Received September 12, 2020, accepted September 30, 2020, date of publication October 7, 2020, date of current version October 19, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3029302

Efficient Mining of Frequent Itemsets Using Only One Dynamic Prefix Tree

JUN-FENG QU[1], BO HANG[1], (Member, IEEE), ZHAO WU[1], ZHONGBO WU[1], QIONG GU[1], AND BO TANG[2]
[1]School of Computer Engineering, Hubei University of Arts and Science, Xiangyang 441053, China
[2]School of Mathematics and Statistics, Hubei University of Arts and Science, Xiangyang 441053, China

Corresponding author: Bo Tang (tangbo0809@163.com)

This work was supported in part by the Science and Technology Development Program of Central Guide to Local Government of China under Grant 2019ZYYD043, in part by the International Science and Technology Cooperation Program of Hubei Province under Grant 2019AHB059, in part by the Key Agricultural Programs of Science and Technology and Innovation of Xiangyang, and in part by the Talent Introduction Project of Hubei University of Arts and Science.

ABSTRACT Frequent itemset mining is a fundamental problem in data mining area because frequent itemsets have been extensively used in reasoning, classifying, clustering, and so on. To mine frequent itemsets, previous algorithms based on a prefix tree structure have to construct many prefix trees, which is very time-consuming. In this paper, we propose a novel frequent itemset mining algorithm called DPT (Dynamic Prefix Tree) which uses only one prefix tree. We first introduce the concept of the post-conditional database of an itemset, and analyze the distribution of an itemset's post-conditional database in a prefix tree representing a database. Subsequently, we illuminate how DPT adjusts the prefix tree to mine frequent itemsets and give three optimization techniques. An interesting advantage of DPT is that the algorithm can directly output a prefix tree representing all frequent itemsets after slight modifications. Using only one dynamic prefix tree, DPT avoids the high cost of constructing many prefix trees and thus gains significant performance improvement. Experimental results show that DPT remarkably outperforms previous algorithms with respect to running time and memory usage, and that a prefix tree representing all frequent itemsets DPT outputs can be used more efficient than a list representing them previous algorithms output.

INDEX TERMS Frequent itemset, post-conditional database, dynamic prefix tree.

I. INTRODUCTION

Frequent itemsets are useful and important information from databases and can be used in many data mining tasks such as association rules mining [1], clustering [2], classification [3], [4], prediction [5], and so on. However, frequent itemsets cannot be easily identified from a database because there are 2^n candidates for a database with n items. Mining frequent itemsets from a database has been a fundamental and crucial research topic in data mining area [8].

The problem of frequent itemsets mining can be described as follows. Let $I = \{i_1, i_2, i_3, \cdots, i_n\}$ be a set of distinct items and DB (Database) be a transaction database, where each transaction is a subset of I. An itemset is a subset of I, and it is called k-itemset if it contains k distinct items. If all items of an itemset are contained in a transaction, this transaction satisfies the itemset. In a database, the number of transactions satisfying an itemset is the support of this itemset.

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

Given a database and a minimal support threshold (denoted as *min_sup*), an itemset is frequent if its support is no smaller than *min_sup*. The task of frequent itemsets mining is to find out all frequent itemsets with their supports. The task is intractable, because there is an enormous number of frequent itemsets in a database, especially for dense databases or small *min_sups*.

The Apriori algorithm [9] is an effective method for frequent itemset mining. In order to obtain all frequent itemsets, Apriori iteratively scans a database. It can find out all frequent k-itemsets ($k = 1, 2, 3, \ldots$) after the kth database scan. Generating and testing a large number of candidate itemsets, Apriori and the Apriori-like algorithms are called candidate generation-and-test approaches. Although these approaches can mine all frequent itemsets from a database, they are unavoidably confronted with two problems: (1) the database is scanned many times, and the time of database scan is exactly equal to the length of the longest frequent itemset; (2) the number of candidate itemsets is very large, and generating and testing these candidates is very costly, especially when the database or all candidate itemsets cannot be completely loaded in main memory.

Pattern growth approaches such as FP-Growth [10] can avoid the above problems and mine frequent itemsets from a database. In FP-Growth, a database is represented as a highly compact prefix tree structure named as FP-tree. After constructing an initial FP-tree from a database, FP-Growth recursively constructs conditional FP-trees to mine frequent itemsets.

A. MOTIVATION

Many previous works [10], [26]–[31], [33]–[35] have shown that pattern growth approaches based on prefix trees are more efficient than candidate generation-and-test approaches. However, pattern growth approaches are confronted with a problem that they have to spend much time in constructing many (conditional) prefix trees. The construction of a conditional prefix tree involves several steps: identifying frequent items, constructing a branch for each transaction, and inserting the branch into the tree. FP-Growth-like algorithms iteratively and recursively execute these steps, which is very time-consuming. To obtain a frequent itemset containing k items, these algorithms have to construct k conditional prefix trees. Frequently constructing prefix trees spends much time and many prefix trees also lead to poor data locality.

From the application perspective [4]–[7], frequent itemsets usually play an important role as an intermediate product. For example, some applications need to judge whether an itemset is frequent or not or fetch the support of a frequent itemset. Factually, there are generally a large number of frequent itemsets especially for dense databases or small *min_sups*. However, previous algorithms simply output a list of frequent itemsets, no matter what candidate generation-and-test approaches or pattern growth approaches these algorithms belong to. This list is really too large to be efficiently used. Before frequent itemsets are further used, it is important to use an appropriate form of representing them.

B. CONTRIBUTIONS

This work focuses on the efficient mining of frequent itemsets and the convenient usage of them. The main contributions of the work are as follows.

(1) We propose a novel pattern growth algorithm for mining frequent itemsets. Different from previous pattern growth algorithms constructing many prefix trees, our algorithm needs only one prefix tree for mining all frequent itemsets with their supports. Frequently constructing prefix trees spends much time and operating on many prefix trees also leads to poor data locality, which is avoided by our algorithm using only one prefix tree. After constructing an initial prefix tree from a database, our algorithm repeatedly adjusts the tree to mine frequent itemsets. Thus, the algorithm is named as the dynamic prefix tree algorithm (abbreviated as DPT).

(2) Another interesting advantage of DPT is that it can directly output a prefix tree representing all frequent itemsets, while previous algorithms always output a frequent itemset list. Compared with a list representing all frequent itemsets, a prefix tree representing them is compact and can be used more efficiently and easily. For example, when an application judges whether an itemset is frequent or not or fetches the support of a frequent itemset, operations on a prefix tree are more efficient than those on a list, especially for a large number of frequent itemsets.

(3) We have done extensive experiments. Experimental results show the performance improvement of DPT over previous algorithms and the advantage of a prefix tree representing frequent itemsets over a frequent itemset list.

The remainder of the paper is organized as follows. Section 2 reviews two kinds of search space for the frequent itemset mining problem and the prefix tree structure, and related work is also discussed in this section. Section 3 introduces the post-conditional database of an itemset which is a fundamental concept used in DPT, and there are several lemmas on which DPT is based. In Section 4, we describe the main frame of DPT, give three optimization techniques used in DPT, and also illuminate how DPT directly outputs a prefix tree representing all mined frequent itemsets. Experimental results are given in Section 5, and the paper ends in the conclusion of Section 6.

II. BACKGROUND

A. SEARCH SPACE

Mining all frequent itemsets with their supports is an intractable problem because the search space is huge. Suppose $|I| = n$, a mining algorithm has to check 2^n itemsets, i.e., there are 2^n nodes in the search space. If n is small enough, it is possible to directly count the supports of all itemsets by only one database scan. Bigger n is, more important it is to organize itemsets in an appropriate search space. All itemsets can be organized in two kinds of search space: a lattice structure and a set-enumeration tree. Each node in the search space represents a subset (i.e. an itemset) of I.

1) LATTICE STRUCTURE

Consider as an example that there is a set of items $I = \{a, b, c, d\}$, and then $2^{|I|} = 16$ subsets can form a lattice structure as depicted in Figure 1(a). Given a *min_sup*, each itemset corresponding to a node in the lattice structure may be frequent, which means that an algorithm has to check all nodes.

The Apriori algorithm [9] searches the lattice structure for frequent itemsets in a bottom-to-top way. After obtaining the set of frequent 1-itemsets by scanning a database, Apriori iteratively generates the set of candidate k-itemsets ($k = 2, 3 \ldots$) from the set of frequent ($k - 1$)-itemsets. Subsequently, Apriori identifies frequent k-itemsets from candidate k-itemsets by scanning the database again. This process progresses until either all candidate itemsets are not frequent or no new candidate k-itemset can be generated from the set of frequent (k-1)-itemsets. Apriori represents a priori which means all subsets of a frequent itemset are frequent and all supersets of an infrequent itemset are infrequent. For example, in Figure 1(a), if itemset $\{ab\}$ is infrequent, itemsets

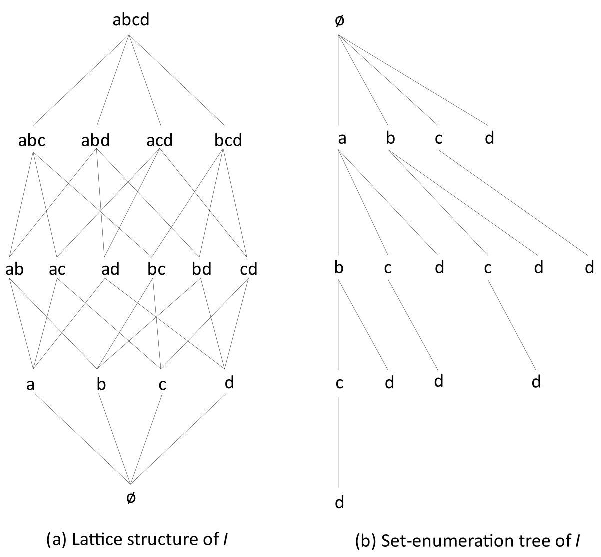

FIGURE 1. Search space.

$\{abc\}$, $\{abd\}$, and $\{abcd\}$ must be infrequent. In the process of the bottom-to-top search, Apriori successively checks 1-itemsets, 2-itemsets, 3-itemsets, ..., and the qualification as a candidate k-itemset is that all of its subsets containing (k-1) items must be frequent. In this way, Apriori can prune the search space well.

2) SET-ENUMERATION TREE

For $I = \{a, b, c, d\}$, a set-enumeration tree [23] representing all itemsets is depicted in Figure 1(b), and it is another kind of search space for frequent itemset mining. Each node in the tree represents an itemset composed of all items from the node to the root.

Different from the lattice structure in which there is no order among items, all items in a set-enumeration tree are sorted. The item order in a set-enumeration tree can be their lexicographic order, their frequency-ascending/descending order, and so on. The tree in Figure 1(b) uses the lexicographic order, i.e., $a < b < c < d$.

Note that all child nodes of each node in a set-enumeration tree are sorted according to the item order. Mining algorithms can search a set-enumeration tree for frequent itemsets in depth-first order or in breadth-first order. In [23], Agarwal *et al.* proposed tree projection algorithms based on a set-enumeration tree. Their algorithms project all transactions containing an itemset to a node representing the itemset, and these transactions are stored in a matric related to the node. Frequent itemsets with their supports can be mined from these matrices. Based on a set-enumeration tree, depth-first mining algorithms and breadth-first mining algorithms are studied and compared in [23].

B. DATE STRUCTURE

Frequent itemset mining is based on databases stored on disk. Apriori and Apriori-like algorithms have to scan a database on disk many times for mining frequent itemsets. FP-Growth and FP-Growth-like algorithms [10], [25]–[29] use a prefix tree structure to mine frequent itemsets. After an initial prefix tree is constructed from a mined database, the task of mining frequent itemsets based on prefix trees takes place in memory. A prefix tree looks like a set-enumeration tree in Figure 1(b) with respect to structure. Each node in a prefix tree represents an itemset composed of all items from the node to the root. Besides an item, each node in a prefix tree also contains a counter that records the partial support of the itemset. All items are sorted in a prefix tree as they do in a set-enumeration tree. Given a database, a *min_sup*, and an item order, Algorithm 1 can construct a prefix tree.

Algorithm 1 Construct a Prefix Tree

```
Input:
    DB a database
    min_sup the minimum support threshold
Output:
    T a prefix tree representing DB
1: scan DB and obtain all frequent items
2: generate a null node as the root of T
3: for each transaction T in DB do
4:     eliminate infrequent items from T
5:     sort frequent items in the item order
6:     construct a new branch from the sorted frequent items
7:     insert this new branch into T
8: end for
```

TABLE 1. Database and frequent items.

Database	Frequent Item	Sorted Frequent Item
a b c e	a b c	b a c
b d	b d	b d
a c e	a c	a c
a b c f	a b c	b a c
b d g	b d	b d
a b d h	a d b	b a d
a h	a	a
b d g	b d	b d

It should be mentioned that all child nodes of each node in a prefix tree are sorted according to the item order. An example of constructing a prefix tree from a database is illustrated as follows. The database is shown in Table 1, and *min_sup* is set to 3. The item order is the frequency-decreasing order. After a database scan, a set of frequent items with their supports is obtained, i.e., $\{(a: 5), (b: 6), (c: 3), (d: 4)\}$. Thus, the item order is $b < a < d < c$. During the second database scan, infrequent items are eliminated and frequent items are sorted for each transaction. Then, a branch constructed from ordered frequent items is inserted into a prefix tree. The generated prefix tree is shown in Figure 2.

C. RELATED WORK

The FP-Growth algorithm [10] based on modified prefix trees called FP-trees is more efficient than Apriori and the Apriori-like algorithms. All nodes containing the same item are linked in a node-link in a FP-tree. A header table corresponding to a FP-tree stores the information about each item in the tree. The information related to an item includes

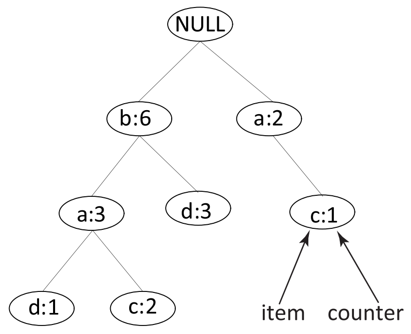

FIGURE 2. Prefix tree.

the item's name, its support and a pointer pointing to the first node of the node-link of the item. After constructing an initial FP-tree from a database, FP-Growth recursively processes each item in the tree. First, a frequent itemset composed of the item and the prefix itemset (assigned as ∅ for the initial FP-tree) is output. Second, FP-Growth identifies frequent items in the conditional database of the item by traversing nodes and related branches along the node-link of the item. Third, FP-Growth constructs a conditional FP-tree of the item with frequent items in its conditional database by traversing nodes and related branches along the node-link. Finally, FP-Growth recursively processes each item in the conditional FP-tree.

There are many FP-Growth-like algorithms such as [10], [25]–[29]. A significant optimization of FP-Growth is to reduce the traversal cost when it constructs conditional FP-trees. Two traversals of a tree are necessary for constructing a conditional FP-tree. Using an array technique, G. Grahne *et al.* proposed the FP-Growth* algorithm which constructs a conditional FP-tree by only one traversal [30]. FP-Growth mines frequent itemsets by processing prefix trees from bottom to top. Another important FP-Growth-like algorithm, AFOPT, mines frequent itemsets by processing prefix trees from top to bottom. The prefix tree used in AFOPT is different from FP-trees. FP-Growth adopts prefix trees using the frequency-descending item order in order to maximize the compactness of prefix trees. However, AFOPT adopts prefix trees using the frequency-ascending item order in order to maximize pruning efficiency. AFOPT was proposed first in [31] (detailed in [32]).

Frequent itemset mining is a seminal problem, there are many extended itemset mining problems and related algorithms. For example, Li Bo *et al.* proposed the TT-Miner algorithm [11] for mining frequent closed itemsets. Sheng Chen *et al.* proposed the APFI-MAX algorithm [12] to mine maximal frequent itemsets over uncertain sensed data. Xiang Li *et al.* proposed the IPDI+ algorithm [13] to mine productive itemsets in dynamic databases. Bay Vo *et al.* proposed the CPHUI-List algorithm [14] for mining closed potential high utility itemsets. Naji Alhusaini *et al.* proposed the LUIM algorithm [15] which is a new low utility itemset mining framework.

With the great advances on hardware and software, many researchers start to make the use of these conveniences to mine frequent itemsets. For example, GPUs computation has been a powerful tool for frequent itemset mining. Using GPUs, A. Cano *et al.* proposed an algorithm for mining association rules derived from frequent itemsets [16]. Exploiting GPU and cluster parallelism, Y. Djenouri *et al.* proposed a frequent itemset mining algorithm which needs only one database scan [17]. G. Teodoro *et al.* proposed a tree projection-based frequent itemset mining algorithm on multicore CPUs and GPUs [18]. Based on the MapReduce framework, there are some algorithms revised from Apriori that can mine frequent itemsets from big databases [19], [20]. Further, there are frequent itemset mining algorithms adopting MapReduce and genetic programming [21], [22].

In this paper, we only focus on the frequent itemset mining problem on a traditional computing configure. A nice solution to the problem is bound to contribute to solutions to the variants and extensions of the problem.

III. CONCEPT AND PRINCIPLE

DPT searches a set-enumeration tree for frequent itemsets in a depth-first manner. DPT uses a prefix tree representing a database. The section introduces several concepts and principles on which DPT is based.

A. PROPERTIES OF A SET-ENUMERATION TREE

Convention 1: According to the item order of a set-enumeration tree, a hind item is larger than a front item and a front item is smaller than a hind item.

A set-enumeration tree holds the following properties according to Convention 1.

Property 1: In a set-enumeration tree, the maximal item in an itemset represented by node N is larger than the maximal item in an itemset represented by any N's left sibling node.

Property 2: In a set-enumeration tree, the maximal item in an itemset represented by node N is smaller than the maximal item in an itemset represented by any N's right sibling node.

Property 3: In a set-enumeration tree, the maximal item in an itemset represented by node N is smaller than the maximal item in an itemset represented by any N's descendant node.

B. CONDITIONAL AND POST-CONDITIONAL DATABASE

For an itemset X, its conditional database and its post-conditional database are defined as follows.

Definition 1: The conditional database of an itemset X is composed of all transactions containing X, and infrequent items and X itself are removed from these transactions.

Each node in a set-enumeration tree represents an itemset. Suppose a mining algorithm searches a set-enumeration tree in the depth-first way, and then the generation order of frequent itemsets is according to the item order which the set-enumeration tree holds. For example, suppose that a depth-first algorithm searches the set-enumeration tree in Figure 3(a) for frequent itemsets and all itemsets are frequent, and then the generation order of frequent itemsets is according to the item order: $b < a < d < c$. Figure 3(b) lists frequent itemsets that are generated in the top-to-bottom order.

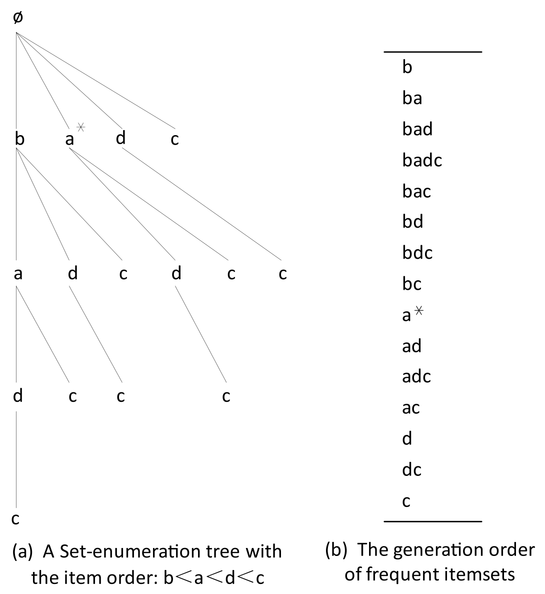
ø
b
a*
d
c
(a) A Set-enumeration tree with the item order: b<a<d<c
b
ba
bad
badc
bac
bd
bdc
bc
a*
ad
adc
ac
d
dc
c
(b) The generation order of frequent itemsets

FIGURE 3. Generation order of frequent itemsets.

TABLE 2. Conditional and post-conditional database.

Conditional database of a	Post-conditional database of a
b c	c
c	c
b c	c
b d	d

When a depth-first mining algorithm arrives at the node representing itemset $\{a\}$ in Figure 3(a), those nodes representing itemsets $\{ba\}$, $\{bad\}$, $\{badc\}$, $\{bac\}$ have been explored, while those nodes representing itemsets $\{ad\}$, $\{adc\}$, $\{ac\}$ are still not explored. In order to judge whether itemsets $\{ad\}$, $\{adc\}$, $\{ac\}$ are frequent or not, the algorithm only needs to analyze a part-rather-than-whole of $\{a\}$'s conditional database. This part of $\{a\}$'s conditional database only relates to the items that are larger than item a, and they are items d and c. Note that the part of $\{a\}$'s conditional database related to item b can be discarded, because all nodes representing itemsets containing items b and a have been explored by the depth-first algorithm. When a depth-first mining algorithm arrives at a node in a set-enumeration tree, the requisite part of the conditional database of the itemset represented by the node is the itemset's post-conditional database.

Definition 2: The post-conditional database of an itemset X is the conditional database of X without these items smaller than the maximal item in X.

For example, the frequency-descending order is $b < a < d < c$ for the database in Table 1. Using the item order, the conditional database of itemset $\{a\}$ and the post-conditional database of $\{a\}$ are listed in Table 2, given $min\_sup = 3$.

Lemma 1: When a depth-first mining algorithm arrives at a node in a set-enumeration tree, the algorithm can mine frequent itemsets by only using the post-conditional database of the itemset represented by the node.

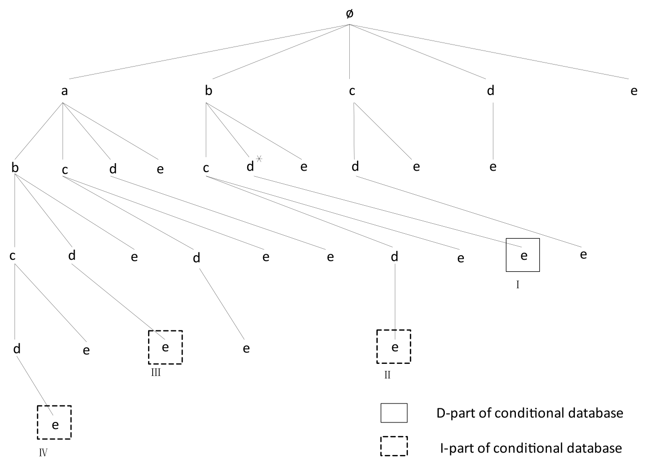
ø
a
b
c
d
e
I
II
III
IV
D-part of conditional database
I-part of conditional database

FIGURE 4. Distribution of the post-conditional database of $\{bd\}$.

Proof: Lemma 1 can be deduced from the generation order of frequent itemsets mined by a depth-first algorithm and Definition 2.

C. SET-ENUMERATION TREE VS. PREFIX TREE

Given the same item set and the same item order, there are several properties between a set-enumeration tree and a prefix tree.

Property 4: After eliminating the counter of each node in a prefix tree, the tree is an exact part of a set-enumeration tree with the same item set and the same item order.

For example, the prefix tree in Figure 2 after eliminating the counter of each node is an exact part of the set-enumeration tree in Figure 3(a).

Lemma 2: A prefix tree holds Property 1, Property 2, and Property 3 of a set-enumeration tree with the same item set and the same item order.

Proof: This lemma can be deduced from Property 4.

D. POST-CONDITIONAL DATABASE DISTRIBUTION

Lemma 3: For an itemset represented by a node in a prefix tree, its conditional database is completely distributed in the prefix tree.

Proof: According to Algorithm 1, a prefix tree stores all transactions without infrequent items in a database, thus Lemma 3 holds.

Lemma 4: For an itemset represented by a node in a prefix tree, its post-conditional database is completely distributed in the prefix tree.

Proof: Lemma 4 can be deduced from Definition 1, Definition 2 and Lemma 3.

Suppose a prefix tree contains items $i_1, i_2, i_3, \ldots, i_n$, and the item order is $i_1 < i_2 < i_3 < \cdots < i_n$. For itemset $X = \{i_{k1}i_{k2}i_{k3}\ldots i_{km}\}$ $(1 \leq k1 < k2 < k3 \cdots < km \leq n)$, its post-conditional database is distributed in both the subtree rooted at the node representing itemset $\{i_{k1}i_{k2}i_{k3}\ldots i_{km}\}$ and those subtrees rooted at the left sibling nodes of the nodes representing itemsets $\{i_{k1}\}$, $\{i_{k1}i_{k2}\}$, $\{i_{k1}i_{k2}i_{k3}\}, \ldots, \{i_{k1}i_{k2}i_{k3}\ldots i_{km}\}$. For example, in Figure 4, the post-conditional database of itemset $\{bd\}$ is composed of part I, II, III and IV. Part I is the subtree rooted at the node representing itemset $\{bd\}$. Parts II, III and IV are in the subtrees rooted at the nodes representing itemsets $\{a\}$ and $\{bc\}$, and these two nodes are the left sibling nodes of the nodes representing itemsets $\{b\}$ and $\{bd\}$ respectively. The counter in each node in the prefix tree in Figure 4 is not depicted, which does not influence our discussions. For the post-conditional database of itemset $X = \{i_{k1}i_{k2}i_{k3}\ldots i_{km}\}$, the following lemma holds.

Definition 3: In a prefix tree, the subtree rooted at the node representing X is a part of X's post-conditional database, and this part is called D-part (direct part) of X's post-conditional database.

Definition 4: In a prefix tree, those parts of X's post-conditional database in the subtrees rooted at the left sibling nodes of the nodes representing X and the prefix itemsets of X are called I-parts (indirect parts) of X's post-conditional database.

Convention 2: All items in an itemset are sorted according to the item order.

Lemma 5: In a prefix tree, the post-conditional database of X consists of the D-part and I-part(s) of X's post-conditional database.

Proof: First, it is obvious that the subtree rooted at the node representing X is a part of X's post-conditional database. Second, for X or its prefix $\{i_{k1}i_{k2}i_{k3}\cdots i_{k(i-1)}i_{ki}\}$ $(1 \leq i \leq m)$ represented by node N in the prefix tree, a N's left sibling node N_l represents itemset $\{i_{k1}i_{k2}i_{k3}\cdots i_{k(i-1)}i_l\}$, and a N's right sibling node N_r represents itemset $\{i_{k1}i_{k2}i_{k3}\cdots i_{k(i-1)}i_r\}$. It can be deduced from Lemma 2 and Property 1 that i_l is smaller than i_{ki}. Therefore, according to Property 3, it is possible that there is a descendant node of N_l representing itemset $\{i_{k1}i_{k2}i_{k3}\cdots i_{k(i-1)}i_l\cdots i_{ki}\}$. We can conclude from Property 3 that it is possible that there is a descendant node N' of N_l representing itemset $\{i_{k1}i_{k2}i_{k3}\cdots i_{k(i-1)}i_l\cdots i_{ki}\cdots i_{k(i+1)}\cdots i_{km}\}$. If N' does exists, the subtree rooted at N' is an I-part of the post-conditional database of itemset X. For N_r, it can be deduced from Lemma 2 and Property 2 that i_{ki} is smaller than i_r. According to Property 3, none of descendant nodes of N_r can represent itemset $\{i_{k1}i_{k2}i_{k3}\cdots i_{k(i-1)}i_r\cdots i_{ki}\}$. Therefore there is no node representing itemset $\{i_{k1}i_{k2}i_{k3}\cdots i_{k(i-1)}i_r\cdots i_{ki}\cdots i_{k(i+1)}\cdots i_{km}\}$ in the subtree rooted at N_r. Therefore, this lemma holds.

For example, for the node representing itemset $\{bd\}$ in Figure 4, its post-conditional database is composed of part I, part II, part III, and part IV. Part I is the D-part of the post-conditional database of $\{bd\}$, while Parts II, III, and IV are the I-parts of the post-conditional database of $\{bd\}$.

IV. MINING FREQUENT ITEMSETS

In this section, we introduce the main frame of the DPT algorithm and three optimization techniques, and we also discuss the advantage of DPT in representing the set of all frequent itemsets.

A. THE MAIN FRAME OF DPT

FP-Growth and FP-Growth-like algorithms are more efficient than Apriori and Apriori-like algorithms, but FP-Growth and FP-Growth-like algorithms have to spend much time in constructing many conditional prefix trees when mining frequent itemsets, which degrades the performance of these algorithms. The DPT algorithm uses only one prefix tree rather than many prefix trees to mine all frequent itemsets with their supports.

DPT is a depth-first mining algorithm. Based on Lemma 1 and Lemma 3, the basic idea of DPT is that the post-conditional database of an itemset represented by a node can be incrementally constructed before DPT arrives at the node. Thus, the complete post-conditional database of the itemset can be used when DPT arrives at the node. To construct the complete post-conditional database of an itemset, the major operation of DPT is, when it arrives at a node, to copy the subtree rooted at the node to the right sibling nodes of the node. Using copying operations, the I-parts of the post-conditional database of an itemset are added to the D-part of the post-conditional database of the itemset. When DPT arrives at the node representing the itemset, the algorithm has added all I-parts of the itemset to the D-part of the itemset. At this time, the complete post-conditional database of the itemset is just the subtree rooted at the node. The justification of the above statement is based on Lemma 4 and Lemma 5.

After constructing a prefix tree from a database, DPT recursively processes each node in the tree as follows. For node N, DPT first copies the subtree rooted at N into its right sibling nodes. Subsequently, if $N.counter$ is no smaller than a given *min_sup*, DPT outputs an itemset composed of the items contained in the path from the root node to N with $N.counter$ as its support. After recursively processing all child nodes of N, DPT frees N. If $N.counter$ is smaller than *min_sup*, DPT frees the whole subtree rooted at N after copying the subtree. For a prefix tree, the counters of all descendant nodes of N are equal to or smaller than $N.counter$, and then these counters are smaller than *min_sup* if $N.counter$ is smaller than *min_sup*. Therefore, all itemsets represented by descendant nodes of N are infrequent if $N.counter$ is smaller than *min_sup*, and DPT can free the subtree rooted at N.

Algorithm 2 gives the main frame of the DPT algorithm.

After a prefix tree rooted at R is constructed from an original database and F is assigned as $\varnothing$, all frequent itemsets with their supports can be output by calling DPT(*F, R, min_sup*).

Note that all child nodes of a node in a prefix tree are sorted in the item order. Line 1 in Algorithm 2 is to add I-parts of the post-conditional databases of the itemsets represented by N's right sibling nodes to D-parts of the post-conditional database of these itemsets. If $N.counter$ is no smaller than *min_sup*, which indicates that the itemset represented by N is frequent and thus be output (lines 3 and 4). Subsequently, each of N's

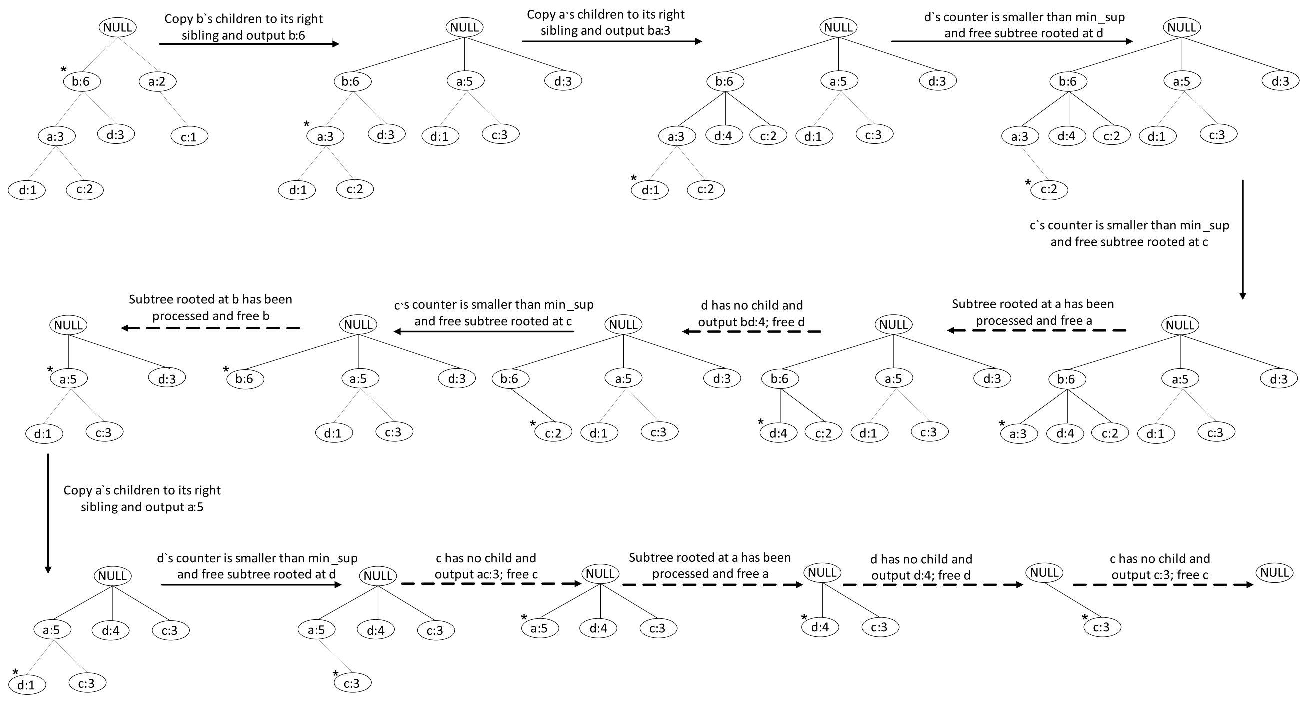
Copy b`s children to its right sibling and output b:6
Copy a`s children to its right sibling and output ba:3
d`s counter is smaller than min_sup and free subtree rooted at d
c`s counter is smaller than min_sup and free subtree rooted at c
Subtree rooted at a has been processed and free a
d has no child and output bd:4; free d
c`s counter is smaller than min_sup and free subtree rooted at c
Subtree rooted at b has been processed and free b
Copy a`s children to its right sibling and output a:5
d`s counter is smaller than min_sup and free subtree rooted at d
c has no child and output ac:3; free c
Subtree rooted at a has been processed and free a
d has no child and output d:4; free d
c has no child and output c:3; free c

FIGURE 5. An illustrated example of DPT.

Algorithm 2 DPT_Frame(F, N, min_sup)

```
Input:
    F a prefix itemset, initially ∅
    N the node representing itemset F∪{N.item}
    min_sup the minimum support threshold
Output:
    all frequent itemsets with F as prefix
1:  copy the subtree rooted at N into N's right sibling nodes
2:  if N.counter ≥ min_sup then
3:      F' = F ∪ {N.item}
4:      output F' and N.counter as support
5:      for each child node C of N do
6:          DPT_Frame(F', C, min_sup)
7:      end for
8:      free N
9:  else
10:     free N and the subtree rooted at N
11: end if
```

child nodes is recursively processed (lines 5-7). After that, DPT frees N. If $N.counter$ is smaller than min_sup, it can be deduced that the counters of all descendant nodes of N is also smaller than min_sup definitely and thus the itemsets represented by these nodes are infrequent. In this case, DPT frees N and the whole subtree rooted at N (line 10).

Figure 5 illustrates how the DPT algorithm processes the prefix tree in Figure 2, given $min\_sup = 3$.

B. OPTIMIZATIONS

It can be observed from Algorithm 2 that there are many copying, freeing, and traversing operations, which are very time-consuming We propose three optimization techniques aiming at reducing these operations as follows.

1) EFFICIENTLY PROCESSING SMALL NODES

Definition 5: A node N in a prefix tree is a big node if $N.counter$ is no smaller than min_sup, when DPT arrives at N.

Definition 6: A node N in a prefix tree is a small node if $N.counter$ is smaller than min_sup, when DPT arrives at N.

The first step of DPT is to copy the subtree rooted at N to N's right sibling nodes, in which DPT has to generate branches that are in the subtree rooted at N but not in the subtrees rooted at N's right sibling nodes. These branches are appended to the subtrees rooted at N's right sibling nodes. However, the subtree rooted at a small node is no longer used according to Algorithm 2, and therefore we can first cut the subtree off and merge the subtree with the subtrees rooted at the right sibling nodes of the small node. This is an efficient technique to process small nodes because there is no need to generate any new branch.

2) A SINGLE BRANCH OPTIMIZATION

As the AFOPT algorithm [31], [32] does, for a node at which a single-branch subtree is rooted, we can generate all combinations of the items contained in the single-branch and the support of each combination is the counter in the lowest level node. Compared with recursive calls, this way is more efficient.

To implement this optimization, AFOPT [31] has to traverse the subtree rooted at each node when the algorithm arrives at the node. An improved AFOPT algorithm [32] uses a array structure to store a single-branch. The former spends extra time and the latter complicates data structure.

Different from these algorithms, a single-branch subtree rooted at a big node can be directly identified instantly after the copying operation, and thus our algorithm can easily perform the optimization.

3) FILTERING NODES CONTAINING INFREQUENT ITEMS OUT

The DPT algorithm spends much time in performing copying subtrees and merging subtrees. In fact, when DPT processes a node, the algorithm can filter its descendant nodes containing infrequent items out, which can lead to the decrease in the size of the subtree rooted at the node.

Suppose node N' representing itemset X' is a child node of node N representing itemset X, and then X is a prefix itemset of X'. Because the post-conditional database of X' is a part of the post-conditional database of X, infrequent items in the post-conditional of X are also infrequent in the post-conditional database of X'. Thus, nodes containing infrequent items for X can be removed.

According to the main frame of DPT, the post-conditional database of X' is just represented by the subtree rooted at N' when DPT arrives at N'. If N' is a big node, the frequencies of items in the subtree rooted at N' can be counted in the process of copying the subtree. Relatively infrequent items that are frequent in the post-conditional database of X but infrequent in the post-conditional database of X' can be identified after the copying operation. Nodes containing relatively infrequent items for X' can be also filtered out when DPT processes all descendant nodes of N'.

Therefore the lower the level of a node is, the fewer frequent items there are in post-conditional databases and consequently the fewer nodes DPT needs to copy or merge.

C. THE DYNAMIC PREFIX TREE ALGORITHM

After incorporating the above optimization techniques into the main frame of DPT, Algorithm 3 shows the DPT algorithm.

Algorithm 3 DPT(*F, SFI, N, min_sup*)

```
Input:
    F a prefix itemset, initially $\varnothing$
    SFI the set of frequent items in the post-conditional database of F
    N the node representing itemset $F \cup \{N.item\}$
    min_sup the minimum support threshold
Output:
    all frequent itemsets with F as prefix
1:  if $N.counter < min\_sup$ then
2:      merge the subtree rooted at N into N's right sibling nodes without considering nodes with items $\notin SFI$
3:  else
4:      copy the subtree rooted at N into N's right sibling nodes without considering nodes with items $\notin SFI$
5:      if the subtree is a single branch then
6:          $L$ = the lowest level big node in the branch
7:          for each item combination X from N to L do
8:              $F' = F \cup X$
9:              output F' with its support
10:         end for
11:         free the single branch
12:     else
13:         $F' = F \cup \{N.item\}$
14:         output F' and N.counter
15:         $SFI' = SFI$ - the relatively infrequent items for N
16:         for each child node C of N do
17:             DPT(F', SFI', C, min_sup)
18:         end for
19:     end if
20: end if
21: free N
```

The DPT algorithm first judges whether node N is a small node or not (line 1). If N is a small node, the algorithm merges its subtree into its right sibling nodes (line 2). If N is a big node, the algorithm copies its subtree into its right sibling nodes (line 4). For a big node, DPT further processes its subtree as follows (lines 5-20). If the subtree is a single branch, DPT locates the lowest level big node L in the branch (lines 5-6). Then, DPT generates all combinations of the items contained in the path from N to L. Each combination is joined to the prefix itemset F, which results in a new frequent itemset F' (line 8). DPT outputs F' with its support that is the smallest one among counters in nodes with items in the combination (line 9). At last, the single branch is freed (line 11). If the subtree is not a single branch, *N.item* is joined to the prefix itemset F, which results in a new frequent itemset F' (line 13). DPT outputs F' with *N.counter* as its support (line 14). In the process of copying the subtree rooted at N, DPT also counts the frequencies of items in the subtree and thus can identify relatively infrequent items for N. After eliminating the relatively infrequent items for N from *SFI*, the algorithm obtains the frequent item set *SFI'* for N's child nodes (line 15). Subsequently, DPT recursively calls itself for each child node C of N. In the end, the algorithm frees node N.

After constructing a prefix tree rooted at R from a database and identifying the set S of frequent items in the database, all frequent itemsets with their supports can be mined using Algorithm 4.

D. REPRESENTING ALL FREQUENT ITEMSETS

Previous algorithms output all frequent itemsets in a list form. However, it is very inconvenient to make use of frequent itemsets in a list.

A prefix tree is a better alternative of representing all frequent itemsets, because its compact structure needs less space than a list structure. Moreover, when applications judges whether an itemset is frequent or not or fetches the support of a frequent itemset, operations on a prefix tree are faster than those on a list. This is because the time complexity of these operations on a prefix tree is proportional to the length of an itemset for these applications, but the time complexity of these operations on a list is proportional to the total number of frequent itemsets. Generally, the number of frequent

Algorithm 4 Mine Frequent Itemsets

```
Input:
    $R$ is the root of a prefix tree from a database
    $S$ is the set of frequent items in the database
    $min\_sup$ the minimum support threshold
Output:
    all frequent itemsets with their supports
1: $F = \varnothing$
2: $SFI = S$
3: for each child node $C$ of $R$ do
4:     DPT($F$, $SFI$, $C$, $min\_sup$)
5: end for
```

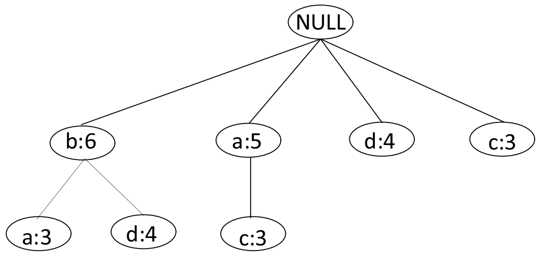

FIGURE 6. A prefix tree representing all frequent itemsets in the example database.

itemsets greatly exceeds the length of the longest frequent itemset.

Although a prefix tree representing frequent itemsets can be constructed from a list of frequent itemsets, it is evident that the extra time is spent in constructing such a prefix tree. An interesting advantage of the DPT algorithm is that a prefix tree representing all frequent itemsets with their supports can be directly output so long as the DPT algorithm is slightly modified as follows. (1) The single branch optimization is removed from the algorithm. (2) The algorithm does not free big nodes after processing them. For example, using these modifications, the algorithm can output a prefix tree depicted in Figure 6, which represents all frequent itemsets in the example database in Table 1, given $min\_sup = 3$.

V. EXPERIMENT

We have done experiments to evaluate the performance of the proposed DPT algorithm. This section reports experimental results.

A. EXPERIMENTAL SETUP

Six databases were used in our experiments. These databases were downloaded from the FIMI repository [37]. Database Accidents is traffic accidents data from a region of Belgium. Databases Chess and Connect are derived from game states. Database mushroom describes characteristics of 23 species of mushrooms. Database Pumsb contains census data. T40I10D100K is a synthetic database generated from IBM Almaden synthetic data generator. Table 3 lists statistical information about these databases, including the number of transactions, the number of distinct items, the average number of items in a transaction, and the attribute of a database. The attribute of a database is defined as the

TABLE 3. Statistical information about databases.

Database	#Tran	#Items	AvgLen	Attribute
Accidents	340183	468	33.8	moderate
Chess	3196	75	37	very dense
Connect	67557	129	43	dense
Mushroom	8124	119	23	dense
Pumsb	49046	2113	74	sparse
T10I4D100K	100000	870	10.1	very sparse

ratio of the average transaction length to the number of distinct items. The bigger the ratio for a database is, the denser the database is.

All codes were written in C++, used the same libraries, and were compiled using GCC (version 7.2.0). The experiments were performed on a 2.10 GHz machine (Intel Xeon E5-2620 v4) with 32 GB of memory, running the Debian 9.1 (Linux 4.9.0-4) operating system.

B. POST-CONDITIONAL DATABASE EVALUATION

We first evaluated the post-conditional database which is the core data of the DPT algorithm. Each post-conditional database is represented by a subtree in a prefix tree constructed from a database. Therefore, we measured the total size of all post-conditional databases with the number of all nodes generated by DPT.

The proposed three optimization techniques can effectively reduce the total size of all post-conditional databases, i.e., the number of generated nodes. We first turned off the three optimization techniques and recorded the number of nodes generated by DPT. Subsequently, we turned on these techniques one by one and recorded the number of nodes generated by DPT. For comparison, we also recorded the number of nodes generated by FP-Growth. Using only one prefix tree, DPT always generates fewer nodes than FP-Growth. Therefore, we denote the number of nodes generated by FP-Growth as 1, and denote the number of nodes generated by DPT as a ratio of the number to the number of nodes generated by FP-Growth.

Figure 7 depicts these ratios on databases Accidents (moderate), Chess (very dense), and T10I4D100K (very sparse). For moderate database Accidents, each of the three optimization techniques can significantly reduce the number of nodes as shown in Figure 7(a). For dense database Chess, it is not very remarkable to reduce the number of nodes using these techniques because (conditional) prefix trees from the database are very small. However, the decrease in generated nodes still achieves about 10%, when the three techniques are used together for the database as shown in Figure 7(b). Sparse database T10I4D100K leads to large and bushy (conditional) prefix trees, and thus the techniques of both efficiently processing small nodes and filtering infrequent items work well as shown in Figure 7(c). Compared with the number of nodes generated by FP-Growth, the number of nodes generated by DPT decreases by 20% at least even though the algorithm does not use any optimization technique as shown by black lines in Figure 7. Using the three optimization techniques together, the number of nodes can decrease by about 50% as shown by red lines in Figure 7.

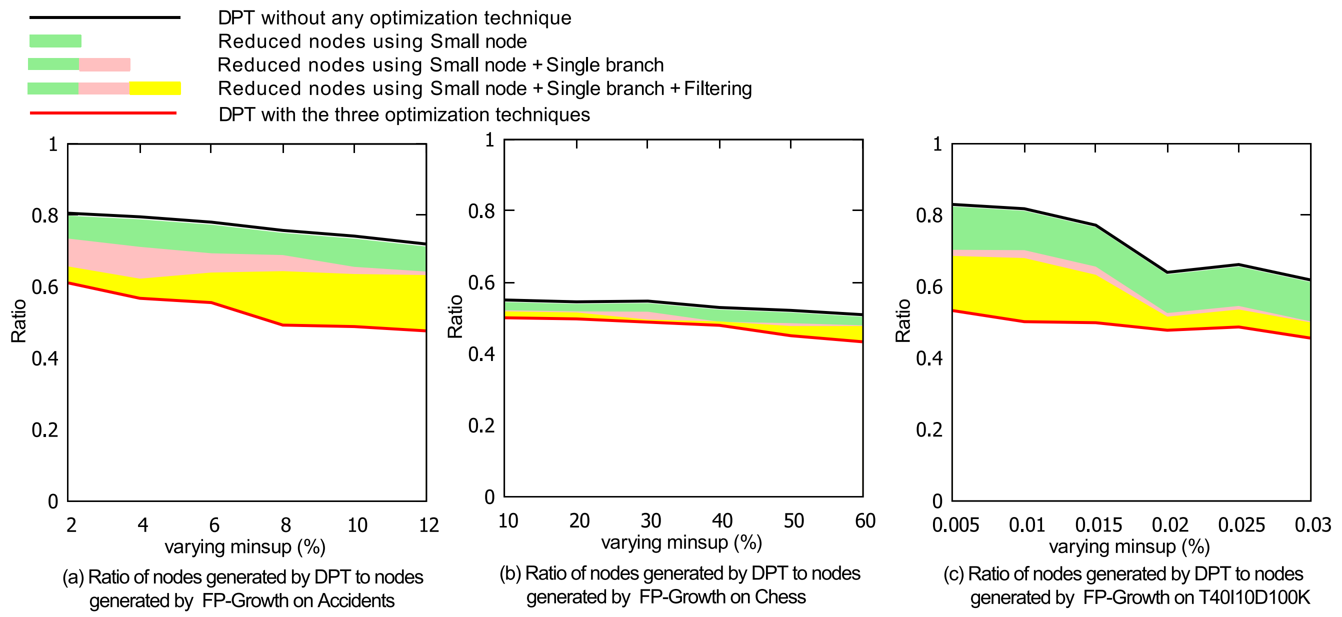

FIGURE 7. **Node number comparison between FP-Growth and DPT (with or without optimization techniques).**

C. PERFORMANCE COMPARISON

We compared the DPT algorithm against Apriori [9], FP-Growth [25], AFOPT [32], and LCM [38]. Apriori and FP-Growth are two classic mining algorithms, and their implementations were downloaded from [36]. The AFOPT algorithm also involves subtree merging operations, but it has to construct many prefix trees as FP-Growth does. The LCM algorithm shows very competitive and robust performance in the first IEEE ICDM Workshop on frequent itemset mining. The implementations of AFOPT and LCM were downloaded from [37].

There are a number of recent algorithms for mining frequent itemsets [16]–[22], and these algorithms make the best use of high performance hardware and software such as GPU and MapReduce. However, these algorithms are still based on classic mining algorithms such as Apriori and FP-Growth. It is inappropriate to compare DPT with these algorithms because DPT does not utilize these conveniences and runs on a traditional computing configure. DPT can be also further improved using these conveniences, but this is out of the topic of the paper and is future work. To the best of our knowledge, FP-Growth, AFOPT, and LCM are ones of the best algorithms on a traditional computing configure, and thus we compared DPT with these algorithms.

Given a database and a *min_sup*, the runtime of an algorithm was recorded, which is calculated as the sum of the time for reading an input file, mining patterns, and writing the results to an output file. The outputs of all algorithms are the same for each mining task and were written to "/dev/null".

Figure 8 depicts the running times of the five algorithms on the six databases. On database Accidents as shown in Figure 8(a), all the algorithms can finish mining tasks for high *min_sups*. For low *min_sups* such as 6%, the running time of Apriori is so long that we had to terminate the algorithm. DPT performs best on this database. On database Chess as shown in Figure 8(b), the five algorithms similarly perform for high *min_sups*. When *min_sups* become low, DPT is the fastest one but the other algorithms become slow. FP-Growth and AFOPT similarly perform on database Connect, and their curves almost overlap as shown in Figure 8(c). For $min\_sup = 70\%$ on Connect, the five algorithm have almost the same running time. However, for $min\_sup = 20\%$, the difference of their running times achieves to several thousand seconds. As shown in Figure 8(d), DPT and LCM are two fastest algorithms on database Mushroom for all *min_sups*. On this database, AFOPT is faster than FP-Growth for high *min_sups* but FP-Growth is faster than AFOPT for low *min_sups*. DPT is faster than the other algorithms on database Pumsb for all *min_sups* as shown in Figure 8(e). Database T10I4D100K is very sparse, which results in large prefix trees with many branches. Therefore, LCM is faster than FP-Growth, AFOPT, and DPT that are based on prefix trees.

D. MEMORY USAGE AND SCALABILITY

Figure 9 depicts the peak memory usages of DPT, FP-Growth, LCM and AFOPT on the six databases. Each chart in the figure corresponds to a chart in Figure 8. We didn't record the peak memory usage of Apriori because the algorithm ran too much time and we had to terminate it for some mining tasks.

It can be observed from Figure 9 that AFOPT consumes the largest amount of memory compared with DPT, FP-Growth, and LCM in most cases. This is because that AFOPT adopts the frequency-ascending order to construct prefix trees that have a low compression ratio. In most cases, DPT using only

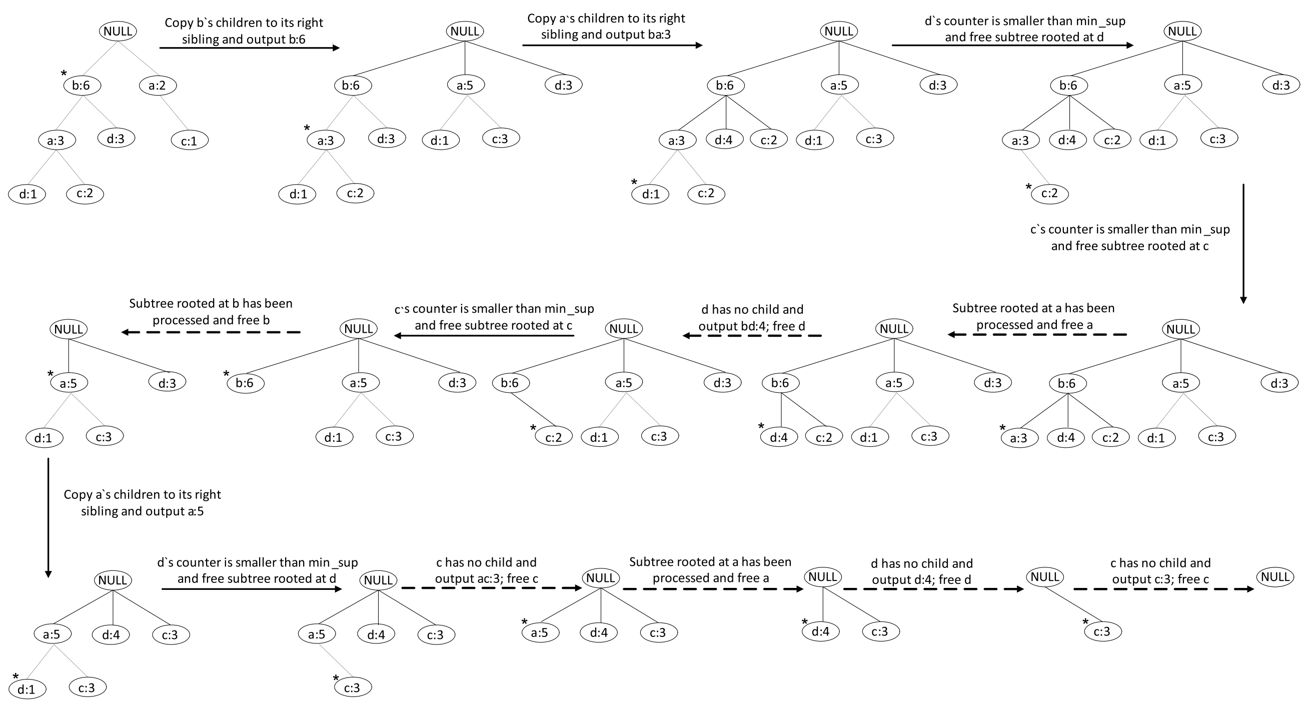
Copy b`s children to its right sibling and output b:6
Copy a`s children to its right sibling and output ba:3
d`s counter is smaller than min_sup and free subtree rooted at d
c`s counter is smaller than min_sup and free subtree rooted at c
Subtree rooted at a has been processed and free a
d has no child and output bd:4; free d
c`s counter is smaller than min_sup and free subtree rooted at c
Subtree rooted at b has been processed and free b
Copy a`s children to its right sibling and output a:5
d`s counter is smaller than min_sup and free subtree rooted at d
c has no child and output ac:3; free c
Subtree rooted at a has been processed and free a
d has no child and output d:4; free d
c has no child and output c:3; free c

FIGURE 8. Runtime.

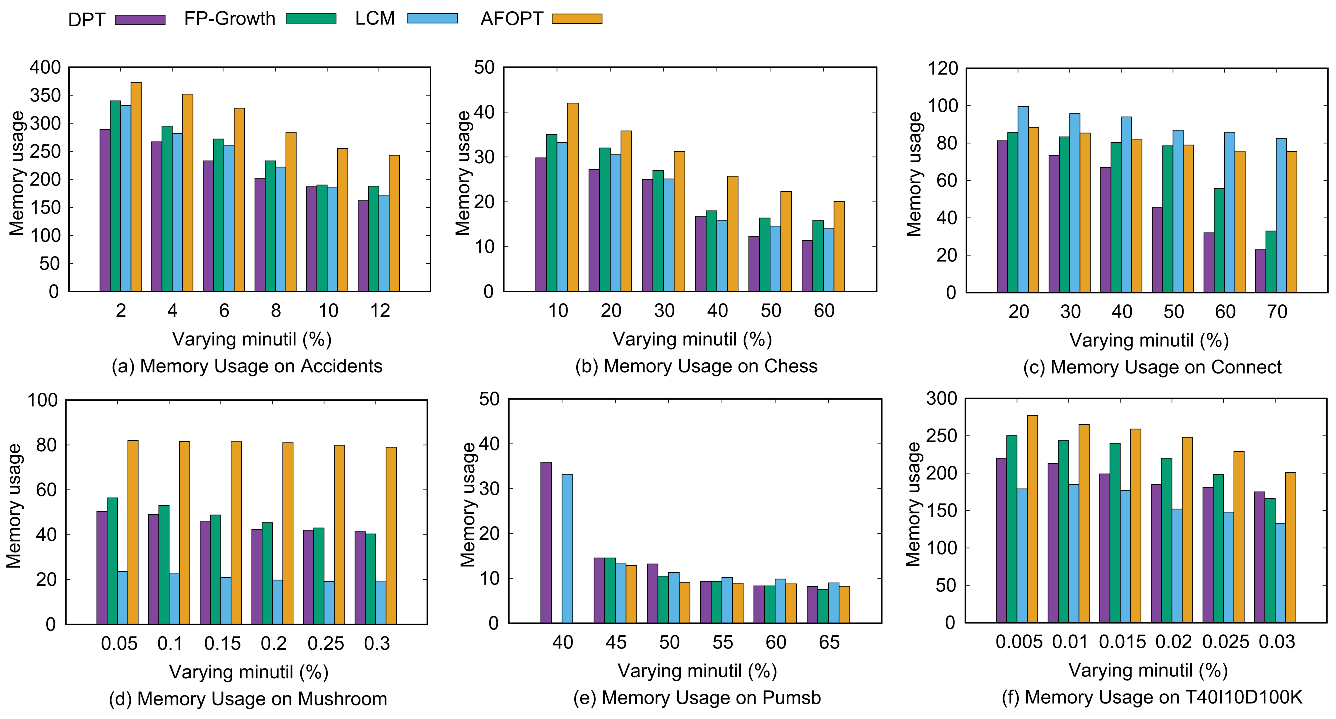

FIGURE 9. Memory usage.

one prefix tree consumes less memory than FP-Growth and AFOPT using many prefix trees. DPT consumes the least amount of memory on databases Accidents, Chess, and Connect. Databases Pumsb and T10I4D100K are sparse, which leads to prefix trees with many branches, and thus LCM consumes the least amount of memory on the databases in most cases.

In order to test the scalability of the DPT algorithm, we varied the size of Accidents and that of T10I4D100K from 10% to 100% at an interval of 10%. Figure 10(a) depicts the runtimes for Accidents with $min\_sup = 2\%$, and Figure 10(b) is the runtimes for T10I4D100K with $min\_sup = 0.005\%$. It can be observed from experimental results that the runtimes of all algorithms increase with the increasing

TABLE 4. A frequent itemset list vs. a prefix tree representing all frequent itemsets.

Task (Database, Min_sup)	Accidents, 30%	Chess, 60%	Connect, 50%	Mushroom, 5%	Pumsb, 60%	T10I4D100K, 0.1%
Number of frequent itemsets	91,831,296	52,055,040	88,324,400	3,755,706	19,537,366	70,470,947
Size of a list representation	1,794 MB	827 MB	1,827 MB	67 MB	593 MB	1,627 MB
Size of a prefix tree representation	193 MB	76 MB	249 MB	12 MB	98 MB	389 MB
Search time on list	354.43 sec.	235.69 sec.	398.55 sec.	11.26 sec.	122.51 sec.	323.38 sec.
Search time on tree	4.34 sec.	3.12 sec.	4.89 sec.	0.70 sec.	2.24 sec.	4.84 sec.

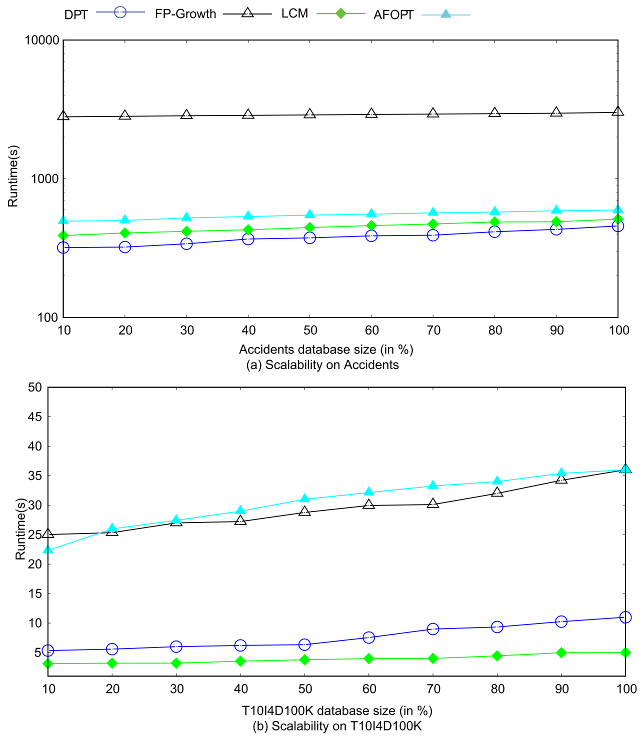

FIGURE 10. Scalability.

number of transactions and that DPT also shows good scalability.

E. APPLYING FREQUENT ITEMSETS

As introduced in Section I, frequent itemsets usually play an important role as an intermediate product in many applications [6]–[8]. In order to access frequent itemsets as fast as possible, they should be stored in an appropriate form. As stated in Section IV, a significant advantage of DPT is that the algorithm can directly output a prefix tree representing all frequent itemsets.

Table 4 compares a frequent itemset list and a prefix tree representing all frequent itemsets in term of size and searching time. We considered six mining tasks, and the number of frequent itemsets of each task is listed in the second line in Table 4. After finishing a task, we stored a frequent itemset list previous algorithms generate and a prefix tree representation DPT generates in memory. The third and fourth lines are the sizes of lists and prefix trees, respectively. It can be observed that the size of a prefix tree is far smaller than that of a corresponding list. Subsequently, we randomly selected 100 frequent itemsets from resultant frequent itemsets and recorded the total time of searching lists and prefix trees for these 100 itemsets in the fifth and sixth lines. It is evident that searching operations on a prefix tree is about two orders of magnitude faster than those on a list.

VI. CONCLUSION

In this paper, we proposed the DPT algorithm for mining all frequent itemsets from a database.

The DPT algorithm is characterized by using only one prefix tree for mining the complete set of frequent itemsets, whereas most of algorithms have to construct many prefix trees. Constructing many prefix trees is very time-consuming, which is avoided by the DPT algorithm. We gave the concept of the post-conditional database of an itemset and analyzed the distribution of the post-conditional database of an itemset in a prefix tree. Using post-conditional databases, DPT can mine frequent itemsets by simply adjusting a prefix tree. We also discussed three optimization techniques, and they had been incorporated into the DPT algorithm. An interesting feature of the DPT algorithm is that it can directly output a prefix tree representing all frequent itemsets after slight modifications, most of mining algorithms only can output a plain list of frequent itemsets.

Experimental results show that the DPT algorithm significantly outperforms FP-Growth-like algorithms and is also competitive with the LCM algorithm, and that a prefix tree representing all frequent itemsets DPT directly outputs can be used more efficient than a list representing them previous algorithms output.

REFERENCES

[1] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1993, pp. 207–216.
[2] Y. Djenouri, D. Djenouri, A. Belhadi, and A. Cano, "Exploiting GPU and cluster parallelism in single scan frequent itemset mining," *Inf. Sci.*, vol. 496, pp. 363–377, Sep. 2019.
[3] F. Guil, "Associative classification based on the transferable belief model," *Knowl.-Based Syst.*, vol. 182, Oct. 2019, Art. no. 104800.
[4] S. Gao, M. Zhou, Y. Wang, J. Cheng, H. Yachi, and J. Wang, "Dendritic neuron model with effective learning algorithms for classification, approximation, and prediction," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 2, pp. 601–614, Feb. 2019.
[5] X. Luo, M. Zhou, Y. Xia, Q. Zhu, A. C. Ammari, and A. Alabdulwahab, "Generating highly accurate predictions for missing QoS data via aggregating nonnegative latent factor models," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 27, no. 3, pp. 524–537, Mar. 2016.
[6] J. Huang, S. Li, and Q. Duan, "Constructing multicast routing tree for inter-cloud data transmission: An approximation algorithmic perspective," *IEEE/CAA J. Automatica Sinica*, vol. 5, no. 2, pp. 514–522, Mar. 2018.
[7] S. B. Prusty, S. Seshagiri, U. C. Pati, and K. K. Mahapatra, "Sliding mode control of coupled tank systems using conditional integrators," *IEEE/CAA J. Automatica Sinica*, vol. 7, no. 1, pp. 118–125, Jan. 2020.
[8] J. M. Luna, P. Fournier-Viger, and S. Ventura, "Frequent itemset mining: A 25 years review," *Wiley Interdiscip. Rev. Data Mining Knowl. Discovery*, vol. 9, no. 6, p. e1329, 2019.

[9] R. Aggrawal and R. Srikant, "Fast algorithm for mining association rules in large databases," in *Proc. VLDB*, 1994, pp. 487–499.

[10] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 1–12.

[11] B. Li, Z. Pei, K. Qin, and M. Kong, "TT-miner: Topology-transaction miner for mining closed itemset," *IEEE Access*, vol. 7, pp. 10798–10810, 2019.

[12] S. Chen, L. Nie, X. Tao, Z. Li, and L. Zhao, "Approximation of probabilistic maximal frequent itemset mining over uncertain sensed data," *IEEE Access*, vol. 8, pp. 97529–97539, 2020.

[13] X. Li, J. Li, P. Fournier-Viger, M. Saqib Nawaz, J. Yao, and J. C.-W. Lin, "Mining productive itemsets in dynamic databases," *IEEE Access*, vol. 8, pp. 140122–140144, 2020, doi: 10.1109/ACCESS.2020.3012817.

[14] B. Vo, L. T. T. Nguyen, N. Bui, T. D. D. Nguyen, V.-N. Huynh, and T.-P. Hong, "An efficient method for mining closed potential high-utility itemsets," *IEEE Access*, vol. 8, pp. 31813–31822, 2020.

[15] N. Alhusaini, S. Karmoshi, A. Hawbani, L. Jing, A. Alhusaini, and Y. Al-sharabi, "LUIM: New low-utility itemset mining framework," *IEEE Access*, vol. 7, pp. 100535–100551, 2019.

[16] A. Cano, J. M. Luna, and S. Ventura, "High performance evaluation of evolutionary-mined association rules on GPUs," *J. Supercomput.*, vol. 66, no. 3, pp. 1438–1461, Dec. 2013.

[17] Y. Djenouri, D. Djenouri, A. Belhadi, and A. Cano, "Exploiting GPU and cluster parallelism in single scan frequent itemset mining," *Inf. Sci.*, vol. 496, pp. 363–377, Sep. 2019.

[18] G. Teodoro, N. Mariano, W. Meira, Jr., and R. Ferreira, "Tree projection-based frequent itemset mining on multicore CPUs and GPUs," in *Proc. 22nd Int. Symp. Comput. Archit. High Perform. Comput.*, Oct. 2010, pp. 47–54.

[19] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based frequent itemset mining algorithms on MapReduce," in *Proc. 6th Int. Conf. Ubiquitous Inf. Manage. Commun. (ICUIMC)*, 2012, pp. 1–8.

[20] J. M. Luna, F. Padillo, M. Pechenizkiy, and S. Ventura, "Apriori versions based on MapReduce for mining frequent patterns on big data," *IEEE Trans. Cybern.*, vol. 48, no. 10, pp. 2851–2865, Oct. 2018.

[21] D. Martín, M. Martínez-Ballesteros, D. García-Gil, J. Alcalá-Fdez, F. Herrera, and J. C. Riquelme-Santos, "MRQAR: A generic MapReduce framework to discover quantitative association rules in big data problems," *Knowl.-Based Syst.*, vol. 153, pp. 176–192, Aug. 2018.

[22] F. Padillo, J. M. Luna, F. Herrera, and S. Ventura, "Mining association rules on big data through map reduce genetic programming," *Integr. Comput.-Aided Eng.*, vol. 25, no. 1, pp. 31–48, 2018.

[23] 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.

[24] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New algorithms for fast discovery of association rules," in *Proc. KDD*, 1997, pp. 283–286.

[25] 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.

[26] G. Gatuha and T. Jiang, "Smart frequent itemsets mining algorithm based on FP-tree and DIFFset data structures," *TURKISH J. Electr. Eng. Comput. Sci.*, vol. 25, pp. 2096–2107, 2017.

[27] N. Shahbazi, R. Soltani, J. Gryz, and A. An, "Building FP-tree on the fly: Single-pass frequent itemset mining," in *Proc. Int. Conf. Mach. Learn. Data Mining Pattern Recognit.* Cham, Switzerland: Springer, 2016, pp. 387–400.

[28] L. Wang, J. Meng, P. Xu, and K. Peng, "Mining temporal association rules with frequent itemsets tree," *Appl. Soft Comput.*, vol. 62, pp. 817–829, Jan. 2018.

[29] P. Fournier-Viger, J. C.-W. Lin, B. Vo, T. T. Chi, J. Zhang, and H. B. Le, "A survey of itemset mining," *Wiley Interdiscipl. Rev., Data Mining Knowl. Discovery*, vol. 7, no. 4, p. e1207, Jul. 2017.

[30] G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using FP-trees," *IEEE Trans. Knowl. Data Eng.*, vol. 17, no. 10, pp. 1347–1362, Oct. 2005.

[31] G. Liu, H. Lu, J. X. Yu, W. Wei, and X. Xiao, "AFOPT: An efficient implementation of pattern growth approach," in *Proc. ICDM Workshop FIMI*, 2003, pp. 1–10.

[32] G. Liu, H. Lu, W. Lou, Y. Xu, and J. X. Yu, "Efficient mining of frequent patterns using ascending frequency ordered prefix-tree," *Date Mining Knowl. Discovery*, vol. 9, pp. 249–274, Nov. 2004.

[33] X. Chen, L. Li, Z. Ma, S. Bai, and F. Guo, "F-miner: A new frequent itemsets mining algorithm," in *Proc. IEEE Int. Conf. e-Bus. Eng. (ICEBE)*, Oct. 2006, pp. 466–472.

[34] M. El-Hajj and O. R. Zaiane, "COFI-tree mining: A new approach to pattern growth with reduced candidacy generation," in *Proc. ICDM Workshop FIMI*, 2003. [Online]. Available: http://www.ceur-ws.org/Vol-90/

[35] M. Adnan and R. Alhajj, "DRFP-tree: Disk-resident frequent pattern tree," *Int. J. Speech Technol.*, vol. 30, no. 2, pp. 84–97, Apr. 2009.

[36] Bart Goethals. (Jan. 2020). *Bart Goethals's Web Pages*. [Online]. Available: http://adrem.ua.ac.be/~goethals/software/

[37] Bart Goethals. (Jan. 2020). *Frequent Itemset Mining Implementations Repository*. [Online]. Available: http://fimi.ua.ac.be/src/

[38] T. Uno, T. Asai, Y. Uchida, and H. Arimura, "LCM: An efficient algorithm for enumerating frequent closed item sets," in *Proc. CEUR Workshop Frequent Item Set Mining Implement.*, vol. 90, 2003, pp. 1–10.

JUN-FENG QU received the M.S. degree in computer science from the China University of Geosciences, China, in 2006, and the Ph.D. degree in computer software and theory from Wuhan University, China, in 2013. He is currently an Associate Professor with the School of Computer Engineering, Hubei University of Arts and Science. His research interests include data mining and database technology.

BO HANG (Member, IEEE) received the M.S. degree in computer science from Harbin Engineering University, Harbin, China, in 2003, and the Ph.D. degree in communication and information system from Wuhan University, in 2012. He is currently a Professor with the School of Computer Engineering, Hubei University of Arts and Science. His research interests include audio signal compression and processing for mobile communication and surveillance.

ZHAO WU received the M.S. degree in computer science from the Wuhan University of Technology, China, in 2003, and the Ph.D. degree in software engineering from Wuhan University, China, in 2007. He is currently a Professor with the School of Computer Engineering, Hubei University of Arts and Science. His research interests include cloud computing and the Internet of Things.

ZHONGBO WU received the B.S. degree in information technology from Central China Normal University, China, in 2001, the M.S. degree in computer application from the Huazhong University of Science and Technology, China, in 2004, and the Ph.D. degree in computer application from the Renmin University of China, in 2010. He is currently a Professor with the School of Computer Engineering, Hubei University of Arts and Science. His research interests include cloud computing, big data, and sensor networks.

QIONG GU received the M.S. degree in computer science and technology and the Ph.D. degree in geosciences information engineering from the China University of Geosciences, Wuhan, China, in 2006 and 2009, respectively. She is currently a Professor with the School of Computer Engineering, Hubei University of Arts and Science. Her research interests include data mining and machine learning. She is a member of the China Computer Federation.

BO TANG received the B.S. degree in mathematics and applied mathematics from the University of Jinan, China, in 2007, the M.S. degree in applied mathematics from the Huazhong University of Science and Technology, China, in 2009, and the Ph.D. degree in computational mathematics from Xi'an Jiaotong University, China, in 2013. Since 2013, he has been working with the Hubei University of Arts and Science, where he is currently an Associate Professor. His research interests include artificial intelligence, big data research, and the Internet of Things.