SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2012 Vol. 55 No. 9: 2008–2030
doi: 10.1007/s11432-012-4638-z

A new algorithm for fast mining frequent itemsets using N-lists

DENG ZhiHong*, WANG ZhongHui & JIANG JiaJian

Key Laboratory of Machine Perception (Ministry of Education), School of Electronics Engineering and Computer Science, Peking University, Beijing 100871*, China*

Received January 18, 2012; accepted March 7, 2012; published online July 19, 2012

Abstract Mining frequent itemsets has emerged as a fundamental problem in data mining and plays an essential role in many important data mining tasks. In this paper, we propose a novel vertical data representation called N-list, which originates from an FP-tree-like coding prefix tree called PPC-tree that stores crucial information about frequent itemsets. Based on the N-list data structure, we develop an efficient mining algorithm, PrePost, for mining all frequent itemsets. Efficiency of PrePost is achieved by the following three reasons. First, N-list is compact since transactions with common prefixes share the same nodes of the PPC-tree. Second, the counting of itemsets' supports is transformed into the intersection of N-lists and the complexity of intersecting two N-lists can be reduced to $O(m + n)$ by an efficient strategy, where m and n are the cardinalities of the two N-lists respectively. Third, PrePost can directly find frequent itemsets without generating candidate itemsets in some cases by making use of the single path property of N-list. We have experimentally evaluated PrePost against four state-of-the-art algorithms for mining frequent itemsets on a variety of real and synthetic datasets. The experimental results show that the PrePost algorithm is the fastest in most cases. Even though the algorithm consumes more memory when the datasets are sparse, it is still the fastest one.

Keywords data mining, frequent itemset mining, data structure, N-lists, algorithm

Citation Deng Z H, Wang Z H, Jiang J J. A new algorithm for fast mining frequent itemsets using N-lists. Sci China Inf Sci, 2012, 55: 2008–2030, doi: 10.1007/s11432-012-4638-z

1 Introduction

Data mining has attracted tremendous amount of attention in the database research community due to its wide applicability in many areas. Frequent itemset mining is a very popular data mining technique and plays an essential role in many important data mining tasks such as mining associations, correlations, causality, sequential itemsets, episodes, multi-dimensional itemsets, max-itemsets, partial periodicity and emerging itemsets [1].

Frequent itemset mining was first proposed by Agrawal et al. [2] for market basket analysis in dealing with the problem of mining association rule. The task of mining association rule is to analyse consumer behaviors by discovering intrinsic associations between different items that customers have bought from stores or supermarkets. A famous example is the discovery that people who buy diapers also frequently buy beers. Such information, about what products are frequently purchased jointly, can be used to

*Corresponding author (email: zhdeng@cis.pku.edu.cn)

optimize the marketing of the products by helping retailers arrange the layout of the store and target certain groups of customers.

Since the first proposal of this new data mining task and its associated efficient mining algorithms, there have been hundreds of follow-up research publications, on various kinds of extensions and applications, ranging from scalable data mining methodologies, to handling a wide diversity of data types, various extended mining tasks, and a variety of new applications [3]. Although lots of papers on mining frequent itemsets have been published in recent years, how to design efficient mining methods is still one of several critical research problems yet to be solved [3,4].

In this paper, we propose a new method, PrePost, for efficiently mining frequent itemsets. PrePost employs a novel data structure, N-list, to represent itemsets. N-list stores all crucial information about itemsets. By combining the search approach of candidate set generation-and-test and the search approach of mining frequent itemsets directly without candidate generation, PrePost achieves very high efficiency in mining frequent itemsets.

Efficiency of PrePost is achieved by the following three reasons. First, N-list has the same compressing degree as the FP-tree, and thus is much more compact than previously proposed vertical structure (such as diffset [5]). Second, the counting of supports is transformed into the intersection of N-lists and the complexity of intersecting two N-lists can be reduced to $O(m+n)$ by an efficient strategy, where m and n are the cardinalities of the two N-lists. For counting supports of itemsets, intersection of N-lists is much more efficient than intersection of Tid-sets by avoiding unnecessary comparison. Third, PrePost can find frequent itemsets without generating candidate itemsets in some cases by using the single path property of N-list. The single path property of N-list can efficiently deal with the problem of too many candidate itemsets of Apriori-like method in local ranges.

For evaluating the performance of PrePost, we conduct a comprehensive performance study to compare the performance of PrePost with four state-of-the-art mining algorithms, which are FP-growth, FP-growth*, dEclat and eclat_goethals. The experimental results show that PrePost is faster than all these four algorithms. In addition, the results also confirm that N-list is a very compact structure, whose average length is about several orders of magnitude smaller than that of diffset, which is a kind of efficient Tid-set.

The remainder of the paper is organized as follows. Section 2 presents related work. Section 3 introduces the detailed problem description, the N-list structure and its basic properties. Section 4 develops an N-list-based algorithm called PrePost for efficiently finding frequent itemsets. Section 5 presents experimental results and performance study. Section 6 summarizes our study and points out some future research issues.

2 Related work

Most of the previously proposed algorithms for mining frequent itemsets can be clustered into two groups: the Apriori-like method and the FP-growth method [6]. The Apriori-like method is based on anti-monotone property [7], called Apriori, which states that if any length k itemset is not frequent, its length $(k+1)$ super-itemset also cannot be frequent. The Apriori-like method employ candidate set generation-and-test strategy to discover frequent itemsets. That is, it generates candidate length $(k+1)$ itemsets in the $(k+1)$th pass using frequent length k itemsets generated in the previous pass, and counts the supports of these candidate itemsets in the database. A lot of studies, such as [5,7–10], adopt the Apriori-like method. Different from the Apriori-like method, the FP-growth method mines frequent itemsets without candidate generation and has proven very efficient. The FP-growth method achieves impressive efficiency by adopting a highly condensed data structure called FP-tree (frequent itemset tree) to store databases and employing a partitioning-based, divide-and-conquer approach to mine frequent itemsets. Some studies, such as [1,11–13], adopt the FP-growth method.

The Apriori-like method achieves good performance by reducing the size of candidates. However, previous studies reveal that it is highly expensive for Apriori-like method to repeatedly scan the database and check a large set of candidates by itemset matching [1]. In order to deal with these problems,

a number of vertical mining algorithms have been proposed [5,9,10]. Unlike the traditional horizontal transactional database format, each item in a vertical database is associated with its corresponding Tid-set, the set of all transaction ids where it appears. The advantage of vertical database format is that the counting of supports of itemsets can be obtained via Tid-set intersection, which avoids scanning a whole database. Tid-set is much simpler than complex hash or trees used in horizontal algorithms and is also more efficient than them in counting supports of itemsets. Vertical mining algorithms have been shown to be very effective and usually outperform horizontal mining methods [5]. Despite the advantages of the vertical database format, the vertical mining algorithms become ineffective since the intersection time becomes too long when Tid-set cardinality (such as for very frequent items) is very large or there are a very large number of transactions [5]. The FP-growth method wins an advantage over the Apriori-like method by reducing search space and generating frequent itemsets without candidate generation. However, the FP-growth method only achieves significant speedups at low minimum supports because the process of constructing and using the FP-trees is complex [14]. In addition, recurrently building conditional itemset bases and trees makes the FP-growth method inefficient when datasets are sparse.

On the one hand, the advantage of the FP-growth method is the FP-tree, which is a highly condensed data structure for storing the database. On the other hand, the biggest advantage of vertical mining algorithm is that itemsets are represented by transaction ids (TID). A question is whether we can integrate the advantages of the two kinds of methods and form a new efficient mining algorithm, which may overcome the shortcomings of the two kinds of methods.

Based on the above analysis, we propose a new mining algorithm, PPV, which combines an FP-tree-like data structure with an intersection-based approach to discover frequent itemsets [6]. PPV is the first work on integrating FP-growth method and vertical method. However, PPV fully employs candidate set generation-and-test strategy to discovery frequent itemsets. Therefore, it has the inherent weakness of Apriori-like methods. That is, PPV becomes inefficient when datasets are dense and frequent itemsets are long [6]. In this paper, we develop a novel algorithm, PrePost, from PPV. The core difference between PrePost and PPV is that PrePost can directly mining frequent itemsets without generating candidates. This makes PrePost overcome the inherent weakness of PPV efficiently.

PrePost adopts a prefix tree structure called PPC-tree to store the database. Each node in a PPC-tree is assigned with a Pre-Post code via traversing the PPC-tree with Pre and Post order. Based on the PPC-tree with Pre-Post code, each frequent item can be represented by an N-list, which is the list of PP-codes that consists of pre-order code, post-order code, and count of nodes registering the frequent item. Like other vertical algorithms, PrePost adopts the Apriori-like approach to find frequent itemsets. That is, it gets N-lists of the candidate itemsets of length $(k+1)$ by intersecting N-lists of frequent itemsets of length k and thus discovers the frequent itemsets of length $(k+1)$. However, PrePost can directly mine frequent itemsets without generating candidates in some cases.

3 Basic principles

3.1 Description of the problem

Let $I = \{i_1, i_2, \ldots, i_m\}$ be the universal item set. Let $DB = \{T_1, T_2, \ldots, T_n\}$ be a transaction database, where each $T_k (1 \leqslant k \leqslant n)$ is a transaction which is a set of items such that $T_k \subseteq I$. We also call A an itemset if A is a set of items. Let A be an itemset. A transaction T is said to contain A if and only if $A \subseteq T$. Let SP_A be the support of itemset A, which is the number of transactions in DB that contain A. Let ξ be the predefined minimum support and $|DB|$ be the number of transactions in DB. An itemset A is frequent if SP_A is no less than $\xi \times |DB|$. Given a transaction database DB and a ξ, the problem of mining frequent itemsets is how to discover the set of all itemsets that have support no less than $\xi \times |DB|$, which is also called minimum threshold.

3.2 PPC-tree definition

Before introducing N-list, we first introduce PPC-tree, which is the basis of N-list. We define a PPC-tree

Table 1 A transaction database

ID	Items	Ordered frequent items
1	a, c, g, f	c, f, a
2	e, a, c, b	b, c, e, a
3	e, c, b, i	b, c, e
4	b, f, h	b, f
5	b, f, e, c, d	b, c, e, f

as follows.

Definition 1. PPC-tree is a tree structure:

1) It consists of one root labeled as "null", and a set of item prefix subtrees as the children of the root.

2) Each node in the item prefix subtree consists of five fields: *item-name*, *count*, *children-list*, *pre-order*, and *post-order*. *item-name* registers which item this node represents. *count* registers the number of transactions presented by the portion of the path reaching this node. *children-list* registers all children of the node. *pre-order* is the pre-order rank of the node. *post-order* is the post-order rank of the node.

According to Definition 1, PPC-tree looks like an FP-tree [1,15]. Note that, FP-tree is called PC-tree in [15]. However, there are three important differences between them.

First, FP-tree has a node-link field in each node and a header table structure to maintain the connection of nodes whose item-names are equal in the tree, where PPC-tree does not have such structures. So, PPC-tree is a simpler prefix tree.

Second, each node in the PPC-tree has *pre-order* and *post-order* fields while nodes in the FP-tree have not. The *pre-order* of a node is determined by a pre-order traversal of the tree. In a pre-order traversal, a node N is visited and assigned the pre-order rank before all its children are traversed recursively from left to right. In other word, the *pre-order* records the time when node N is accessed during the pre-order traversal. In the same way, the *post-order* of a node is determined by a post-order traversal of the tree. In a post-order traversal, a node N is visited and assigned its post-order rank after all its children have been traversed recursively from left to right.

Third, after a FP-tree is built, it will be used for frequent itemset mining during the total process of FP-growth algorithm, which is a recursive and complex process. However, PPC-tree is only used for generating the Pre-Post code of each node. Later, we will find that after collecting the Pre-Post code of each frequent item first, the PPC-tree finishes its entire task and could be deleted.

Based on Definition 1, we have the following PPC-tree construction algorithm.

To get a better understanding of the concept and the construction algorithm of PPC-tree, let's examine the following example.

Example 1. Let the transaction database, DB, be represented by the information from the left two columns of Table 1 and $\xi = 0.4$. The frequent 1-itemsets set $F_1 = \{a, b, c, e, f\}$.

Figure 1 shows the PPC-tree resulting from Example 1 after Algorithm 1 execution. The node with (4, 8) means that its *pre-order* is 4, *post-order* is 8, the *item-name* is b, and *count* is 4. Note that the PPC-tree is constructed using the right most column of Table 1 according to Algorithm 1. Obviously, the second column and the last column are equivalent for mining frequent itemsets under the given minimum support. In the rightmost columns of Table 1, all infrequent items are eliminated and frequent items are listed in support-descending order. This ensures that the DB can be efficiently represented by a compressed tree structure.

3.3 N-list: definitions and properties

In this section, we will give the definition of N-list and introduce some important properties of N-list, which determine the efficiency of our new proposed algorithm for mining frequent itemsets. We first define PP-code, which is the basic component of N-list.

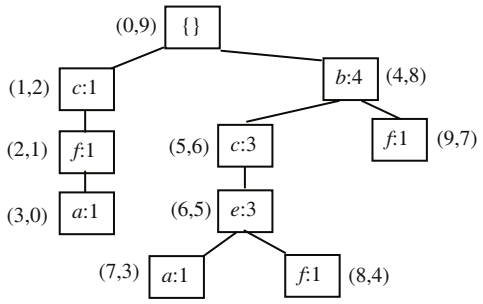

Figure 1 The PPC-tree resulting from Example 1 after running Algorithm 1.

Algorithm 1 (PPC-tree construction)

Input: A transaction database DB and a minimum support ξ.

Output: A PPC-tree and F1 (the set of frequent 1-itemsets).

Method: Construct-PPC-tree(DB,ξ)

1: [Frequent 1-itemsets generation]
2: According to ξ , scan DB once to find F_1, the set of frequent 1-itemsets (frequent items), and their supports.
3: Sort F1 in support descending order as L1, which is the list of ordered frequent items. Note that, if the supports of some frequent items are equal, the orders can be assigned arbitrarily.
4: [PPC-tree construction]
5: Create the root of a PPC-tree, Tr , and label it as "null".
6: **for** each transaction $Trans$ in DB **do**
7: Select the frequent items in $Trans$ and sort out them according to the order of F_1. Let the sorted frequent-item list in $Trans$ be $[p|P]$, where p is the first element and P is the remaining list.
8: Call $insert\_tree([p|P], Tr)$.
9: **end for**
10: [Pre-Post code generation]
11: Scan PPC-tree to generate the *pre-order* and the *post-order* of each node.
12:
13: [Function $insert\_tree([p|P], Tr)$]
14: **if** T_r has a child N such that $N.item\text{-}name = p.item\text{-}name$ **then**
15: increase N's *count* by 1;
16: **else**
17: create a new node N, with its count initialized to 1, and add it to $T_r's$ children-list;
18: **if** P is nonempty **then**
19: call $insert\_tree(P, N)$ recursively.
20: **end if**
21: **end if**

Definition 2 (PP-code)**.** For each node N in a PPC-tree, we call $\langle (N.pre{-}order, N.post\text{-}order) : count \rangle$ the PP-code of N.

In fact, the goal of constructing PPC-tree is to generate the PP-codes of frequent items, since the PP-codes can reflect the structure of the PPC-tree as follows.

Property 1. Given any two different nodes N_1 and N_2, N_1 is an ancestor of N_2, if and only if $N_1.pre\text{-}order < N_2.pre\text{-}order$ and $N_1.post\text{-}order > N_2.post\text{-}order$.

For the proof of Property 1, please refer to [16]. Property 1 also shows that nodes and their PP-code are 1-1 mapping. That is, a node uniquely determines a PP-code and a PP-code also uniquely determines a node. Therefore, we have the following definition.

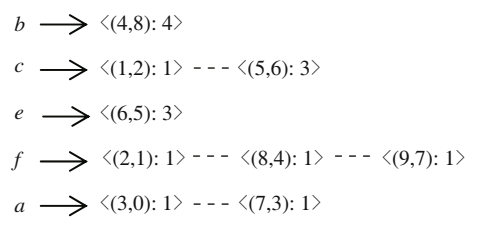

Figure 2 The N-lists of frequent items in Example 1.

Definition 3 (Ancestor-descendant relationship of PP-codes)**.** Given two PP-codes X_1 and X_2, X_1 is an ancestor of X_2 if and only if the node represented by X_1 is an ancestor of the node represented by X_2.

Let X_1 be $\langle(x_1, y_1) : z_1\rangle$ and X_2 be $\langle(x_2, y_2) : z_2\rangle$, Definition 3 is equal to the saying that X_1 is an ancestor of X_2 if and only if $x_1 < x_2$ and $y_l > y_2$. We also call Y a descendant of X if X is an ancestor of Y.

Definition 4 (N-list of frequent item)**.** Given a PPC-tree, the N-list of a frequent item is a sequence of all the PP-codes of nodes registering the item in the PPC-tree. The PP-codes are arranged in an ascending order of their *pre-order* values.

Each PP-code in the N-list is denoted by $\langle(x, y) : z\rangle$, where x is its *pre-order*, y is its *post-order* and z is its *count*. And the N-list of a frequent item is denoted by $\langle(x_1, y_1) : z_1\rangle, \langle(x_2, y_2) : z_2\rangle, \ldots, \langle(x_1, y_l) : z_l\rangle$, where $x_1 < x_2 < \cdots < x_l$. For example, the N-list of f includes three nodes, which are $\langle(2, 1) : 1\rangle$, $\langle(8, 4) : 1\rangle$, and $\langle(9, 7) : 1\rangle$. Figure 2 shows the N-lists of all frequent items in Example 1.

Property 2. Given any two different nodes N_1 and N_2, which represent the same item (N_1.*item-name* = N_2.*item-name*), if N_1.*pre-order* < N_2.*pre-order*, then N_1.*post-order* < N_2.*post-order*.

When N_1.*pre-order* < N_2.*pre-order*, it means that N_1 is traveled earlier than N_2 during the pre-order traversal. Since N_1 can not be an ancestor of N_2 because they both register the same item, N_1 must be on the left branch of PPC-tree compared with N_2. During the post-order traversal, the left branch will also be traversed earlier than N_2, so N_1.*post-order* < N_2.*post-order*.

Given the N-list of item i which is denoted by $\langle(x_1, y_1) : z_1\rangle, \langle(x_2, y_2) : z_2\rangle, \ldots, \langle(x_l, y_l) : z_l\rangle$, since we arrange the PP-code in the accessed order during pre-order traversal, we have $x_1 < x_2 < \cdots < x_l$. According to Property 2, we also have $y_1 < y_2 < \cdots < y_l$. For example, in Figure 2 the N-list of item c is $\langle(1, 2) : 1\rangle, \langle(5, 6) : 3\rangle$ and the N-list of item f is $\langle(2, 1) : 2\rangle, \langle(8, 4) : 1\rangle, \langle(9, 7) : 1\rangle$. They both confirm this property.

Property 3. Given the N-list of item i, which is denoted by $\langle(x_1, y_1) : z_1\rangle, \langle(x_2, y_2) : z_2\rangle, \ldots, \langle(x_m, y_m) : z_m\rangle$, the support of item i is $z_1 + z_2 + \cdots + z_m$.

It is a consequence of the definition of PP-code. Since each PP-code corresponds to a node in PPC-tree, whose *count* registers the number of transactions including item i, the sum of *counts* of nodes registering item i is $i's$ support.

Note that in this paper we denote by L_1 the set of frequent items, in which frequent items are sorted in support descending order. Based on L_1, we define $\succ$ relation of two items as follows.

Definition 5 ($\succ$ relation)**.** For any two frequent items i_1 and i_2 ($i_1, i_2 \in L_1$). $i_1 \succ i_2$ if and only if i_1 is before i_2 in L_1.

For simplicity of description, any itemset P in this paper is denoted by $i_1 i_2 \cdots i_k$, where $i_1 \succ i_2 \succ \cdots \succ i_k$. We define the N-list of a 2-itemset, which only contains two different items, as follows.

Definition 6 (N-list of 2-itemset)**.** Given any two different frequent items i_1 and i_2 (i_1 is before i_2 in L_1), whose N-lists are $\langle(x_{11}, y_{11}) : z_{11}\rangle, \langle(x_{12}, y_{12}) : z_{12}\rangle, \ldots, \langle(x_{1m}, y_{1m}) : z_{1m}\rangle$ and $\{\langle(x_{21}, y_{21}) : z_{21}\rangle, \langle(x_{22}, y_{22}) : z_{22}\rangle, \ldots, \langle(x_{2n}, y_{2n}) : z_{2n}\rangle\}$ respectively. The N-list of 2-itemset $i_1 i_2$ is a sequence of PP-codes according to *pre-order* ascending order and is generated by intersecting the N-lists of i_1 and i_2, which follows the rule below:

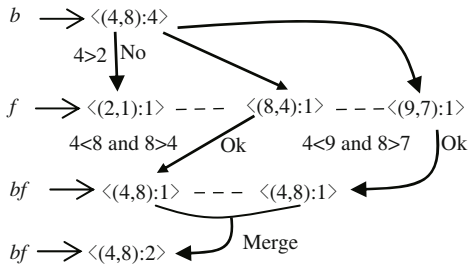

Figure 3 The N-list of bf in Example 1.

1) for any $\langle(x_{1p}, y_{1p}) : z_{1p}\rangle \in$ the N-list of $i_1 (1 \leqslant p \leqslant m)$ and $\langle(x_{2q}, y_{2q}) : z_{2q}\rangle \in$ the N-list of $i_2 (1 \leqslant q \leqslant n)$, if $\langle(x_{1p}, y_{1p}) : z_{1p}\rangle$ is an ancestor of $\langle(x_{2q}, y_{2q}) : z_{2q}\rangle$, then $\langle(x_{1p}, y_{1p}) : z_{2q}\rangle$ is added to the N-list of $i_1 i_2$. After that, we get an initial N-list of $i_1 i_2$.

2) check the initial N-list of $i_1 i_2$ again. Merge the nodes with the form of $\langle(x_{1b}, y_{1b}) : z_{1b}\rangle\langle(x_{1b}, y_{1b}) : z_{2b}\rangle \cdots \langle(x_{1b}, y_{1b}) : z_{rb}\rangle$ to get a new node $\langle(x_{1b}, y_{1b}) : (z_{1b} + z_{2b} \cdots + z_{rb})\rangle$.

As shown in Figure 2, the N-list of b is $\langle(4, 8) : 4\rangle$ and the N-list of f is $\langle(2, 1) : 1\rangle, \langle(8, 4) : 1\rangle, \langle(9, 7) : 1\rangle$. Let's see how to generate the N-list of bf by combining the N-list of b and the N-list of f. According to Definition 6, we should check an ancestor-descendant relationship of $\langle(4, 8) : 4\rangle$ with each PP-code in the N-list of f. Because 4, the *pre-order* of $\langle(4, 8) : 4\rangle$, is greater than 2, the *pre-order* of $\langle(2, 1) : 1\rangle$, $\langle(4, 8) : 4\rangle$ cannot be an ancestor of $\langle(2, 1) : 1\rangle$ in terms of Definition 3. Let's go on comparing $\langle(4, 8) : 4\rangle$ and $\langle(8, 4) : 1\rangle$. Since 4, the *pre-order* of $\langle(4, 8) : 4\rangle$, is less than 8, the *pre-order* of $\langle(8, 4) : 1\rangle$, and 8, the *post-order* of $\langle(4, 8) : 4\rangle$, is greater than 4, the *post-order* of $\langle(8, 4) : 1\rangle$. According to Definition 3, $\langle(4, 8) : 4\rangle$ is an ancestor of $\langle(8, 4) : 1\rangle$. In terms of Definition 6 (first rule), $\langle(4, 8) : 1\rangle$, which is composed of the *pre-order* and *post-order* of $\langle(4, 8) : 4\rangle$ and the *count* of $\langle(8, 4) : 1\rangle$, is added to the N-list of bf. In the same way, we know that $\langle(4, 8) : 4\rangle$ is an ancestor of $\langle(9, 7) : 1\rangle$. So, $\langle(4, 8) : 1\rangle$, which is composed of the *pre-order* and *post-order* of $\langle(4, 8) : 4\rangle$ and the *count* of $\langle(9, 7) : 1\rangle$, is also added to the N-list of bf. By checking the elements in the N-list of bf, we find that the only two elements that have the same *pre-order* and *post-order*, are 4 and 8 respectively. According to Definition 6 (second rule), we should combine $\langle(4, 8) : 1\rangle$ and $\langle(4, 8) : 1\rangle$ to form $\langle(4, 8) : 2\rangle$. So, the N-list of bf is $\langle(4, 8) : 2\rangle$. Because there are no other elements except $\langle(4, 8) : 4\rangle$ in the N-list of b, the processing is stopped. Figure 3 shows the whole processing procedure. In the same way, we know that the N-list of cf is $\langle(1, 2) : 1\rangle, \langle(5, 6) : 1\rangle$.

Based on Definition 6, let us generalize it to the concept of the N-list of a *k-itemset*$(k \geqslant 3)$.

Definition 7 (N-list of k-itemset)**.** Let $P = i_x i_y i_1 i_2 \ldots i_{(k-2)}$ be a itemset ($k \geqslant 3$ and each item in P is a frequent item), the N-list of $P_1 = i_x i_1 i_2 \ldots i_{(k-2)}$ be $\langle(x_{11}, y_{11}) : z_{11}\rangle, \langle(x_{12}, y_{12}) : z_{12}\rangle, \ldots,$ $\langle(x_{1m}, y_{1m}) : z_{1m}\rangle$, and the N-list of $P_2 = i_y i_1 i_2 i_{(k-2)}$ be $\langle(x_{21}, y_{21}) : z_{21}\rangle, \langle(x_{22}, y_{22}) : z_{22}\rangle, \ldots, \langle(x_{2n},$ $y_{2n}) : z_{2n}\rangle$. The N-list of P is a sequence of PP-codes according to *pre-order* ascending order and generated by intersecting the N-lists of P_1 and P_2, which follows the rule below:

1) for any $\langle(x_{1p}, y_{1p}) : z_{1p}\rangle \in$ the N-list of $P_1 (1 \leqslant p \leqslant m)$ and $\langle(x_{2q}, y_{2q}) : z_{2q}\rangle \in$ the N-list of $P_2 (1 \leqslant q \leqslant n)$, if $\langle(x_{1p}, y_{1p}) : z_{1p}\rangle$ is an ancestor of $\langle(x_{2q}, y_{2q}) : z_{2q}\rangle$, then $\langle(x_{1p}, y_{1p}) : z_{2q}\rangle$ is added to the N-list of $i_x i_y i_1 i_2 \ldots i_{(k-2)}$. After that, we get an initial N-list of P.

2) we check the initial N-list of P again and merge the nodes with the form of $\langle(x_{1b}, y_{1b}) : z_{1b}\rangle\langle(x_{1b}, y_{1b}) : z_{2b}\rangle \cdots \langle(x_{1b}, y_{1b}) : z_{rb}\rangle$ to get a new node $\langle(x_{1b}, y_{1b}) : (z_{1b} + z_{2b} \cdots + z_{rb})\rangle$.

For example, we have known the N-list of bf and cf are $\langle(4, 8) : 2\rangle$ and $\langle(1, 2) : 1\rangle, \langle(5, 6) : 1\rangle$. According to Definition 7, the N-list of bcf can be built as follows. Fist, we compare $\langle(4, 8) : 2\rangle$ with $\langle(1, 2) : 1\rangle$. We find that $\langle(4, 8) : 2\rangle$ cannot be an ancestor of $\langle(1, 2) : 1\rangle$ because 4, the *pre-order* of the former, is larger than 1, the *pre-order* of the latter. Then, we go on comparing $\langle(4, 8) : 2\rangle$ with $\langle(5, 6) : 1\rangle$. Obviously, $\langle(4, 8) : 2\rangle$ is an ancestor of $\langle(5, 6) : 1\rangle$ because 4, the *pre-order* of the former, is less than 5, the *pre-order* of the latter, and 8, the *post-order* of the former, is bigger than 6, the *post-order* of the latter. So, $\langle(4, 8) : 1\rangle$ are added to the N-list of bcf. Because there are no other elements except $\langle(4, 8) : 2\rangle$ in the N-list of bf, the processing is stopped. Finally, we know that the N-list of bcf is $\langle(4, 8) : 1\rangle$. Figure 4 shows the whole processing procedure.

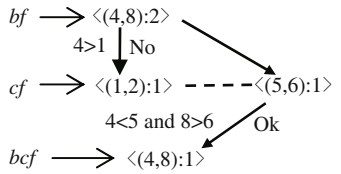

Figure 4 The N-list of bcf in Example 1.

Based on Definitions 4, 6, and 7, we have property as follow.

Property 4. Let $\langle (x, y) : z\rangle$ be a PP-code in the N-list of k-itemset $i_1 i_2 \cdots i_k$. We have the following conclusions:

1) There must be a node registering i_1 having the PP-code with the same (x, y);

2) z is the count of all the k-itemset $i_1 i_2 \cdots i_k$ in the sub-tree with the corresponding node registering i_1 as the root.

Proof. We will prove this property by mathematical induction.

$k = 1$. According to Definition 4, we know that each PP-code in the N-list of any frequent item i represents a node registering i. Therefore, Property 4 is true for $k = 1$.

$k = 2$. On the one hand, according to Definition 6, for any PP-code in the N-list of $i_1 i_2$, there must be a PP-code in the N-list of i_1 with the same (x, y). And according to Definition 4, every PP-code in the N-list of i_1 corresponds to a node registering i_1. So there must be a node registering i_1 having the PP-code with the same (x, y). So Conclusion 1) is right. On the other hand, from the merging process of Definition 6, we can clearly find that z of each PP-code in the N-list of $i_1 i_2$ is the sum of the *count* of all the itemsets $i_1 i_2$ in the sub-tree with the corresponding node registering i_1 as the root. Then, we know that Conclusion 2) is also right. Therefore, Property 4 holds for $k = 2$.

$k = 3$. The N-list of 3-itemset $i_1 i_2 i_3$ is generated by the N-list of $i_1 i_3$ and $i_2 i_3$. As with the case of $k = 2$, on the one hand, according to Definition 7, for any PP-code in the N-list of $i_1 i_2 i_3$, there must be a PP-code in the N-list of $i_1 i_3$ with the same (x, y). And according to the case $k = 2$, for any PP-code in the N-list of $i_1 i_3$, there must be node registering i_1 having the PP-code with the same (x, y). So Conclusion 1) is true. On the other hand, from the merging process of Definition 7, we can clearly find that z of each PP-code in the N-list of $i_1 i_2 i_3$ is the sum of the *count* of all the itemsets $i_1 i_2 i_3$ in the sub-tree with the corresponding node registering i_1 as the root. Then, we know that Conclusion 2) also holds true. Therefore, Property 4 is true for $k = 3$.

For $k > 3$, the rationale is the same as $k = 3$. Therefore, we have Property 4.

Based on Property 4, we can easily conclude the following.

Property 5. Given an N-list of any k-itemset $P = i_1 i_2 \cdots i_k$, which is denoted by $\{\langle (x_1, y_1) : z_1\rangle, \langle (x_2, y_2) : z_2\rangle, \ldots, \langle (x_m, y_m) : z_m\rangle\}$, the support of itemset P is $z_1 + z_2 + \cdots + z_m$.

Property 6. Let $P = i_1 i_2 \cdots i_k$ be a k-itemset and the N-list of P be $\langle (x_1, y_1) : z_1\rangle, \langle (x_2, y_2) : z_2\rangle, \ldots, \langle (x_m, y_m) : z_m\rangle$. Then we have $x_1 < x_2 < \cdots < x_m$ and $y_1 < y_2 < \cdots < y_m$.

Proof. In terms of Definitions 4, 6 and 7, The N-list of P is a sequence of PP-codes according to *pre-order* ascending order. Therefore, we have $x_1 < x_2 < \cdots < x_m$. In addition, each $\langle (x_j, y_j) : z_j\rangle$ corresponds to a node with *item-name* $= i_1$ according to Property 4. By Property 2, we have $y_1 < y_2 < \cdots < y_m$.

4 Mining frequent itemsets using N-list

In this section, we briefly describe the main idea and steps of PrePost, which is the algorithm that we propose for mining frequent itemsets using N-lists. PrePost adopts Apriori-like approach for mining frequent itemsets. According the processing sequence, the main steps involved in the PrePost algorithm are: 1) Construct PPC-tree and identify all frequent 1-itemsets; 2) based on PPC-tree, construct the N-list of each frequent 1-itemset; 3) scan PPC-tree to find all frequent 2-itemsets; 4) mine all frequent

Algorithm 2 N-lists construction

Input: PPC-tree and L_1, the set of frequent 1-itemsets.

Output: NL_1, the set of the N-lists of frequent 1-itemsets.

Procedure N-lists_construction (PPC-tree)

1: Create NL_1, let $NL_1[k]$ be the N-list of $L_1[k]$.
2: **for** each node N of PPC-tree accessed by *pre-order* traversal **do**
3: **if** $(N.item-name = L_1[k].item-name)$ **then**
4: insert $\langle(N.\text{pre-order}, N.\text{post-order}) : N.count\rangle$ into $NL_1[k]$
5: **end if**
6: **end for**

$k(>2)$-*itemsets*. For step 1), Algorithm 1 in Subsection 3.2 shows its details. For the other steps, we will present their details in following sections.

4.1 Building N-lists of frequent 1-itemsets

Given transaction database DB and minimum support ξ, once we build the PPC-tree, it is easy to obtain the N-list of each frequent 1-itemset. By traversing the PPC-tree with *pre-order*, we can access each node in PPC-tree. For each node N, we insert $\langle(N.pre\text{-}order, N.post\text{-}order) : N.count\rangle$ into the N-list of the item registered by N. Algorithm 2 shows the details of how to construct the N-lists of all frequent 1-itemsets.

4.2 Mining frequent 2-itemsets

Many studies [5,10] reveal that the process of mining frequent 2-itemsets is high-cost for Apriori-based algorithms because the number of candidate 2-itemsets is usually huge, so it will be time-consuming to find frequent 2-itemsets by joining frequent 1-itemsets. Ref. [5] proposed a typical way to find all frequent 2-itemsets without joining frequent 1-itemsets. The method goes as follows. First, for each transaction, we get all its 2-itemsets (subset). Then, we can get the support of each 2-itemset after we deal with all transactions. Finally, it's easy to find frequent 2-itemsets when the support of each 2-itemset is known.

However, we think out a better method, which can find out all frequent 2-itemsets by traversing the PPC-tree. Our method finds frequent 2-itemsets as follows. By traversing the PPC-tree with pre-order, we can access every node of the PPC-tree. For each node N, letting N_a be one of its ancestors, we increase the count of 2-itemset $N.item\text{-}name \cup Na.item\text{-}name$ by $N.count$. After scanning the PPC-tree, we get all 2-itemsets containing in the PPC-tree and their support (their final counts). In finding frequent 2-patters, we just need to check whether the supports of these 2-itemsets are not less than $\xi \times |DB|$. Because the PPC-tree is more compressed than the corresponding transaction database and our method does not need to generate all 2-itemsets in each transaction, our method is more efficient than the method in [5]. Algorithm 3 shows the details of our method. To raise efficiency, we implement Algorithms 2 and 3 in the same pre-order traversal.

4.3 A linear-time-complexity method for intersecting N-lists

PrePost finds frequent $(k+1)$-itemsets by intersecting the N-lists of frequent k-itemsets. It is obvious that the efficiency of intersecting two N-lists is vital to the efficiency of PrePost. Before giving our intersecting method, let's first examine the following example.

Let $P_1 = i_u i_1 i_2 \cdots i_{(k-2)}$ and $P_2 = i_v i_1 i_2 \cdots i_{(k-2)} (i_u \succ i_v)$ be two $(k-1)$-itemsets. The N-list of P_1 is $\{\langle(x_{11}, y_{11}) : z_{11}\rangle, \langle(x_{12}, y_{12}) : z_{12}\rangle, \ldots, \langle(x_{1m}, y_{1m}) : z_{1m}\rangle\}$. The N-list of P_2 is $\{\langle(x_{21}, y_{21}) : z_{21}\rangle$, $\langle(x_{22}, y_{22}) : z_{22}\rangle, \ldots, \langle(x_{2n}, y_{2n}) : z_{2n}\rangle\}$. To generate the N-list of $P = i_u i_v i_1 \cdots i_{(k-2)}$, a naïve method is by comparing each PP-code of the N-list of P_1 with each PP-code of the N-list of P_2 to decide whether they satisfy an ancestor-descendant relationship. It is obvious that the time complexity of the naïve method is $O(mn)$. This time complexity is unsatisfying. After some careful analysis, we find a linear-time-complexity method, which is based on the following Lemma.

Algorithm 3 mining frequent 2-itemsets

Input: PPC-tree and L_1, the set of all frequent 1-itemsets.

Output: L_2, the set of all frequent 2-itemsets.

Procedure *L_2_Construction* (PPC-tree)

```
1: Let L1[k] be the kth element in L1, set L1[k].order = k.
2: Create Temp2 = int[L1.size()][L1.size()].
3: for each node N of PPC-tree accessed by pre-order traversal do
4:    for each ancestor node of N, Let it be Na. do
5:       Temp2[N.item − name.order][Na.item − name.order]+ = N.count;
6:    end for
7: end for
8: for each element Temp2[i, j] in Temp2 do
9:    if Temp2[i, j] ⩾ ξ × |DB| then
10:      insert L1[i] ∪ L1[j] into L2
11:   end if
12: end for
```

Lemma 1. Let $P_1 = i_u i_1 i_2 \cdots i_{(k-2)}$ and $P_2 = i_v i_1 i_2 \cdots i_{(k-2)} (i_u \succ i_v)$ be two $(k-1)$-itemsets. The N-list of P_1 is $\{\langle (x_{11}, y_{11}) : z_{11}\rangle, \langle (x_{12}, y_{12}) : z_{12}\rangle, \ldots, \langle (x_{1m}, y_{1m}) : z_{1m}\rangle\}$. The N-list of P_2 is $\{\langle (x_{21}, y_{21}) : z_{21}\rangle, \langle (x_{22}, y_{22}) : z_{22}\rangle, \ldots, \langle (x_{2n}, y_{2n}) : z_{2n}\rangle\}$. If $\exists \langle (x_{1s}, y_{1s}) : z_{1s}\rangle \in P_1$ and $\langle (x_{2t}, y_{2t}) : z_{2t}\rangle \in P_2, \langle (x_{1s}, y_{1s}) : z_{1s}\rangle$ is an ancestor of $\langle (x_{2t}, y_{2t}) : z_{2t}\rangle$, then any $\langle (x_{1k}, y_{1k}) : z_{1k}\rangle \in P_1 (k \neq s)$ cannot be an ancestor of $\langle (x_{2t}, y_{2t}) : z_{2t}\rangle$.

Proof. Let $\langle (x_{1s}, y_{1s}) : z_{1s}\rangle$ be an ancestor of $\langle (x_{2t}, y_{2t}) : z_{2t}\rangle$, N_1 be the node represented by $\langle (x_{1s}, y_{1s}) : z_{1s}\rangle$, N_2 be the node represented by $\langle (x_{2t}, y_{2t}) : z_{2t}\rangle$, and N be the node represented by $\langle (x_{1k}, y_{1k}) : z_{1k}\rangle (k \neq s)$. If N is an ancestor of N_2, then N_1 and N must have an ancestor-descendant relationship. According to Property 4, the item-names of N_1 and N are both i_u. But, by the construction of PPC-tree, nodes with the same *item-name* cannot have an ancestor-descendant relationship. End of proof.

Based on Property 6 and Lemma 1, the generation of the N-list of $P = i_u i_v i_1 i_2 \cdots i_{(k-2)}$ can be efficiently implemented by a linear-time-complexity Method called *NL_intersection*. The *NL_intersection* method first selects a PP-code from $\{\langle (x_{11}, y_{11}) : z_{11}\rangle, \langle (x_{12}, y_{12}) : z_{12}\rangle, \ldots, \langle (x_{1m}, y_{1m}) : z_{1m}\rangle\}$ according to the order from left to right. Then, it checks an ancestor-descendant relationship of the PP-code and PP-codes in $\{\langle (x_{21}, y_{21}) : z_{21}\rangle, \langle (x_{22}, y_{22}) : z_{22}\rangle, \ldots, \langle (x_{2n}, y_{2n}) : z_{2n}\rangle\}$. In a word, our method makes use of the characteristic that PP-codes in an N-list are ordinal.

Let $\langle (x_{1i}, y_{1i}) : z_{1i}\rangle$ and $\langle (x_{2j}, y_{2j}) : z_{2j}\rangle$ be the current PP-codes to be proceeded. The detailed procedures of *NL_intersection* method are as follows:

1) Check an ancestor-descendant relationship of $\langle (x_{1i}, y_{1i}) : z_{1i}\rangle$ and $\langle (x_{2j}, y_{2j}) : z_{2j}\rangle$.

2) If $\langle (x_{1i}, y_{1i}) : z_{1i}\rangle$ is an ancestor of $\langle (x_{2j}, y_{2j}) : z_{2j}\rangle$, then check whether there exists a node with the form of $\langle (x_{1i}, y_{1i}) : z_{\mathrm{m}j}\rangle$ in the N-list of P: if so, change $\langle (x_{1i}, y_{1i}) : z_{\mathrm{m}j}\rangle$ to $\langle (x_{1i}, y_{1i}) : z_{\mathrm{m}j} + z_{2j}\rangle$, otherwise insert $\langle (x_{1i}, y_{1i}) : z_{2j}\rangle$ into the N-list of P. After that, if $\langle \big(x_{2(j+1)}, y_{2(j+1)}\big) : z_{2(j+1)}\rangle$ is no null, then go to 1) to check an ancestor-descendant relationship of $\langle (x_{1i}, y_{1i}) : z_{1i}\rangle$ and $\langle \big(x_{2(j+1)}, y_{2(j+1)}\big) : z_{2(j+1)}\rangle$; else stop.

3) If $\langle (x_{1i}, y_{1i}) : z_{1i}\rangle$ is not an ancestor of $\langle (x_{2j}, y_{2j}) : z_{2j}\rangle$, there would be two cases: $x_{1i} > x_{2j}$ or $x_{1i} < x_{2j} \wedge y_{1i} < y_{2j}$. x_{1i} cannot be equal to x_{2j} because they are the *pre-order* ranks of different nodes. Similarly, y_{1i} cannot be equal to y_{2j}.

(3.1) For $x_{1i} > x_{2j}$, if $\langle \big(x_{2(j+1)}, y_{2(j+1)}\big) : z_{2(j+1)}\rangle$ is no null, then go to 1) to check an ancestor-descendant relationship of $\langle (x_{1i}, y_{1i}) : z_{1i}\rangle$ and $\langle \big(x_{2(j+1)}, y_{2(j+1)}\big) : z_{2(j+1)}\rangle$; otherwise stop.

(3.2) If $x_{1i} < x_{2j} \wedge y_{1i} < y_{2j}$, if $\langle \big(x_{1(i+1)}, y_{1(i+1)}\big) : z_{1(i+1)}\rangle$ is no null, then go to 1) to check an ancestor-descendant relationship of $\langle \big(x_{1(i+1)}, y_{1(i+1)}\big) : z_{1(i+1)}\rangle$ and $\langle (x_{2j}, y_{2j}) : z_{2j}\rangle$; otherwise stop.

The rationale of step (3.1) can be explained as follows. According to Definition 6, for any $v(< j)$, we have $x_{2v} < x_{2j}$. Obviously, $\langle (x_{1i}, y_{1i}) : z_{1i}\rangle$ cannot be the ancestor of $\langle (x_{2v}, y_{2v}) : z_{2v}\rangle$. So, we need only to check those x_{2u} with $u > j$ to judge whether it is greater than x_{1i}. The rationale of step (3.2) can be explained as follows. According to Property 6, we have $y_{2j} < y_{2t}$ for $j < t$. Because of $y_{1i} < y_{2j}$ in

step (3.2), we have $y_{1i} < y_{2t}$. That is, $\langle(x_{1i}, y_{1i}) : z_{1i}\rangle$ cannot be an ancestor of $\langle(x_{2t}, y_{2t}) : z_{2t}\rangle$. So, we do not need to check an ancestor-descendant relationship of $\langle(x_{1i}, y_{1i}) : z_{1i}\rangle$ and $\langle(x_{2t}, y_{2t}) : z_{2t}\rangle$, which means $\langle(x_{1i}, y_{1i}) : z_{1i}\rangle$ need not be processed any more. So, $\langle\big(x_{1(i+1)}, y_{1(i+1)}\big) : z_{1(i+1)}\rangle$, the next PP-code of $\langle(x_{1i}, y_{1i}) : z_{1i}\rangle$, should be selected as the next proceeded PP-code to check an ancestor-descendant relationship with the PP-codes from the N-list of P_2. For any $\langle(x_{2k}, y_{2k}) : z_{2k}\rangle (k < j)$, there are two cases:

Case 1: there exists $\langle(x_{1v}, y_{1v}) : z_{1v}\rangle (1 \leqslant v \leqslant i)$ that is an ancestor of $\langle(x_{2k}, y_{2k}) : z_{2k}\rangle$;

Case 2: $\langle(x_{2k}, y_{2k}) : z_{2k}\rangle$ cannot be a descendant of $\langle(x_{1v}, y_{1v}) : z_{1v}\rangle$ for any $v (1 \leqslant v \leqslant i)$.

For case 1, $\langle(x_{2k}, y_{2k}) : z_{2k}\rangle$ cannot be a descendant of $\langle\big(x_{1(i+1)}, y_{1(i+1)}\big) : z_{1(i+1)}\rangle$ according to Lemma 1.

For case 2, suppose $x_{1i} < x_{2k}$. We have $y_{1i} < y_{2k}$ because $\langle(x_{1i}, y_{1i}) : z_{1i}\rangle$ is not an ancestor of $\langle(x_{2k}, y_{2k}) : z_{2k}\rangle$. According to the above procedure, an ancestor-descendant relationship of $\langle(x_{1i}, y_{1i}) : z_{1i}\rangle$ and $\langle(x_{2u}, y_{2u}) : z_{2u}\rangle$ has not been checked for any $u (u > k)$, which conflicts with the fact that we are checking an ancestor-descendant relationship of $\langle(x_{1i}, y_{1i}) : z_{1i}\rangle$ and $\langle(x_{2j}, y_{2j}) : z_{2j}\rangle$. So we have $x_{1i} > x_{2k}$. According to Property 6, we have $x_{1(i+1)} > x_{1i}$. So we have $x_{1(i+1)} > x_{2k}$, which means that $\langle(x_{2k}, y_{2k}) : z_{2k}\rangle$ cannot be a descendant of $\langle\big(x_{1(i+1)}, y_{1(i+1)}\big) : z_{1(i+1)}\rangle$. That is, we do not need to check an ancestor-descendant relationship of $\langle\big(x_{1(i+1)}, y_{1(i+1)}\big) : z_{1(i+1)}\rangle$ and $\langle(x_{2k}, y_{2k}) : z_{2k}\rangle$. Therefore, we should go to 1) to check an ancestor-descendant relationship of $\langle\big(x_{1(i+1)}, y_{1(i+1)}\big) : z_{1(i+1)}\rangle$ and $\langle(x_{2j}, y_{2j}) : z_{2j}\rangle$.

To get a better understanding our method, let's examine an example. As shown in Figure 2, the N-list of f is $\langle(2,1):1\rangle, \langle(8,4):1\rangle, \langle(9,7):1\rangle$ and the N-list of a is $\langle(3,0):1\rangle, \langle(7,3):1\rangle$. Let's see the procedures that our method generates the N-list of fa.

We first check an ancestor-descendant relationship of $\langle(2,1):1\rangle$ and $\langle(3,0):1\rangle$. Because of $2 < 3$ and $1 > 0$, we know that $\langle(2,1):1\rangle$ is an ancestor of $\langle(3,0):1\rangle$. So, $\langle(2,1):1\rangle$ are added to the N-list of fa. Then, we check an ancestor-descendant relationship of $\langle(2,1):1\rangle$ and $\langle(7,3):1\rangle$, the next element of $\langle(3,0):1\rangle$. We know $2 < 7$ and $1 < 3$, which satisfy the condition of (3.2). So, we go on checking an ancestor-descendant relationship of $\langle(8,4):1\rangle$, the next element of $\langle(2,1):1\rangle$, and $\langle(7,3):1\rangle$. Because of $8 > 7$, we should check an ancestor-descendant relationship of $\langle(8,4):1\rangle$ and the next element of $\langle(7,3):1\rangle$ according to (3.1). However, there is no element after $\langle(7,3):1\rangle$ in the N-list of a. So, the processing is stopped. Finally, we conclude that the N-list of fa is $\langle(2,1):1\rangle$. Let a comparison of two elements in N-lists be a computing unit. For the above example, our method needs 3 computing units while the naïve method needs $2 \times 3 (= 6)$ computing units. In fact, the time complexity of *NL_intersection* is linear. For the sake of discussion, we define a computing unit as a comparison for checking an ancestor-descendant relationship of two elements in N-lists as the above paragraph. We have the following conclusion.

Lemma 2. Let P_1 and P_2 be two $(k-1)$-itemsets as mentioned in Lemma 1. The time complexity of *NL_intersection* for intersecting the N-lists of P_1 and P_2 is $O(m+n)$.

Proof. Let's regard the elements in the N-list of P_1 and P_2 as nodes in a graph. If E_1, an element in the N-list of P_1, and E_2, an element in the N-list of P_2, have been confirmed to have an ancestor-descendant relationship, we add an edge between E_1 and E_2. Adopting the above method, we generate a graph G after finishing the intersection of the N-list of P_1 and P_2 according to *NL_intersection*. G consists of a tree, T_G, and some isolated nodes, which have no edge connecting other nodes. The number of the edges in T_G stands for the number of all comparisons of elements. The number of nodes in G is $m+n$, which is the number of all elements in the N-list of P_1 and P_2. It is clear that the number of nodes in T_G is no more than that in G. As we know, every n-node tree has exactly $n-1$ edges [17]. So, the number of edges in T_G is no more than $m+n-1$. That is, the number of all comparisons is no more than $m+n-1$. End of proof.

From the above proof, we know that $O(m+n)$ is actually the worst-case time complexity of *NL_intersection*. For a better understanding of our proof, we illustrate the processing procedure of intersecting the N-list of f and a by Figure 5, where steps 1 to 3 stand for the processing sequence of *NL_intersection*.

Based on the above analysis, we have the following Algorithm 4.

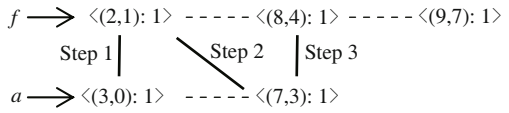

Figure 5 The graph corresponding to the processing of intersecting the N-list of f and a according to $NL\_intersection$.

Algorithm 4 NL_intersection

Input: $NL_1 = \{\langle(x_{11}, y_{11}) : z_{11}\rangle, \langle(x_{12}, y_{12}) : z_{12}\rangle, \ldots, \langle(x_{1m}, y_{1m}) : z_{1m}\rangle\}$ and $NL_2 = \{\langle(x_{21}, y_{21}) : z_{21}\rangle, \langle(x_{22}, y_{22}) : z_{22}\rangle, \ldots, \langle(x_{2n}, y_{2n}) : z_{2n}\rangle\}$, which are the N-list of $P_1 = i_u i_1 i_2 \cdots i_{(k-2)}$ and $P_2 = i_v i_1 i_2 \cdots i_{(k-2)}(i_u \succ i_v)$ respectively.

Output: NL_3, the N-list $P_3 = i_u i_v i_1 i_2 \cdots i_{(k-2)}$.

Procedure $NL\_intersection(NL_1, NL_2)$ (PPC-tree)

```
1: i ← 1;
2: j ← 1;
3: while i ⩽ m&&j ⩽ n do
4:    if (x_{1i} < x_{2j}) then
5:       if (y_{1i} > y_{2j}) then
6:          insert ⟨(x_{1i}, y_{1i}) : z_{2j}⟩ into NL_3;
7:          j++;
8:       else
9:          i++;
10:      end if
11:   else
12:      j++;
13:   end if
14: end while
15: ptr_1 ← NL_3.first_element; ////the first element of NL_3
16: ptr_2 ← ptr_1.next_element; ////the next element of ptr_1
17: while ptr_1 is not the last element of NL_3 do
18:   if ptr_1.pre − code = ptr_2.pre − code and ptr_1.post − code = ptr_2.post − code then
19:      ptr_1.count ← ptr_1.count + ptr_2.count;
20:      delete ptr_2 from NL_3;
21:      ptr_2 ← ptr_1.next_element;
22:   else
23:      ptr_1 ← ptr_2;
24:      ptr_2 ← ptr_1.next_element;
25:   end if
26: end while
```

4.4 Mining frequent k-itemsets

Before presenting the method for mining frequent k-itemsets, we provide a very import property on mining frequent itemsets when some N-lists contain only one element.

Lemma 3 (Single path property)**.** Let $P_1 = j_1 i_1 i_2 \cdots i_v, P_2 = j_2 i_1 i_2 \cdots i_v, \ldots, P_n = j_n i_1 i_2 \cdots i_v, (j_1 \succ j_2 \succ \cdots \succ j_n \succ i_1 \succ i_2 \succ \cdots \succ i_v)$ be itemsets and denote their intersections by $P_a \cup P_b = j_a j_b i_1 i_2 \cdots i_v (1 \leqslant a < b \leqslant n)$. If the N-list of P_n has only one PP-code, and there is a set $S = \{s_1, s_2, \ldots, s_t\}(1 \leqslant s_1 < s_2 < \cdots < s_t < n)$ such that the N-list of each $P_m \cup P_n (m \in S)$ is not null, then

1) For each $m \in S$, the N-list of $P_m \cup P_n$ has only one PP-code and the support of $P_m \cup P_n$ is C_n, the support of P_n.

2) Let Nd_m be the only node in the PPC-tree that represents $P_m \cup P_n$. Then there must be $Nd_{s1} <_{(a-d-r)} Nd_{s2} \rangle_{(a-d-r)} \cdots <_{(a-d-r)} Nd_{st}$, while $X <_{(a-d-r)} Y$ means that X is an ancestor of Y.

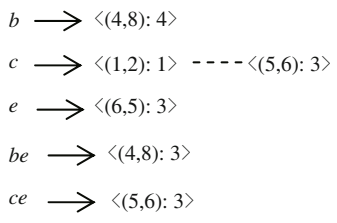

Figure 6 An Example illustrating Corollary 1.

Proof. Lemma 3 is easy to be proved by Definition 7. Limited by space, we just give a brief proof. For $m \in S$, the N-list of $j_m j_n i_1 i_2 \cdots i_v$ is the result of intersecting the N-lists of $P_m\,(= j_m i_1 i_2 \cdots i_v)$ and $P_n\,(= j_n i_1 i_2 \cdots i_v)$. According to the precondition of Lemma 3, the N-list of $P_m \cup P_n$ is not null. According to Definition 7, the N-list of $P_m \cup P_n$ has only one PP-code and the count of this PP-code is C_n. Obviously the support of $P_m \cup P_n$ is C_n. Therefore, we have conclusion 1).

Let $Nd_{sk}(1 \leqslant k \leqslant t)$ be the only node in the PPC-tree that represents $P_{sk} \cup P_n$ and Let Nd_n be the only node in the PPC-tree that represents P_n. According to Definition 7, Nd_{xk} must be an ancestor of Nd_n. So, any two nodes of $\{Nd_{s1}, Nd_{s2}, \ldots, Nd_{st}\}$ must have an ancestor-descendant relation. At the same time, the registering item $(item - name)$ of Nd_{sk} is j_{sk} in terms of Definition 7. According to the precondition of Lemma 3, $j_{s1} \succ j_{s2} \succ \cdots \succ j_{st}$. According to the PPC-tree construction algorithm, $Nd_{s1} <_{(a-d-r)} Nd_{s2} <_{(a-d-r)} \cdots <_{(a-d-r)} Nd_{st}$. End of proof.

In fact, Lemma 3 is similar to Lemma 3.2 (Single FP-tree path itemset generation) of [1]. Therefore, Lemma 3 is also called single path property. For a better understanding of Lemma 3, we offer an example. As shown in Figure 2, the N-list of b, c, and e are $\langle(4,8):4\rangle$, $\langle(1,2):1\rangle$, $\langle(5,6\rangle:3\rangle)$, and $\langle(6,5):3\rangle$ respectively. We know $b \succ c \succ e$ in terms of the definition of L_1. In addition, e have only one PP-code, which is $\langle(6,5):3\rangle$. According to Definition 6, the N-list of be is $\langle(4,8):3\rangle$, and the N-list of ce is $\langle(5,6):3\rangle$. Both the support of be and the support of ce are equal to 3, which is the support of e. Let Nd_{be} denoted by (4,8) be the only node in the PPC-tree that represents be, and Nd_{ce} denoted by (5,6) be the only node in the PPC-tree that represents ce. Nd_{be} is one ancestor of Nd_{ce} because of $4 < 5$ and $8 > 6$, as is evident in Figure 1. Based on Lemma 3, we can easily derive an important corollary, which is useful for efficiently reducing candidate generation in the mining procedure.

Corollary 1. Suppose that $P_1, P_2, \ldots,$ and P_n satisfy the conditions of Lemma 3. Let $S = \{s_1, s_2, \ldots, s_t\}$ $(1 \leqslant s_1 < s_2 < \cdots < s_t < n)$ be the set such that the N-list of each $P_m \cup P_n (m \in S)$ is not null. Then the support of any $j_{sx1} j_{sx2} \ldots j_{sxy} j_n i_1 i_2 \cdots i_v (s_1 \leqslant s_{x1} < s_{x2} < \cdots < s_{xy} \leqslant s_t$, and $s_{xk} \in S$ for $1 \leqslant k \leqslant y)$ is equal to C_n, the support of P_n.

Proof. For $y = 1$. According to conclusion 1) of Lemma 3, for $s_v \in S$, we know that $P_{sv} \cup P_n = j_{sv} j_n i_1 i_2 \cdots i_v$ has only one PP-code and the support of $P_{sv} \cup P_n$ is C_n. Without loss of generality, let $Nd_{sk} = \{\langle(Pre\_s_k, Post\_s_k) : C_n\rangle\}$ be the N-list of $P_{sv} \cup P_n$. That is, Corollary 1 is correct for $y = 1$.

For $y = 2$. According to Definition 7, the N-list of $j_{sx1} j_{sx2} j_{sx3} j_n i_1 i_2 \cdots i_v (s_1 \leqslant s_{x1} < s_{x2} \leqslant s_t)$ is the result of intersecting $\{\langle(Pre\_s_{x1}, Post\_s_{x1}) : C_n\rangle\}$ and $\{\langle(Pre\_s_{x2}, Post\_s_{x2}) : C_n\rangle\}$. According to conclusion 2) of Lemma 3, the node $(Pre\_s_{x1}, Post\_s_{x1})$ must be an ancestor of the node $(Pre\_s_{x2}, Post\_s_{x2})$ in the PPC-tree. That is, we have $Pre\_s_{x1} < Pre\_s_{x2}$ and $Post\_s_{x1} > Post\_s_{x2}$. So, the N-list of $j_{sx1} j_{sx2} j_n i_1 i_2 \cdots i_v$ is $\{\langle(Pre\_s_{x1}, Post\_s_{x1}) : C_n\rangle\}$. That is, Corollary 1 is correct for $y = 2$.

For $y = 3$. According to Definition 7, the N-list of $j_{sx1} j_{sx2} j_{sx3} j_n i_1 i_2 \cdots i_v (s_1 \leqslant s_{x1} < s_{x2} < s_{x3} \leqslant s_t)$ is the result of intersecting $\{\langle(Pre\_s_{x1}, Post\_s_{x1}) : C_n\rangle\}$, the N-list of $j_{sx1} j_{sx3} j_n i_1 i_2 \cdots i_v$, and $\{\langle(Pre\_s_{x2}, Post\_s_{x2}) : C_n\rangle\}$, the N-list of $j_{sx2} j_{sx3} j_n i_1 i_2 \cdots i_v$. Similarly, we have $Pre\_s_{x1} < Pre\_s_{x2}$ and $Post\_s_{x1} > Post\_s_{x2}$. So, the N-list of $j_{sx1} j_{sx2} j_{sx3} j_n i_1 i_2 \cdots i_v$ is $\{\langle(Pre\_s_{x1}, Post\_s_{x1}) : C_n\rangle\}$. That is, Corollary 1 is correct for $y = 3$.

In the same way, we can prove that Corollary 1 is correct for $y > 3$ by repeating the above processing procedures. Thus, Corollary 1 is confirmed. End of proof.

If P_n is frequent, $j_{sx1} j_{sx2} \cdots j_{sxy} j_n i_1 i_2 \cdots i_v$ must be also frequent. Let's re-examine the above example. We have the results as shown in Figure 6. Itemset e has only one PP-code, which is $\langle(6,5):3\rangle$. Both

Algorithm 5 Mining frequent k-itemsets

Input: the minimum support ξ, the frequent 1-itemsets L_1 and their N-lists NL_1. Note that frequent 1-itemsets in L_1 are sorted in support descending order.

Output: The frequent itemset set F.

Method: call $mining\_L(L_1, NL_1)$

Procedure $mining\_L(L_k, NL_k)$

```
1: for i ← L_k.size() − 1 to 1 do
2:    L_{k+1}^i ← ∅;
3:    NL_{k+1}^i ← ∅;
4:    for j ← i − 1 to 0 do
5:     Assume L_k [i] = x_1 x_2 ··· x_k and L_k [j] = y x_2 ··· x_k (y ≻ x_1 ≻ x_2 ≻ ··· ≻ x_k); //y ∈ L_1, x_s (1 ⩽ s ⩽ k) ∈ L_1
6:      l ← y x_1 x_2 ··· x_k; //L_k [i] ∪ L_k [j]
7:      l.N − list ← NL_intersection(NL_k [i], NL_k [j]);
8:      if l.count ⩾ |DB| × ξ then
9:        L_{k+1}^i ← L_{k+1}^i ∪ {l};
10:       F ← F ∪ {l};
11:       NL_{k+1}^i ← NL_{k+1}^i ∪ {l.N − list};
12:     end if
13:   end for
14:   if L_{k+1}^i ≠ ∅ then
15:     if NL_k [i].length() = 1 then
16:       Assume L_{k+1}^i = {P_1, ..., P_n} where P_i = y_i x_1 x_2 ··· x_k
17:       for any p = y_{v1} y_{v2} ··· y_{vu} x_1 x_2 ··· x_k (1 ⩽ v_1 < v_2 < ··· < v_u ⩽ n) do
18:         p.count ← NL_k [i].count;
19:         F ← F ∪ {p};
20:       end for
21:     else
22:       Call mining_L (L_{k+1}^i, NL_{k+1}^i);
23:     end if
24:   end if
25: end for
```

itemset *be* and itemset *ce* also contain only one PP-code. So, According to Corollary 1, the support of *bce* is 3 without intersecting the N-lists of *be* and *ce*.

Let L_k be the set of frequent k-itemsets, and NL_k be the set of N-lists of frequent k-itemsets. Based on the above analysis, we have the following algorithm for mining frequent $k(>2)$-*itemsets* using N-lists.

Note that *l.count* means the support of itemset l. According to Property 5, it can be computed by summing value of count attribution of each PP-code in the N-list of l. The Procedure *mining_L* uses deep-first strategy to mine frequent itemsets. L_{k+1}^i stands for the set of frequent $(k+1)$-itemsets that end with $x_1x_2\cdots x_k$ (that is, $L_k[i]$). NL_{k+1}^i is the set of N-lists of elements in L_{k+1}^i.The code section from line 15 to 20 generates all relvenat frequent itemsets without candidate generation by making use of Corollary 1.

Note that we do not generate frequent 2-itemsets in the implementation of *mining_L*. In our implementation, the frequent 2-itemsets are generated by Algorithm 3 in Subsection 4.2. In fact, our implementation of *mining_L* begins by generating frequent $k(>2)$-itemsets from frequent 2-itemsets. For the sake of integrity and easy of illustration, the Procedure *mining_L* begin with frequent 1-itemsets as shown in Algorithm 5.

To get a better understanding of our method, let's take Example 1, mentioned in Subsection 3.2, as a running example. In Example 1, the minimum support ξ is 0.4 and the frequent 1-itemsets set F_1 is $\{a, b, c, e, f\}$. By running Algorithm 2, we know that the N-lists of b, c, e, f, and a are $\langle(4,8):4\rangle$, $\langle(1,2):1\rangle$,$\langle(5,6):3\rangle$, $\langle(6,5):3\rangle$, $\langle(2,1):1\rangle$,$\langle(8,4):1\rangle$,$\langle(9,7):1\rangle$, and $\langle(3,0):1\rangle$,$\langle(7,3):1\rangle$) respectively. By running Algorithm 3, we directly find all frequent 2-itemsets, which are *ca*, *bf*, *cf*, *be*, *ce*, and *bc*. Then, by running Algorithm 4, we get the N-lists of these frequent 2-itemsets, which are

$\langle(1,2):1\rangle, \langle(5,6\rangle:1\rangle)$ (the N-list of ca), $\langle(4,8):2\rangle$ (the N-list of bf), $\langle(1,2):1\rangle, \langle(5,6):1\rangle)$ (the N-list of cf), $\langle(4,8):3\rangle$ (the N-list of be), $\langle(5,6):3\rangle)$ (the N-list of ce), $\langle(4,8):3\rangle$ (the N-list of bc). With Algorithm 5, we obtain two candidate 3-itemsets: bcf and bce. bcf is generated by combining bf and cf and bce is generated by combining be and ce. By running Algorithm 4, the N-lists of bcf and bce are $\langle(4,8):1\rangle$ and $\langle(4,8):3\rangle$ respectively. By scanning their N-lists, we know that the supports of bcf and bce are 1 and 3 respectively. The minimum threshold is $2(0.4*5)$. Therefore, bcf is infrequent and bce is frequent. No candidate 4-itemsets have been generated so far because we have only one frequent 3-itemsets. Therefore, the running of our method is finished. Note that the single path property is not used in the example because the example is too simple and has not enough long frequent itemsets.

5 Experimental evaluation

In this section, we report three sets of experiments in which the running time, memory consumption, and scalability of PrePost are compared with well-known algorithms. Note that, the frequent itemsets discovered by the PrePost algorithm are the same as those discovered by the compared algorithms in all experiments. This confirms that the result generated by the PrePost algorithm is correct and complete.

5.1 Experiment setup

We used three real datasets and two synthetic datasets in our experiments. These datasets were often used in previous study of frequent itemset mining. The three real datasets are Pumsb, Retail, and Accidents (from http://fimi.cs.helsinki.fi/testdata.html). The PUMS dataset contains census data. The Retail dataset contains retail market basket data from an anonymous retail store. The Accident datasets contain a rich source of information on different circumstances in which accidents have occurred. The two synthetic datasets are T10I4D100K and T25I10D100K that were generated by the IBM generator (http://www.almaden.ibm.com/cs/quest/syndata.html). To generate T10I4D100K, the average transaction size and average maximal potentially frequent itemset size are set to 10 and 4, respectively, while the number of transactions in the dataset and different items used in the dataset are set to 100K and 1K, respectively. To generate T25I10D100K, the average transaction size and average maximal potentially frequent itemset size are set to 25 and 10, respectively, while the number of transactions in the dataset and different items used in the dataset are set to 100K and 1K, respectively.

Pumsb and Accidents are both quite dense, so a large number of frequent itemsets will be mined even for very high values of minimum support. Compared with the real datasets, the synthetic dataset is much sparser. Table 2 shows the characteristics of the real and synthetic datasets used in our experiments, where shows the average transaction length (denoted by Avg. length), the number of items (denoted by #Items) and the number of transactions (denoted by #Trans) in each dataset. For the sake of evaluation, we choose dense and sparse datasets with different size to study PrePost performance on different data environments.

We have compared the PrePost algorithm with two FP-growth algorithms, i.e. original FP-growth algorithm and FP-growth* algorithm, and two vertical algorithms, i.e. original dEclat algorithm and eclat_goethals algorithm. Based on the original published papers [1,5], we implemented the original FP-growth algorithm and the original dEclat algorithm by Microsoft/Visual C++. For the sake of simplicity, we denote them by FP-growth and dEclat respectively. FP-growth* [13] is one of the best FP-growth algorithms and is the winner of the FIMI03. The implementation of FP-growth* was downloaded from http://fimi.cs.helsinki.fi/src/. The eclat_goethals algorithm, which optimizes Eclat algorithm [10] by using of diffsets [5], is one of the best vertical mining algorithms and shows its advantage over other vertical mining algorithms in FIMI03 and FIMI04 (please refer to http://fimi.cs.helsinki.fi/). The implementation of eclat_goethals was downloaded from http://www.adrem.ua.ac.be/ goethals/software/. FP-growth* and eclat_goethals were both implemented by C/C++.

All experiments were performed on a Dell OptiPlex 755DT PC with Intel Core2 Duo 2.66 GHz and 2 GB Memory. The operating system was Microsoft Windows XP. Note that we did not directly compare

Table 2 A summary of the datasets

Database	Avg. length	#Items	#Trans
Pumsb	74	2113	49046
Retail	10.3	16470	88162
Accidents	33.8	468	340183
T10I4D100K	10.1	870	100000
T25I10D100K	24.9	990	99822

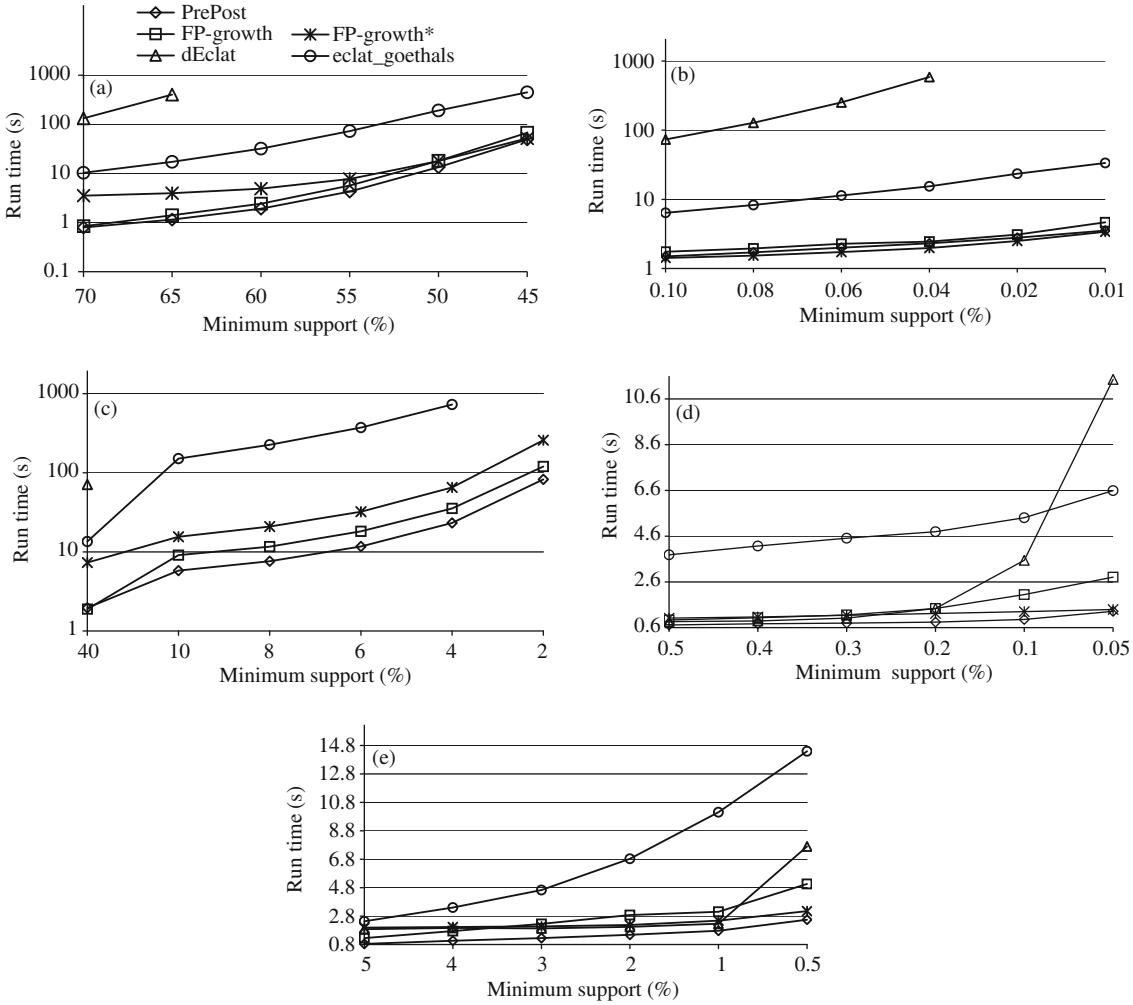

Figure 7 Running time for (a) Pumsb, (b) Retail, (c) Accidents, (d) T10I4D100K and (e) T25I10D100K.

our results with those in some published papers on different experiment platforms because different experiment platforms, such as software and hardware, may differ greatly in the runtime for the same algorithms. So, it is very fair that we compare these algorithms in the same running environment.

5.2 Comparison of running time

Figure 7 show the running time of the compared algorithms on Pumsb, Retail, Accidents, T10I4D100K, and T25I10D100K with different minimum supports. Due to the fact that data distributions of different datasets are different, we set different ranges of minimum support for different datasets in order to mine frequent itemsets in a reasonable time. Note that running time here means the total execution time, which is the period between input and output.

The X and Y axes in the five figures stand for running time and minimum support, respectively. In addition, Y axis in Figures 7 (a), (b) and (c) is logarithmic for better demonstration.

Figure 7(a) shows the running time of the compared algorithms on Pumsb. PrePost runs fastest among five algorithms under all minimum supports. Under large minimum supports, FP-growth runs faster than FP-growth*. However, FP-growth* runs faster than FP-growth when minimum support is no more than 50%. Although eclat_goethals run faster than dEclat, it is about an order of magnitude slower than PrePost, FP-growth, and FP-growth*. When minimum support is not more than 60%, dEclat fails to discover all frequent itemsets in 1000 s.

Figure 7(b) shows the running time of the compared algorithms on Retail. PrePost and FP-growth* run fastest among five algorithms under all minimum supports. The differences between PrePost and FP-growth* can be neglected. Under small minimum supports, FP-growth is slower than PrePost and FP-growth*. eclat_goethals is still faster than dEclat and is still about an order of magnitude slower than PrePost, FP-growth, and FP-growth*. Once again, we find that when minimum support is not more than 0.02%, dEclat cannot find all frequent itemsets in 1000 s.

Figure 7(c) shows the running time of the compared algorithms on Accidents. PrePost is most efficient and runs about two times faster than FP-growth, and runs about three times faster than FP-growth*. eclat_goethals is still inferior to PrePost, FP-growth, and FP-growth* and is about an order of magnitude slower than them. dEclat is still most inefficient and runs more than 1000 s again when the minimum support is 10%. In addition, eclat_goethals become inefficient and runs more than 1000 s when the minimum support is 2%.

Figure 7(d) shows the performance comparison of the algorithms on T10I4D100K. PrePost is most efficient among five algorithms under all minimum supports. FP-growth runs faster than FP-growth* under large minimum supports while FP-growth* runs faster than FP-growth under small minimum supports. Overall, FP-growth* is a little superior to FP-growth. eclat_goethals is slower than PrePost, FP-growth, and FP-growth*. However, the differences in running time are within an order of magnitude. The performance of dEclat is surprising. It is just inferior to PrePost and is superior to other three algorithms under high minimum supports.

Figure 7(e) shows the performance comparison of the algorithms on T25I10D100K. The results on T25I10D100K are almost the same as those on T10I4D100K except in one case where dEclat is faster than eclat_goethals under all minimum supports.

Figures 7 (a) and (c) show that when dense datasets are used, PrePost, FP-growth*, and FP-growth are much faster than dEclat and eclat_goethals. Figure 7(b) shows that when the values of minimum support are very low, PrePost, FP-growth*, and FP-growth are also much faster than dEclat and eclat_goethals. This is because both dense datasets and low minimum supports mean that the number of frequent itemsets will be very large. Although dEclat and eclat_goethals adopt efficient data structures for boosting the mining speed, they still employ candidate set generation-and-test strategy to discover frequent itemsets, as has been shown to be far from efficient for mining a huge number of frequent itemsets [1,13]. Both FP-growth* and FP-growth employ partitioning-based, divide-and-conquer method to directly discover frequent itemsets without candidate generation. Although PrePost also employs candidate set generation-and-test strategy to discovery frequent itemsets, it also adopts an efficient technique based on Corollary 1 to find frequent itemsets quickly without generating candidates. We believe that the technique has great influence on the performances of PrePost. However, the exact influence of the technique is still unknown because it is very hard to be analyzed quantitatively.

When datasets are sparse or the values of minimum support are moderate, the advantage of PrePost, FP-growth*, and FP-growth over dEclat and eclat_goethals is not so very distinct. Figures 7 (d) and (e) support the above conclusion.

In summary, no matter which dataset is chosen, experimental results suggest the following ordering of these algorithms as running time is concerned:

$$PrePost > \{FP\text{-}growth*, FP\text{-}growth\} > eclat\_goethals > dEclat,$$

where "$>$" means "faster than".

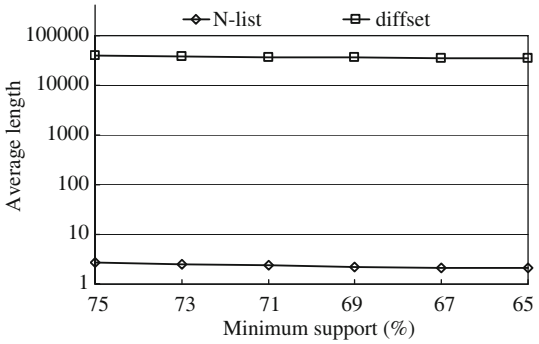

Figure 8 Comparison of average length for Pumsb.

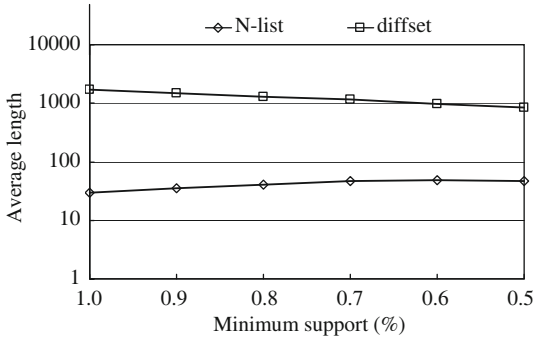

Figure 9 Comparison of average length for T25I10D100K.

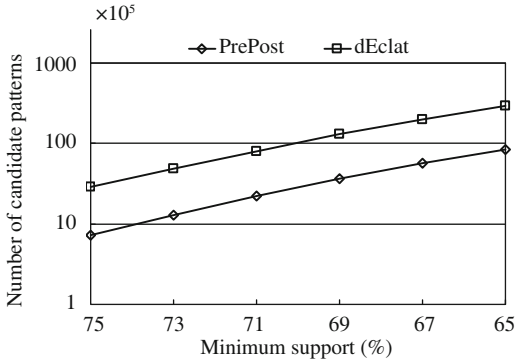

Figure 10 Comparison of candidate itemsets for Pumsb.

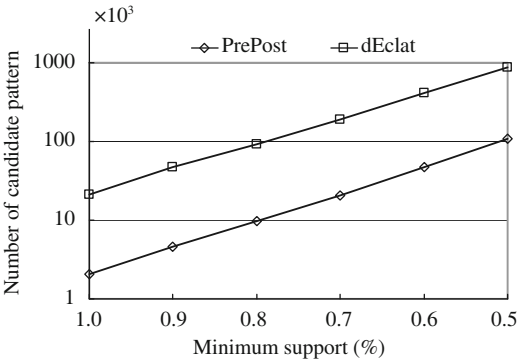

Figure 11 Comparison of candidate itemsets for T25I10D100K.

The reason why PrePost is superior to FP-growth-alike algorthms can be explained as follows. Although PrePost adopts the structure of PPC-Tree, which is similar to FP-tree, it avoids the time consuming process of constructing a lot of conditional FP-tree as in FP-growth by simply intersecting N-lists. In fact, the intersection of two N-lists can be corresponding to the construction of a conditional FP-tree. However, our experimental result shows that the intersecting action of two N-lists is more efficient. In addition, PrePost can also directly find frequent itemsets without generating candidates as FP-growth does. This makes PrePost still efficient when datasets are dense and frequent itemsets are long.

The reason why PrePost outperforms Tid-set based vertical algorithms lies in the following facts: 1) the length of N-list is much shorter than previously proposed vertical structures (such as Diffsets), and 2) PrePost adopts Corollary 1 to find relevant frequent itemsets quickly without generating candidates in some cases. To validate 1) and 2), we chose Pumsb and T25I10D100K as the test datasets. Figures 8 and 9 show the average length of N-lists and diffsets on Pumsb and T25I10D100K with different minimum supports respectively. Note that the basic unit of diffsets is transaction ID number and the basic unit of N-lists is PP-code, which is triple. The length of a diffset or an N-list is the number of basic units. Figure 8 indicates that the average length of diffsets of dEclat is about four orders of magnitude bigger than that of N-lists of PrePost on Pumsb. Figure 9 shows that the average length of diffsets of dEclat is over one order of magnitude bigger than that of N-lists of PrePost on T25I10D100K. It should be pointed out that the average length becomes shorter with decreasing minimum support because diffsets or N-lists of itemsets with lower support are usually shorter. Figures 10 and 11 shows the number of candidate itemsets generated by PrePost and dEclat. We observe that the number of candidate itemsets of dEclat is about 4 times of that of PrePost on Pumsb. On T25I10D100K, the number of candidate itemsets of dEclat is over 10 times of that of PrePost.

5.3 Comparison of memory usage

As for storage cost, we used Pumsb and T25I10D100K as test datasets and adopted the largest memory usage in the whole runtime of an algorithm as its memory usage.

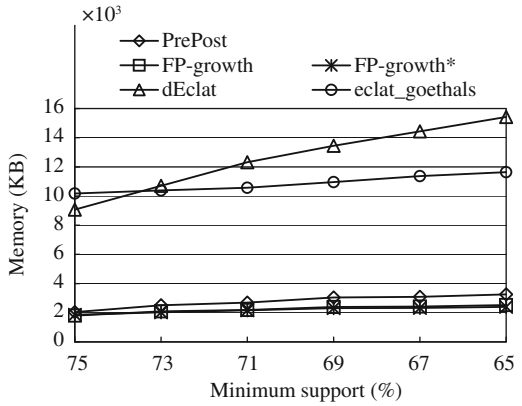

Figure 12 Memory usage comparison for Pumsb.

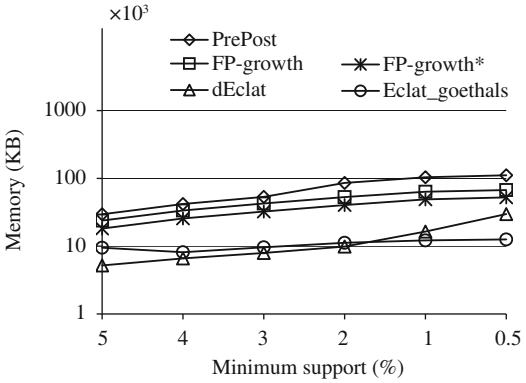

Figure 13 Memory usage comparison for T25I10D100K.

Table 3 Running time (s) of all algorithms on datasets generated from T10I4

Algorithms	Datasets				
	D100K	D200K	D500K	D1000K	D2000K
PrePost	0.953	2.218	5.859	12.546	27.156
FP-growth	2.046	3.922	10.171	21.218	45
dEclat	3.546	9.859	13.093	26.203	53.015
FP-growth*	1.296	2.797	7.843	16.421	34.656
eclat_goethals	5.406	10.312	24.828	48.921	96.73

Figure 12 shows the memory cost of the compared algorithms on Pumsb, which is a dense dataset. The memory consumed by eclat_goethals and dEclat is about 4 to 5 times of that of FP-growth, FP-growth*, and PrePost. The memory usage of PrePost is on average about 1.2 times of the memory usage of FP-growth and FP-growth*. The difference between FP-growth and FP-growth* can be neglected.

Figure 13 shows the memory cost of the compared algorithms on T25I10D100K, which is a sparse dataset. However, the memory consumed by FP-growth, FP-growth*, and PrePost is more than that of eclat_goethals and dEclat. The memory usage of PrePost is on average about 1.8 times that of the memory usage of FP-growth* and about 1.4 times of the memory usage of FP-growth.

The reason why PrePost consumes more memory than FP-growth and FP-growth* can be explained as follows. For these algorithms, the most consumption of memory is the original PPC-tree or FP-tree. Since a node of PPC-tree contains more information (*pre-order* and *post-order*) than a node of FP-tree, the PPC-tree of a dataset is a little bigger than its FP-tree. In addition, Algorithm 3 adopted by PrePost for directly finding frequent 2-itemsets need maintains a two-dimension array, which needs large memories when there are a lot of frequent 1-itemsets.

In summary, FP-growth, FP-growth*, and PrePost consume less memory than eclat_goethals and dEclat when datasets are dense, while the situation is the opposite when datasets are sparse.

5.4 Scalability

Because the parameters of synthetic datasets are easily adjustable, we tested the scalability of all algorithms by running them on data sets generated from T10I4 and T25I10. The number of transactions in the data sets for Tables 3 and 4 ranges from 100k to 2 M. In all data sets generated from T10I4, the number of items is 1000, the average transaction length is 10, and the average maximal potentially frequent itemset size is 4. In all data sets generated from T25I10, the number of items is 1000, the average transaction length is 25, and the average maximal potentially frequent itemset size is 10. All algorithms ran on data sets from T10I4 for minimum support 0.1 percent and data sets from T25I10 for minimum support 1 percent.

Table 4 Running time (s) of all algorithms on datasets generated from T25I10

Algorithms	Datasets				
	D100K	D200K	D500K	D1000K	D2000K
PrePost	1.765	3.754	9.812	20.500	38.610
FP-growth	3.109	7.375	18.500	38.078	79.437
dEclat	2.265	4.937	13.125	23.296	49.796
FP-growth*	2.484	5.031	13.140	28.031	62.421
eclat_goethals	10.109	19.562	47.843	94.812	189.015

Table 5 Speed scalability of all algorithms on datasets generated from T10I4

	D100K–D200K	D200K–D500K	D500K–D1000K	D1000K–D2000K	Average
PrePost	2.327387	2.641569	2.141321	2.164515	2.318698
FP-growth	1.916911	2.59332	2.086127	2.120841	2.1793
dEclat	2.780316	1.328025	2.001298	2.023242	2.03322
FP-growth*	2.158179	2.804076	2.093714	2.110468	2.291609
eclat_goethals	1.90751	2.804076	1.970396	1.977269	2.164813

Table 6 Speed scalability of all algorithms on datasets generated from T25I10

	D100K–D200K	D200K–D500K	D500K–D1000K	D1000K–D2000K	Average
PrePost	2.126912	2.613745	2.089278	1.883415	2.178338
FP-growth	2.372145	2.508475	2.05827	2.086165	2.256264
dEclat	2.179691	2.658497	1.774933	2.137534	2.187664
FP-growth*	2.025362	2.611807	2.133257	2.226856	2.249321
eclat_goethals	1.935107	2.611807	1.981732	1.993577	2.130556

Tables 3 and 4 show the running time of all algorithms on all datasets. For evaluating the scalability of a algorithm, we use the ratio of the running time on a dataset to the running time on the previous dataset (the biggest one that is small than it). For example, the value of the element, corresponding to PrePost and D100K–D200K in Table 5, is 2.327387. This value is the result of dividing 2.218, the running time of PrePost on D200K (T10I4), by 0.953, the running time of PrePost on D100K (T10I4). Tables 5 and 6 show the speed scalability of all algorithms. The last columns of Tables 5 and 6 stand for the average speed scalability. From these tables, we find that all five algorithms show almost the same scalability.

However, if we fix occurring frequency of each itemsets and just replicate transactions, PrePost, FP-growth*, and FP-growth will scale much better than eclat_goethals and dEclat. This is because no matter how many times the transactions are replicated, the former builds up an FP-tree or PPC-tree with a size identical to that of the original one, which is built on nonreplicated transactions.

From the figures of runtime, memory consumption, and scalability of all algorithms, we can draw the conclusion that PrePost is one of the best algorithms for mining frequent itemsets whether the data sets are dense or sparse. For sparse datasets, the memory consumption of PrePost is unfortunately very high because it is essentially based on data structure similar to FP-tree. This is a common fault of algorithms based on FP-tree [13]. However, when sparse datasets are used, PrePost is still the fastest algorithm, thanks to the N-list technique.

5.5 Discussion

The datasets we have used in our experiments have been widely used in previous researches. Our algorithm not only obviously outperforms two state-of-the-art vertical algorithms (dEclat and eclat_goethals) and FP-growth, but also outperforms FP-growth*, the state-of-the-art implementation of FP-growth, in most

cases. We did not use Apriori algorithm and other vertical algorithms (such as Eclat algorithms) as baseline algorithms because FP-growth, dEclat and eclat_goethals have been shown to outperform these algorithms [1,5]. In addition, we did not make comparison with PPV because FP-growth is claimed to be better than PPV on real datasets [6].

In fact, PrePost combines the advantages of FP-growth and vertical algorithms. Both PrePost and FP-growth use compressed prefix tree to store the datasets. In some cases, PrePost also employs the same way as FP-growth to find frequent itemsets directly without generating candidates. However, PrePost computes the count of an itemset by simply scanning its N-list without constructing complexity conditional FP-tree. On the other hand, PrePost and vertical algorithms adopt similar data structure originating from inverted list, which makes the counting of itemsets' supports very simple and avoids scanning the datasets repeatedly. However, N-list used by PrePost is much smaller than that used by vertical algorithms. As a consequence, the efficiency of PrePost is much better than vertical algorithms. PrePost can directly find frequent itemsets without generating candidates. This is another important reason for PrePost to outperform vertical algorithms.

Storage cost for maintaining the N-lists of itemsets is decided by the dataset used. If the dataset is dense, the storage cost is low. If the dataset is sparse, the storage cost is high, because the density of datasets decides the size of PPC-tree, and hence the storage cost for maintaining the N-lists. As stated in Subsection 5.3, because PPC-tree contains more information than FP-tree, the memory usage of PrePost is larger than that of FP-growth or FP-growth*. Therefore, how to design a good structure to store N-lists is an interesting issue.

Note that this paper focuses on algorithmic concepts rather than implementations. As we know, the runtime is different for different implementations of the same algorithm. For example, we find the implementation of FP-tree construction in FP-growth* is faster than our implementation of FP-tree construction in FP-growth. Therefore, although the implementations of FP-growth* and eclat_goethals used in this experiment are all in C/C++, the different data structures and libraries employed by these two algorithm make the comparison unfair. At present, our implementation of PrePost in this paper only adopts basic programming techniques. We think that if we employ some specific data structures and libraries suitable for PrePost, it will be improved both in time and space. This will be our future work.

6 Conclusions

In this paper, we have introduced a novel data structure, N-list, for storing compressed and lossless information about frequent itemsets. Based on N-lists, we developed an algorithm, PrePost, for fast mining frequent itemsets in databases. The advantages of the PrePost algorithm over other algorithms lie in the following facts: 1) It employs a compact data structure, N-list, which is usually substantially smaller than the original databases and thus avoids costly database scans in the subsequent mining processes. 2) The counting of itemsets' supports is transformed into the intersection of N-lists and it employs an efficient strategy with complexity of $O(m+n)$ for intersecting two N-lists, where m and n are the cardinalities of the two N-lists respectively. 3) By making use of the single path property of N-list, it directly finds frequent itemsets without generating candidate itemsets in some cases.

We have implemented the PrePost algorithm and studied its performance in comparison with several well-known frequent itemset mining algorithms on a variety of real and synthetic datasets. Our performance study shows that the PrePost algorithm is more efficient than the algorithms compared. It is best in 4 out of 5 datasets as running time is concerned and it was not worse than the best competitor in the remaining dataset. Even though the algorithm consumes more memory when the data sets are sparse, it is still the fastest one.

As future extensions of this work, first we will extend our algorithm to mine maximal frequent itemsets [18,19], closed frequent itemsets [20,21], compressed frequent-itemset sets [22,23], high utility patterns [24,25], and impact-targeted activity patterns [26], second, we will further explore our method to discovery frequent itemsets from data streams [27,28], third we will try to effectively integrate the idea of PrePost into the process of mining patterns from complex data [29–31]. Finally, as the available data is

growing exponentially, parallel/distributed techniques, such as cloud computing, are becoming the main tools for analyzing and mining very huge databases. Therefore, the parallel/distributed implementation of our algorithm is also an interesting work.

Acknowledgements

This work was partially supported by National Natural Science Foundation of China (Grant No. 61170091) and National High Technology Research and Development Program of China (863 Program) (Grant No. 2009AA-01Z136).

References

1 Han J W, Pei J, Yin Y W. Mining frequent itemsets without candidate generation. In: The 2000 ACM SIGMOD International Conference on Management of data (SIGMOD'00), New York, 2000. 1–12
2 Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases. In: The 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93), Washington, 1993. 207–216
3 Han J, Cheng H, Xin D, et al. Frequent itemset mining: current status and future directions. Data Min Knowl Discov, 2007, 15: 55–86
4 Baralis E, Cerquitelli T, Chiusano S. IMine: index support for item set mining. IEEE TKDE J, 2009, 21: 493–506
5 Zaki M J, Gouda K. Fast vertical mining using diffsets, In: The 9th ACM SIGKDD International Conference on. Knowledge Discovery and Data Mining (SIGKDD'03), Washington, 2003. 326–335
6 Deng Z H, Wang Z H. A new fast vertical method for mining frequent itemsets. Int J Comput Intell Syst, 2010, 3: 733–744
7 Agrawal R, Srikant R. Fast algorithm for mining Association rules. In: The 20th International Conference on Very Large Data Bases (VLDB'94), Santiago de Chile, 1994. 487–499
8 Savasere A, Omiecinski E, Navathe S. An efficient algorithm for mining association rules in large databases. In: The 21th International Conference on Very Large Data Bases (VLDB'95), Zurich, 1995. 432–443
9 Shenoy P, Haritsa J R, Sundarshan S, et al. Turbo-charging vertical mining of large databases. In: ACM International Conference on Management of Data and Symposium on Principles of Database Systems (SIGMOD'00), Dallas, 2000. 22–33
10 Zaki M J. Scalable algorithms for association mining. IEEE TKDE J, 2000, 12: 372–390
11 Pei J, Han J, Lu H, et al. H-mine: Hyper-structure mining of frequent itemsets in large databases. In: IEEE International Conference on Data Mining (ICDM'01), San Jose, 2001. 441–448
12 Liu G, Lu H, Lou W, et al. Efficient mining of frequent itemsets using ascending frequency ordered prefix-tree. DMKD J, 2004, 9: 249–274
13 Grahne G, Zhu J. Fast algorithms for frequent itemset mining using FP-Trees. IEEE TKDE J, 2005, 17: 1347–1362
14 Woon Y K, Ng W K, Lim E P. A support-ordered trie for fast frequent itemset discovery. IEEE TKDE J, 2004, 16: 875–879
15 Ananthanarayana V S, Murty N M, Subramanian D K. An incremental data mining algorithm for compact realization of prototypes. Pattern Recognit, 2001, 34: 2249–2251
16 Grust T. Accelerating xpath location steps, In: The 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD'02), Madison, 2002. 109–120
17 Kleinberg J, Tardos E. Algorithm design. Boston: Addison Wesley, 2005
18 Bayardo Jr R J. Efficiently mining long itemsets from databases. In: ACM SIGMOD International Conference on Management of Data (SIGMOD'98), Seattle, 1998. 85–93
19 Burdick D, Calimlim M, Flannick J, et al. Mafia: A maximal frequent itemset algorithm. IEEE TKDE J, 2005, 17: 1490–1504
20 Wang J Y, Han J, Pei J. CLOSET+: Searching for the Best Strate-gies for Mining frequent closed itemsets. In: The 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'03), Washington, 2003. 236–245
21 Lee A J T, Wang C S, Weng W Y, et al. An efficient algorithm for mining closed inter-transaction itemsets. Data Knowle Eng, 2008, 66: 68–91

22 Xin D, Han J, Yan X, et al. On compressing frequent itemsets. Data Knowl Eng, 2007, 60: 5–29

23 Li H, Chen H. Mining non-derivable frequent itemsets over data stream. Data Knowl Eng, 2009, 68: 481–498

24 Yao H, Hamilton H J, Butz C J. A foundational approach to mining itemset utilities from databases. In: The SIAM Data Mining (SDM'04), Florida, 2004. 482–486

25 Yao H, Hamilton H J. Mining itemset utilities from transaction databases. Data Knowl Eng, 2006, 59: 603–626

26 Cao L B, Zhao Y C, Zhang C Q. Mining impact-targeted activity itemsets in imbalanced data. IEEE TKDE, 2008, 20: 1053–1066

27 Chang J H, Lee W S. Finding recent frequent itemsets adaptively over online data streams. In: The 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'03), Washington, 2003. 487–492

28 Li X, Deng Z H. Mining frequent itemsets from network flows for monitoring network. Expert Syst Appl, 2010, 37: 8850–8860

29 Agrawal R, Srikant R. Mining sequential itemsets. In: The 11th International Conference on Data Engineering (ICDE'95), Taiwan, 2003. 3–14

30 Yan X, Han J. gSpan: Graph-based substructure pattern mining. In: The 2002 IEEE International Conference on Data Mining (ICDM'02), Maebashi, 2002. 721–724

31 Cao L B, Zhang H F, Zhao Y C, et al. Combined mining: Discovering informative knowledge in complex data. IEEE Trans Syst Man Cybern Part B-Cybern, 2011, 41: 699–712