beta_m.supCount$ is greater than or equal to the support threshold for the updated database; if so, the pattern β_m is added to the set of *newFIS*, β_{mDB} is deleted from *originalFIS*, and item m and its support count are inserted into an *fList* set.

sub-step 7-2: in line (10)-(11), the algorithm inserts item m and its support count into a *rescanOriginal* set because this pattern β_m is a frequent pattern in the incremental database but an infrequent pattern in the original database. Hence, this pattern falls under case#3 (**step 9-10).**

Step 8: For case#2, the algorithm generates pattern β_n by joining each item n in *infrequentList* with all items in the *root_item* and then checks whether or not β_n is a frequent pattern (β_{nDB}) in the *originalFIS* set. If so, go to sub-step 8-1; otherwise, do nothing.

sub-step 8-1: in line (16)-(22), the algorithm updates the support count for β_n by adding $\beta_n.supCount$ to $\beta_{nDB}.sup$ *Count*. After that, the algorithm verifies if $\beta_n.supCount$ is greater than or equal to the support threshold for the updated database; if so, pattern β_n is added to the *newFIS* set; pattern β_{nDB} is deleted from *originalFIS*; and item m and its support count are inserted into the *fList* set. Otherwise, the algorithm does nothing.

Step 9: For case#3, in line (23)-(31), the algorithm considers each item in the *rescanOriginal* set and then counting it in each path in the *condOrgPath* under the condition that each path must be a super-set of *sub_item*.

Step 10: Pattern β_p is generated by joining each item p in the set of *rescanOriginal* to all items in the *root_item* set. Then, the algorithm updates the support count of pattern β_p by adding $\beta_p.supCount$ to *supCount* from *condOrgPath*. After that, the algorithm verifies if $\beta_p.supCount$ is greater than or equal to the support threshold for the updated database; if so, go to sub-step 10-1. Otherwise, do nothing.

sub-step 10-1: pattern β_p is added to the *new-FIS* set, and item p and its support count in the incremental database is inserted into the *fList*.

Step 11: In line (32)-(43), the algorithm checks whether it will continue to mine (k + 1)item-long frequent itemsets ending with item i or not. If the number of all items in the *fList* set is greater than 1, continues to mine such frequent itemsets in step 12-14.

Step 12: In line (34), the algorithm constructs an increment FP-tree (ICP-tree) from only the incremental database (*condNewPath*) by utilizing the frequent items in the *fList* set, sorted in descending order according to its incremental support count.

Step 13: Next, in line (35)-(40), the algorithm continues to discover frequent itemsets starting from item $'i'$ from the bottom of the *sub_headertable* by recursively calling the incremental pattern growth algorithm until the number of all items in the *fList* set is less than or equal to 1.

Step 14: Finally, all (k + 1)-frequent itemsets are mined out.

TABLE 4. All Frequent itemsets mined out from the original database.

Item	Frequent itemset
'*f*'	('*f*'): 3
'*e*'	('*e*'): 3, ('*ae*'): 3, ('*be*'): 3, ('*abe*'): 3
'*d*'	('*d*'): 5, ('*ad*'): 5, ('*bd*'): 5, ('*abd*'): 5
'*c*'	('*c*'): 5, ('*bc*'): 5, ('*ac*'): 3, ('*abc*'): 3
'*a*'	('*a*'): 8, ('*ba*'): 7
'*b*'	('*b*'): 9

C. EXAMPLE OF STEPS OF FIUFP-GROWTH

An example to illustrate our approach follows. The FPISC-tree used is shown in Fig. 6. Frequent itemsets that were mined from the original database are shown in Table 4. When an incremental database was added, the FPISC-tree was improved based on FUFP-tree maintenance [32] as shown in Fig. 7. The minimum support was set at 30%; the support count in the original database was 3; the support count in the incremental database was 2, and the support count in the updated database was 5.

To compare performance of our algorithm with FP-Growth, FUFP-tree maintenance, Pre-FUFP, and FCFPIM algorithms on this newly added transaction problem, we set our algorithm to mine all frequent itemsets from the original FPISC-tree by using the FP-Growth algorithm, in a similar way to the basic FP-Growth, FUFP-tree algorithm or Pre-FUFP algorithm. However, the FP-Growth, FUFP-tree maintenance, Pre-FUFP, and FCFPIM algorithms still used the FP-Growth algorithm for mining all frequent itemsets from the updated FP-tree. Instead, our method found frequent itemsets from the updated FPISC-tree. Similar to FP-Growth, we started to mine frequent itemsets, associated with item '*e*' at the bottom of the header table. Numbers for the steps refer to steps in the pseudo code in Algorithms 1 and 2.

Step 1: **FIUFP-Growth algorithm** starts to mine frequent itemsets associated with item '*e*'.

Step 2: Item '*e*' is not a new frequent item or *newFrequentItem*, then go to Step 3-11.

Step 3: The algorithm generates a conditional pattern base for the incremental database (*condNewPath*) and the original database (*condOrgPath*). Three paths ending with node '*e*' are found in *condNewPat*h including ['*b*','*a*','*c*','*d*']:3, '*b*','*c*','*d*']:1, and ['*a*','*d*']:1, and two paths in the *condOrgPath* including ['*b*','*a*','*d*']:2, and ['*b*','*a*']:.

Step 4: The algorithm counts each item only from the *condNewPath* and then put them into either the *frequentList* or *infrequentList* by comparing the support count for each item with the support threshold of the incremental database which is 2; hence, the *frequentList* is now{'*a*':4,'*b*':4,'*d*':5,'*c*':4} and the *infrequentList* is{}.

Step 5: The algorithm starts to categorize the *frequentList* and *infrequentList*, that is{'*a*':4,'*b*':4,'*d*':5,'*c*':4} and{}, into four cases.

Step 6: For case#1, the algorithm generates patterns by concatenating each item in the *freqeuntList* with the root-item '*e*'. In this case, item '*a*':4 is processed first, then a pattern ('*ae*') is generated, followed by pattern ('*be*'), '*ce*'), and '*de*').

Step 7: Pattern '*ae*'):4, and '*be*'):4 are available in the set of frequent itemsets in the original database as shown in Table 4, sogo to sub-step 7-1. Unlike patterns ('*ae*') and '*be*'), both ('*de*'):5 and '*ce*'):4 are not available in the set of frequent itemsets in the original database. Hence, do sub-step 7-2, i.e., put them into a set of *rescan_original* for mining frequent itemsets in the step 9 (case#3).

sub-step 7-1: The updated support count for pattern ('*ae*') is thus 3+ = 7, and the count for pattern '*be*') is 3 + 4 = 7. The counts of both patterns are also greater than the updated support threshold, hence they become frequent patterns. The algorithm collects ('*ae*'):7, and ('*be*'):7 into the *newFI*S.

sub-step 7-2: *rescan_original* = {('*d*'):5, ('*c*'):}.

Step 8: For case#2, the algorithm considers each item in the *frequentList* set, but there are no items in *frequentList*, hence there are no items that satisfy the condition for this case.

Step 9: For case#3, the algorithm considers each item in the *rescanOrigina*l set, that is, {('*d*'):5, ('*c*'):}. Item'' is considered first. In order to update the support count of patterns'*de*', its original support count must be obtained by rescanning and counting item in the *condOrgPat*h (['*b*','*a*','*d*']:2, and ['*b*','*a*']:1.). The support count of item '*d*' turns out to be 2. Hence, the updated support count for item '*d*' is 2 + 5 = 7, which is also greater than the updated support threshold.

Step 10: The algorithm generates pattern (*de*).

sub-step 10-1: Appends ('*de*'):5 to the *newFIS.* Item d is inserted into *fList*{'*a*':4, '*b*':4, '*d*':}. On the other hand, the updated support count for item '*c*', 0 + 4 = 4, which is less than 5; hence, the algorithm ignores this item.

Step 11: Then, the set of new 2-item-long frequent itemsets associated with item '*e*' is {('*ae*'): 7, ('*be*'): 7,('*de*'): 7}, and the sorted *fList* with frequent items and their support count in the incremental database is{'*d*':5, '*a*':4, '*b*':4}. The number of all items in *fList* is greater than 1; hence, the process for mining 3-item-long frequent pattern associated with item '*e*' is performed.

Step 12: The algorithm constructs an incremental conditional pattern tree, or ICP-tree from only the incremental database, or conditional pattern based from the incremental database (*condNewPath*): ['*b*','*a*','*c*','*d*']:3, ['*b*','*c*','*d*']:1, and ['*a*','*d*']:1, by utilizing the frequent items in the sorted *fList*:{'*d*':5, '*a*':4, '*b*':} as shown in Fig. 8 (a) to 8 (c).

Step 13: Next, the algorithm continues to discover frequent itemsets starting from item '*b*′ from the bottom of the *sub_headertable*: {'*b*':*4,* '*a*':*4,* '*d*':*5}* by recursively calling the incremental pattern growth algorithm in step 3-12 until the number of all items in *fList* is less than or equal to.

Step 14: Finally, when the iteration process of mining frequent pattern ending with item '*e*' is completed, all frequent itemsets associated with item '*e*' are as follows: ('*e*'):8, ('*ae*'):7, ('*be*'):7, ('*de*'):7, ('*ab*'):6, ('*db*'):6, ('*dabe*'):5, and ('*da*'):6.

The example, presented above, is an example of only (k = 2) item frequent itemsets mining, associated with item '*e*'. However, when frequent itemsets mining with '*e*' ended, all conditional pattern trees were also constructed from only the conditional pattern base of the incremental database, as shown in Fig. 9. The list of frequent itemsets, associated with each item, in the header table, is shown in Table 5.

This example shows that the number of constructed ICP-trees, as shown in Fig. 8 and Fig. 9, was reduced, compared to the pattern growth mining described in Section III: our improved technique constructed sub-trees, only from

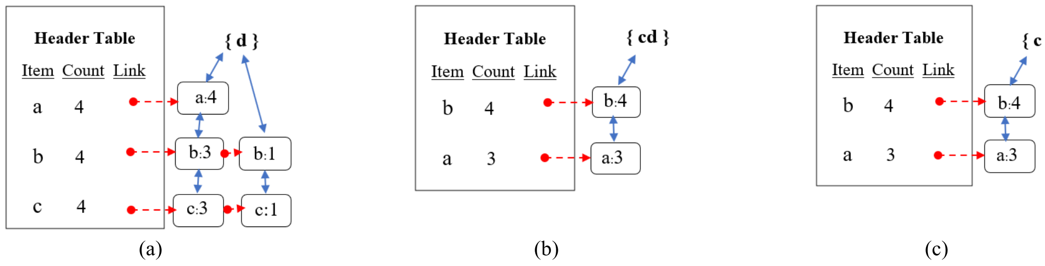

FIGURE 9. Incremental conditional pattern tree (ICP-tree) constructed for item 'd' and 'c.'

TABLE 5. Frequent itemsets mining by the proposed algorithm.

Item	Frequent itemset
'*e*'	('*e*'):8, ('*ae*'):7, ('*be*'):7, ('*de*'):7, ('*abe*'):6, ('*dbe*'):6, ('*dabe*'):5, ('*dae*'):6
'*d*'	('*d*'):10, ('*bd*'):9, ('*ad*'):9, ('*cd*'):6, ('*bcd*'):6, ('*acd*'):5, ('*bacd*'):5, ('*abd*'):8
'*c*'	('*c*'):9, ('*bc*'):9, ('*ac*'):6, ('*bac*'):6
'*a*'	('*a*'):12, ('*ba*'):10
'*b*'	('*b*'):13

the incremental database, and not from the whole updated database.

D. COMPLEXITY ANALYSIS

Regarding time complexity, the time complexity of our previously developed algorithm was $t(m_1, m_2, n) \in O\{(m_1 + m_2)(nlogn + n)\}$, where m_1 was the number of conditional pattern bases of the original path; m_2 was the number of conditional pattern bases of the incremental path; and n was the number of items in the header table. On the other hand, since our algorithm construct ICP-tree or sub-tree only from conditional pattern bases of the incremental database, its time complexity was $t(m_2, n) \in O\{(m_2)(n \log n + n)\}$, less than that of the previously algorithm.

Regarding space complexity, the space complexity of our previously proposed algorithm was $s(m_1, m_2) \in O(m_1 + m_2)$, and our proposed algorithm was $s(m_2) \in O(m_2)$, the value of the parameter m_2 (number of conditional pattern bases of the incremental path) in the complexity formula of our algorithm was smaller than those of our previously algorithm.

V. EXPERIMENTAL RESULTS AND DISCUSSION

We evaluated and compared the performances of our proposed algorithm with FP-Growth [11], FUFP-tree maintenance [27], Pre-FUFP [28], and FCFPIM [37] algorithms, in terms of execution time and number of generated sub-trees in the process of finding all frequent itemsets from the updated tree. All evaluated algorithms were coded in Python 3.60 and ran on Intel Xeon PC (2.93 GHz, 4 GB main memory, Microsoft Windows 10). A test dataset was generated by synthesis [1] with the following parameters: I = 10, represent the average maximum size of frequent items per transaction; T = 15, the average maximum length of transactions; and D = 100,' represented the number of transactions. The synthetic dataset was called I10T15D100K and contained 100,000 original transactions. Incremental databases appended to the original database had 3%, 10%, 20% and 30% of the size of the original database. The minimum support thresholds tested were 0.03, 0.05, and 0.1. In general, newly added transactions may or may not have the same patterns as those in the original database. Therefore, to evaluate the performance of the FIUFP-Growth algorithm fairly, we used two kinds of incremental databases: one with the same pattern as in the original database with 0% weight (see Tables 6 to 9) and another with 30% adjusting weight for increasing the difference between the patterns of newly added transactions. The reason for choosing an incremental database with 30% weight adjustment for testing was that this kind of weight was acceptable for incremental association rule mining. To put it as another disadvantage, when the weight too high, it is faster to rescan the whole database. Strong results are such as the following. Our algorithm was still effective, adequately fast, even when the patterns in the new database differed by more than 30% from those in the original database. In addition, the number of generated sub-trees in frequent itemset finding step was less than those generated by the other tested algorithms.

Three different types of execution time were recorded: time spent to update the tree structure, time taken to find all frequent itemsets from the updated tree, and the total time spent processing the incremental database. The results show that our algorithms executed faster than all the other four algorithms because our algorithm performed the incremental association rule mining problem by both updating tree structure and finding frequent itemset from the updated tree. All of the other four algorithms used traditional FP-growth to find all frequent itemsets from the updated tree, but our algorithm found all frequent itemsets by retrieving frequent itemsets directly from the original database without any rescanning. This kind of retrieval reduced the time to scan the conditional pattern bases of the original database. Moreover, the time to generate and scan the ICP-trees in our algorithm was also less than the time to generate sub-trees by the other

TABLE 6. Experimental results with incremental data of 30,000 transactions having the same statistical characteristics as those in the original database.

Algorithm	Minimum support	Number of sub-trees	Execution time for modified FP-tree (s)	Execution time for pattern growth mining (s.)	Execution time (s.)
FP-Growth	3%	6393	1467.4849	17.1004	1484.5853
FUFP-tree		6393	582.6920	17.4675	600.1596
Pre-FUFP		6393	583.8167	17.3503	600.1670
FCFPIM		6393	616.2953	17.7487	633.8783
FIUFP-Growth		**1927**	**580.8402**	**10.6399**	**591.4801**
FP-Growth	5%	4218	1352.1719	9.7728	1361.9447
FUFP-tree		4218	533.6483	9.9213	543.5695
Pre-FUFP		4218	533.3829	9.7571	543.1399
FCFPIM		4218	621.2854	10.1555	631.4409
FIUFP-Growth		**1284**	**531.7657**	**6.6871**	**538.4528**
FP-Growth	10%	135	760.7754	3.8591	764.6345
FUFP-tree		135	295.8533	3.8669	299.7202
Pre-FUFP		135	295.1034	3.9294	299.0328
FCFPIM		135	621.2697	4.0211	625.2819
FIUFP-Growth		**14**	**293.5253**	**2.8748**	**296.4002**

TABLE 7. Experimental results with incremental data of 20,000 transactions having the same statistical characteristics as those in the original database.

Algorithm	Minimum support	Number of sub-trees	Execution time for modified FP-tree (s)	Execution time for pattern growth mining (s.)	Execution time (s.)
FP-Growth	3%	6429	1245.1403	17.3972	1262.5375
FUFP-tree		6429	380.8471	16.6160	397.4631
Pre-FUFP		6429	377.8473	16.6082	394.4555
FCFPIM		6429	412.0218	16.7957	428.8175
FIUFP-Growth		**1938**	**377.9098**	**10.6555**	**388.5653**
FP-Growth	5%	4219	1154.0216	9.0462	1163.0679
FUFP-tree		4219	347.0215	9.3196	356.3411
Pre-FUFP		4219	347.1387	9.4681	356.6068
FCFPIM		4220	411.5280	9.2650	420.7930
FIUFP-Growth		**1284**	**346.3418**	**6.2261**	**352.5679**
FP-Growth	10%	137	652.9625	3.6013	656.5639
FUFP-tree		137	194.4467	3.6716	198.1183
Pre-FUFP		137	195.4857	3.6950	199.1807
FCFPIM		146	408.3781	3.7184	412.0966
FIUFP-Growth		**15**	**194.9154**	**2.6482**	**197.5637**

algorithms because the ICP-trees were constructed from a smaller incremental database.

Regarding execution time for updating tree structure, the results show clearly that FP-Growth took the longest time to update because FP-growth had to rebuild the tree from a whole updated database. The FCFPIM algorithm took more time to update the tree than our algorithm did and took even more time when the number of minimum supports was large, most probably due to the small number of frequent itemsets and the substantial number of infrequent itemsets. In general, FUFP, Pre-FUFP, and our algorithm performed similarly well because they were based on the same FP-Tree principles. Table 6–13 shows that our algorithm generated fewer sub-trees than FP-Growth, FUFP-tree, Pre-FUFP, and FCFPIM did. Unlike the other four algorithms that used the original FP-growth to find all frequent itemsets from a fully updated tree, our algorithm found frequent itemsets from conditional pattern bases without constructing a conditional pattern tree. If necessary, our algorithm would construct an ICP-tree for finding the next k-frequent itemsets. Furthermore, the execution time for finding frequent itemsets from the ICP-tree was shorter than those required by the other four algorithms employing a conventional tree because the frequent itemsets in the original database could already be retrieved directly without any rescanning, eliminating the need to scan the conditional pattern base of the original

TABLE 8. **Experimental results with incremental data of 10,000 transactions having the same statistical characteristics as those in the original database.**

Algorithm	Minimum support	Number of sub-trees	Execution time for modified FP-tree (s)	Execution time for pattern growth mining (s.)	Execution time (s.)
FP-Growth	3%	6429	1101.0096	15.1474	1116.1570
FUFP-tree		6429	178.8619	15.0770	193.9389
Pre-FUFP		6429	181.2055	15.1865	196.3919
FCFPIM		6406	191.0958	15.4052	206.5010
FIUFP-Growth		**1938**	**178.6041**	**8.8744**	**187.4785**
FP-Growth	5%	4219	1008.9848	8.4291	1017.4139
FUFP-tree		4219	163.9333	8.4369	172.3702
Pre-FUFP		4219	164.2302	8.4135	172.6436
FCFPIM		4219	190.7525	8.4837	202.4548
FIUFP-Growth		**1284**	**164.1208**	**5.4606**	**169.5813**
FP-Growth	10%	137	559.3206	3.3435	562.6641
FUFP-tree		137	92.5713	3.4764	96.0477
Pre-FUFP		137	93.5400	3.5701	97.1101
FCFPIM		135	190.7525	3.4685	194.2210
FIUFP-Growth		**15**	**92.5635**	**2.5232**	**95.0868**

TABLE 9. **Experimental results with incremental data of 3,000 transactions having the same statistical characteristics as those in the original database.**

Algorithm	Minimum support	Number of sub-trees	Execution time for modified FP-tree (s)	Execution time for pattern growth mining (s.)	Execution time (s.)
FP-Growth	3%	6425	971.0499	14.5849	985.6348
FUFP-tree		6425	52.8399	14.4677	67.3076
Pre-FUFP		6425	51.4884	14.5458	66.0342
FCFPIM		6425	55.4805	14.5458	70.0263
FIUFP-Growth		**1917**	**52.4415**	**7.7338**	**60.1752**
FP-Growth	5%	4219	894.6802	7.8979	902.5781
FUFP-tree		4219	47.4731	7.9135	55.3866
Pre-FUFP		4219	48.1449	7.9448	56.0896
FCFPIM		4219	55.0492	8.0775	63.1268
FIUFP-Growth		**1284**	**49.1604**	**4.9372**	**54.0976**
FP-Growth	10%	137	508.8007	3.1560	511.9567
FUFP-tree		137	26.9355	3.1560	30.0915
Pre-FUFP		137	27.0371	3.1482	30.1853
FCFPIM		137	54.0462	3.1841	57.2303
FIUFP-Growth		**15**	**26.9668**	**2.2108**	**29.1775**

database. Notice, also, that our algorithm was faster at pattern growth mining from the updated tree than the other four algorithms.

The time required for updating tree and mining frequent itemsets by our algorithm at 3% minimum support threshold was only ~88 % of that required by FP-Growth, ~11 % of that required by FUFP-tree, ~9% of that required by Pre-FUFP, ~9 %, and ~14% of that required by FCFPIM. Moreover, the execution time for mining frequent itemsets from the updated tree required by our algorithm was only ~46 % of the average execution time required by FP-Growth, FUFP-tree, Pre-FUFP, and FCFPIM. These results are consistent with the complexity analysis, explained in Section IV. According to the complexity analysis, the time complexity of our algorithm was lower than those of the compared algorithms from not having an m_1 parameter—the number of conditional pattern bases of the original path—in its complexity equation. Moreover, our algorithm was still effective even when the minimum support threshold was very low and when the frequent itemset patterns in the incremental database differed considerably from the patterns in the original database.

Regarding space complexity, from the results, shown in table 6-13, it is clear that the number of sub-trees generated by our algorithm was fewer than those generated by the other four algorithms. According to the complexity analysis (Section IV), the space complexity of our algorithm was lower than those of the compared algorithms because it did

TABLE 10. Experimental results with incremental data of 30,000 transactions having 30% difference in statistical characteristics from those in the original database.

Algorithm	Minimum support	Number of sub-trees	Execution time for modified FP-tree (s)	Execution time for pattern growth mining (s.)	Execution time (s.)
FP-Growth	3%	6416	1497.373	17.8049	1515.1779
FUFP-tree		6416	565.0763	17.4441	582.5205
Pre-FUFP		6416	575.5087	17.5859	593.0946
FCFPIM		6416	569.7536	17.41004	587.1637
FIUFP-Growth		**1937**	**564.7480**	**10.5070**	**575.2550**
FP-Growth	5%	4218	1322.1640	9.6789	1331.8425
FUFP-tree		4218	514.6269	9.6321	524.2590
Pre-FUFP		4218	516.5638	9.7255	526.2892
FCFPIM		4218	511.3982	9.6282	521.0263
FIUFP-Growth		**1284**	**512.0562**	**6.6480**	**518.7042**
FP-Growth	10%	132	706.7766	3.8435	710.6201
FUFP-tree		132	281.4872	3.7263	285.2135
Pre-FUFP		132	280.2373	3.8279	284.0651
FCFPIM		132	657.8687	3.9059	661.7747
FIUFP-Growth		**13**	**280.1281**	**2.7888**	**282.9169**

TABLE 11. Experimental results with incremental data of 20,000 transactions having 30% difference in statistical characteristics from those in the original database.

Algorithm	Minimum support	Number of sub-trees	Execution time for modified FP-tree (s)	Execution time for pattern growth mining (s.)	Execution time (s.)
FP-Growth	3%	6198	1298.1351	16.3269	1314.4620
FUFP-tree		6198	371.0275	16.4363	387.4638
Pre-FUFP		6198	370.3333	16.3661	386.6994
FCFPIM		6198	419.6389	16.7838	436.4228
FIUFP-Growth		**1872**	**370.3481**	**7.2416**	**377.5897**
FP-Growth	5%	4215	1191.6394	9.1741	1200.8135
FUFP-tree		4215	332.8350	9.3743	342.2093
Pre-FUFP		4215	332.8666	9.3822	342.2487
FCFPIM		4215	417.2838	9.4837	426.7675
FIUFP-Growth		**1283**	**329.2340**	**6.1167**	**335.3508**
FP-Growth	10%	135	684.3487	3.7189	688.0676
FUFP-tree		135	185.4630	3.5466	189.0096
Pre-FUFP		135	184.1820	3.5856	187.7677
FCFPIM		135	409.4874	3.8747	413.3622
FIUFP-Growth		**14**	**184.5335**	**2.5702**	**187.1037**

not have an m_1 parameter in its complexity equation. The space complexity of our algorithm was lower than those of the other four algorithms. Procedurally, the other four algorithms used FP-Growth for finding frequent itemsets, and in each iteration of the frequent itemset finding process, created a sub-tree. In contrast, our algorithm did not create a sub-tree in each iteration of the frequent itemset finding process (see step-11 pseudocode of our algorithm in Section IV). Moreover, our proposed ICP-tree was constructed only from the conditional pattern bases of the incremental database, while the generated sub-trees by the other four algorithms were constructed from the conditional pattern bases of the whole updated database, original plus incremental, hence the space required to generate all ICP-trees were smaller than the space required to generate all sub-trees by the other four algorithms. To summarize, the other four algorithms generated a larger number of sub-trees than our algorithm did, hence, in principle, they would require more execution time and storage space to generate them than our algorithm would.

Experimentally, the outcomes of evaluation runs of all tested algorithms, in terms of execution time and storage space needed for all generated trees, indicate that our algorithm was more efficient because it executed successful runs in shorter time and generated fewer trees to be stored than the other four algorithms.

TABLE 12. Experimental results with incremental data of 10,000 transactions having 30% difference in statistical characteristics from those in the original database.

Algorithm	Minimum support	Number of sub-trees	Execution time for modified FP-tree (s)	Execution time for pattern growth mining (s.)	Execution time (s.)
FP-Growth	3%	6387	1095.4737	15.1004	1110.5742
FUFP-tree		6387	179.2447	15.2177	194.4623
Pre-FUFP		6387	179.3384	15.1630	194.5014
FCFPIM		6387	200.6580	15.6551	216.3132
FIUFP-Growth		**1927**	**177.1823**	**7.7963**	**184.9786**
FP-Growth	5%	4221	1010.2240	8.3509	1018.5749
FUFP-tree		4221	157.8009	8.3979	166.1988
Pre-FUFP		4221	157.0979	8.3822	165.4801
FCFPIM		4221	2007674	8.6400	209.4074
FIUFP-Growth		**1285**	**156.7698**	**5.1325**	**161.9022**
FP-Growth	10%	135	579.8718	3.2966	583.1684
FUFP-tree		135	88.7982	3.4607	92.2589
Pre-FUFP		135	87.9701	3.4764	91.4464
FCFPIM		135	195.4553	3.4990	198.9550
FIUFP-Growth		**14**	**88.1107**	**2.4217**	**90.5324**

TABLE 13. Experimental results with incremental data of 3,000 transactions having 30% difference in statistical characteristics from those in the original database.

Algorithm	Minimum support	Number of sub-trees	Execution time for modified FP-tree (s)	Execution time for pattern growth mining (s.)	Execution time (s.)
FP-Growth	3%	6367	981.6775	14.6161	996.2936
FUFP-tree		6367	51.3556	14.6631	66.0186
Pre-FUFP		6367	50.7618	14.3115	65.0733
FCFPIM		6367	56.4804	14.4990	70.9794
FIUFP-Growth		**1904**	**49.6370**	**7.8510**	**57.4879**
FP-Growth	5%	4221	902.7050	8.0351	910.7401
FUFP-tree		4221	47.2231	7.8901	55.1131
Pre-FUFP		4221	47.4105	7.9136	55.3241
FCFPIM		4221	54.6993	7.9994	62.6988
FIUFP-Growth		**1285**	**46.0903**	**4.7028**	**50.7931**
FP-Growth	10%	137	516.1131	3.3201	519.4332
FUFP-tree		137	28.1522	3.3436	31.4958
Pre-FUFP		137	27.3622	3.2967	30.6589
FCFPIM		137	54.6681	3.3435	58.0016
FIUFP-Growth		**15**	**26.3262**	**2.1874**	**28.5135**

VI. CONCLUSION

We improved the efficiency of incremental frequent pattern mining with our fast incremental updating frequent pattern growth algorithm (FIUFP) and designed a new incremental conditional pattern tree (ICP-tree) suitable for incremental mining of frequent itemsets. The basic idea was to use frequent itemsets, that were already mined from the original database and their support counts, to support the frequent itemset mining from the updated FPISC-tree. Our new algorithm and tree was faster and used less space, because it constructed a smaller number of sub-trees than FP-Growth, FUFP-tree maintenance, Pre-FUFP and FCFPIM. From the results, at 3% minimum support threshold, the average execution time for pattern growth mining of our algorithm performs ~46% faster than FP-Growth, FUFP-tree, Pre-FUFP, and FCFPIM, which all performed similarly. In term of overall execution time, our algorithm also performs faster than FP-Growth (~88 %), FUFP-tree (~11 %), and Pre-FUFP (~9 %). In addition, the execution time required for updating tree and mining frequent itemsets with our algorithm at 3% minimum support threshold was only ~88 % of that required by any of the FP-Growth, ~11 % FUFP-tree, ~9 % Pre-FUFP, and ~14 % FCFPIM algorithms. Moreover, when the minimum support threshold was very low or the characteristic pattern of the incremental database differed considerably from the original database, FIUFP-Growth algorithm was still very efficient.

ACKNOWLEDGMENT

The authors would like to thank the Faculty of Information Technology, King Mongkut's Institute of Technology Ladkrabang for facility support.

REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *ACM SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, Jun. 1993.

[2] Y. Djenouri, A. Belhadi, P. Fournier-Viger, and H. Fujita, "Mining diversified association rules in big datasets: A cluster/GPU/genetic approach," *Inf. Sci.*, vol. 459, pp. 117–134, Aug. 2018.

[3] N. Sael, A. Marzak, and H. Behja, "Multilevel clustering and association rule mining for learners' profiles analysis," *Int. J. Comput. Sci. Issues*, vol. 10, no. 3, pp. 188–194, 2013.

[4] P.-S. Chien, Y.-F. Tseng, Y.-C. Hsu, Y.-K. Lai, and S.-F. Weng, "Frequency and pattern of chinese herbal medicine prescriptions for urticaria in Taiwan during 2009: Analysis of the national health insurance database," *BMC Complementary Alternative Med.*, vol. 13, no. 1, p. 209, Aug. 2013.

[5] L. Yao, Y. Zhang, B. Wei, W. Wang, Y. Zhang, X. Ren, and Y. Bian, "Discovering treatment pattern in traditional Chinese medicine clinical cases by exploiting supervised topic model and domain knowledge," *J. Biomed. Informat.*, vol. 58, pp. 260–267, Dec. 2015.

[6] H. Gao, W. Huang, and X. Yang, "Applying probabilistic model checking to path planning in an intelligent transportation system using mobility trajectories and their statistical data," *Intell. Automat. Soft Comput.*, vol. 25, no. 3, pp. 547–559, Jan. 2019.

[7] W. Altaf, M. Shahbaz, and A. Guergachi, "Applications of association rule mining in health informatics: A survey," *Artif. Intell. Rev.*, vol. 47, no. 3, pp. 313–340, Mar. 2017.

[8] P. N. Tan, M. Steinbach, A. Karpatne, and V. Kumar, *Introduction to Data Mining*, 2nd ed. London, U.K.: Pearson, 2018.

[9] J. Han, J. Pei, and M. Kamber, *Data Mining: Concepts and Techniques*. Amsterdam, The Netherlands: Elsevier, 2011.

[10] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. VLDB*, vol. 1215, Sep. 1994, pp. 487–499.

[11] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining frequent patterns without candidate generation: A frequent-pattern tree approach," *Data Mining Knowl. Discovery*, vol. 8, no. 1, pp. 53–87, Jan. 2004.

[12] X. Liu, K. Zhai, and W. Pedrycz, "An improved association rules mining method," *Expert Syst. Appl.*, vol. 39, no. 1, pp. 1362–1374, Jan. 2012.

[13] M. S. Hoseini, M. N. Shahraki, and B. S. Neysiani, "A new algorithm for mining frequent patterns in can tree," in *Proc. 2nd Int. Conf. Knowl.-Based Eng. Innov. (KBEI)*, Tehran, Iran, Nov. 2015, pp. 843–846.

[14] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad, "A tree projection algorithm for generation of frequent item sets," *J. Parallel Distrib. Comput.*, vol. 61, no. 3, pp. 350–371, Mar. 2001.

[15] Z.-H. Deng and S.-L. Lv, "PrePost+: An efficient N-lists-based algorithm for mining frequent itemsets via children–parent equivalence pruning," *Expert Syst. Appl.*, vol. 42, no. 13, pp. 5424–5432, Aug. 2015.

[16] Z.-H. Deng and S.-L. Lv, "Fast mining frequent itemsets using nodesets," *Expert Syst. Appl.*, vol. 41, no. 10, pp. 4505–4512, 2014.

[17] M. J. Zaki, "Scalable algorithms for association mining," *IEEE Trans. Knowl. Data Eng.*, vol. 12, no. 3, pp. 372–390, Jun. 2000.

[18] S. Bhanderi, N. C. Chauhan, and S. D. Bhanderi, "Incremental mining of association rules: A survey," *Int. J. Comput. Sci. Inf. Technol.*, vol. 3, no. 3, pp. 4071–4074, 2013.

[19] D. W. Cheung, J. Han, V. T. Ng, and C. Y. Wong, "Maintenance of discovered association rules in large databases: An incremental updating technique," in *Proc. 12th Int. Conf. Data Eng.*, New Orleans, LA, USA, Feb. 1996, pp. 106–114.

[20] R. Feldman, Y. Aumann, O. Lipshtat, and H. Manilla, "Borders: An efficient algorithm for association generation in dynamic databases," *J. Intell. Inf. Syst.*, vol. 12, pp. 61–73, Apr. 1999.

[21] T.-P. Hong, C.-Y. Wang, and Y.-H. Tao, "A new incremental data mining algorithm using pre-large itemsets," *Intell. Data Anal.*, vol. 5, no. 2, pp. 111–129, Mar. 2001.

[22] R. Amornchewin and W. Kreesuradej, "Probability-based incremental association rule discovery algorithm," in *Proc. Int. Symp. Comput. Sci. Appl.*, Hobart, TAS, Australia, Oct. 2008, pp. 212–215.

[23] R. Amornchewin and W. Kreesuradej, "Mining dynamic database using probability-based incremental association rule discovery algorithm," *J. Universal Comput. Sci.*, vol. 15, pp. 2409–2428, Jun. 2009.

[24] A. Ariya and W. Kreesuradej, "Probability-based incremental association rule discovery using the normal approximation," in *Proc. IEEE 14th Int. Conf. Inf. Reuse Integr. (IRI)*, San Francisco, CA, USA, Aug. 2013, pp. 432–439.

[25] A. Ariya and W. Kreesuradej, "An enhanced incremental association rule discovery with a lower minimum support," *Artif. Life Robot.*, vol. 21, no. 4, pp. 466–477, Dec. 2016.

[26] J.-L. Koh and S.-F. Shieh, "An efficient approach for maintaining association rules based on adjusting FP-tree structures," in *Proc. DASFAA*, Jeju, South Korea, 2004, pp. 417–424.

[27] T.-P. Hong, C.-W. Lin, and Y.-L. Wu, "Incrementally fast updated frequent pattern trees," *Expert Syst. Appl.*, vol. 34, no. 4, pp. 2424–2435, May 2008.

[28] T.-P. Hong, C.-W. Lin, and Y.-L. Wu, "Maintenance of fast updated frequent pattern trees for record deletion," *Comput. Statist. Data Anal.*, vol. 53, no. 7, pp. 2485–2499, May 2009.

[29] C.-W. Lin, T.-P. Hong, and W.-H. Lu, "The pre-FUFP algorithm for incremental mining," *Expert Syst. Appl.*, vol. 36, no. 5, pp. 9498–9505, Jul. 2009.

[30] C.-W. Lin and T.-P. Hong, "Maintenance of prelarge trees for data mining with modified records," *Inf. Sci.*, vol. 278, pp. 88–103, Sep. 2014.

[31] W.-G. Teng and M.-S. Chen, "Incremental mining on association rules," in *Foundations and Advances in Data Mining*, vol. 180. Berlin, Germany: Springer, 2005, pp. 125–162.

[32] W. Kreesuradej and W. Thurachon, "Discovery of incremental association rules based on a new FP-growth algorithm," in *Proc. IEEE 4th Int. Conf. Comput. Commun. Syst. (ICCCS)*, Singapore, Feb. 2019, pp. 184–188.

[33] M. Karanjikar and S. V. Kedar, "Secure association rule mining using Bi-Eclat algorithm on vertically partitioned databases," in *Proc. Int. Conf. Intell. Sustain. Syst. (ICISS)*, Palladam, India, Dec. 2017, pp. 176–181.

[34] L. Li, R. Lu, K.-K.-R. Choo, A. Datta, and J. Shao, "Privacy-preserving-outsourced association rule mining on vertically partitioned databases," *IEEE Trans. Inf. Forensics Security*, vol. 11, no. 8, pp. 1847–1861, Aug. 2016.

[35] P. D. Lambhate and R. Khairnar, "AssociationRule on vertically partitioned data," in *Proc. Int. Conf. Comput., Commun., Control Automat. (ICCUBEA)*, Pune, India, Aug. 2017, pp. 1–5.

[36] Y. Djenouri, J. C.-W. Lin, K. Norvag, and H. Ramampiaro, "Highly efficient pattern mining based on transaction decomposition," in *Proc. IEEE 35th Int. Conf. Data Eng. (ICDE)*, Macao, China, Apr. 2019, pp. 1646–1649.

[37] J. Sun, Y. Xun, J. Zhang, and J. Li, "Incremental frequent itemsets mining with FCFP tree," *IEEE Access*, vol. 7, pp. 136511–136524, 2019.

[38] J. C.-W. Lin, Y. Shao, P. Fournier-Viger, Y. Djenouri, and X. Guo, "Maintenance algorithm for high average-utility itemsets with transaction deletion," *Int. J. Speech Technol.*, vol. 48, no. 10, pp. 3691–3706, Apr. 2018.

[39] P. Thusaranon and W. Kreesuradej, "Frequent itemsets mining using random walks for record insertion and deletion," in *Proc. 8th Int. Conf. Inf. Technol. Electr. Eng. (ICITEE)*, Yogyakarta, Indonesia, Oct. 2016, pp. 1–6.

[40] P. Thusaranon and W. Kreesuradej, "A probability-based incremental association rule discovery algorithm for record insertion and deletion," *Artif. Life Robot.*, vol. 20, no. 2, pp. 115–123, Jun. 2015.

WANNASIRI THURACHON received the B.Sc. degree in computer science from Lampang Rajabhat University, Thailand, in 1996, and the M.Sc. degree in information technology from Naresuan University, Thailand, in 2004. She is currently pursuing the Ph.D. degree with KMITL. Her main research interests include data mining, association rule mining, incremental association rule mining, and data mining.

WORAPOJ KREESURADEJ received the B.Eng. degree (Hons.) in electronics engineering from the King Mongkut's Institute of Technology Ladkrabang, Thailand, in 1989, and the M.S.E.E. and Ph.D. degrees in electrical engineering from Texas Tech University, Lubbock, T'X, USA, in 1993 and 1996, respectively.

He is currently an Associate Professor with the Faculty of Information Technology, King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand. His main research interests include data mining fuzzy system neural networks, business intelligence, and big data analytics.