The Local Power Set in Apriori algorithm's Transaction Scanning for Eficient Frequent Item sets Mining

Pitaya Poompuang[1] and Anucha Tungkasthan[2]

ABSTRACT: Frequent itemsets generation has known by researchers as the most computationally expensive task in association rule mining. To extract association rules, the classical algorithm, Apriori, needs to scan a database several times during frequent itemsets generation. Some researchers have solved such problems by applying new data structures instead of using the original structure. However, some mechanisms for scanning data through the new data structure must be provided. Each mechanism also needs CPU-time for its operation. In this paper, we present an algorithm that generates frequent itemsets based on the concept of power set enumeration over transactions, which takes only three iterations over the data. We have evaluated it against the closest algorithm which also applies the concept of a power set. The results show that our proposed method takes less CPU-time than other methods. Based on the proposed method, two new versions were developed by taking advantage of set operations in the pruning phase. We have evaluated them against the most popular frequent itemset mining algorithm, FP-Growth. The results show that our two modified versions give the best performance in terms of taking less CPU-time when the support thresholds are given at lower levels..

Keywords: Frequent Item Sets, Local Power Set, Pruning

DOI: 10.37936/ecti-cit.2021151.216479
Received September 16, 2019; revised February 20, 2020; accepted June 9, 2020; available online January 5, 2021

1. INTRODUCTION

Association rules refer to confident patterns of items that appear together in a large database. They provide useful information that can be adopted in various areas [1], for instance in medical diagnosis [2][3], protein sequence analysis [4], census data analysis [5-6], customer relationship management [7-8], and so on. The process of generating association patterns is so-called association rule mining or ARM. It was first introduced by Agrawal et al [9] in 1993. In the paper, the authors defined the problem of finding frequent itemsets as a sub-problem of association rule mining. In 1994, they created an algorithm named Apriori for association rule mining, and followed that by two extensions of Apriori, named AprioriTid and AprioriHybrid [10]. The authors also showed that association rules are very helpful in market basket analysis. Currently, it is the most well-known classical algorithm for ARM in data mining and knowledge discovery.

Basically, the process of ARM can be achieved by performing two important sub-sequential tasks. One is known as frequent itemsets generation, and the other is association rules generation. It is well known by researchers that the former is more complex and requires much more CPU time than the later. As the former requires too many system resources, it then has become an incentive for researchers to turn their attention toward improving the efficiency of the task. Most of the researchers have focused their work on reducing time and memory usage in computation. For example, FP-Growth transforms the dataset into a frequent pattern tree structure and then generates frequent itemsets by traversing this structure instead of scanning the original dataset. This approach can finish with two scans of the dataset without candidate item sets generation. In 2013, an algorithm that applies the concept of global power set enumeration was

[1,2]Department of Computer Technology, Rajamangala University of Technology Thanyaburi(RMUTT), Pathum Thani, Thailand. E-mail: p_pitaya@rmutt.ac.th and anucha_t@rmutt.ac.th

proposed [11]. Although the authors claimed that it takes only two dataset scans, it must also spend time to manipulate data stored in the global power set.

One can observe that most algorithms try to reduce the number of passes over the dataset by using other formats of data structures rather than the original format of dataset. However, they must face the same problem (i.e., using the time to manipulate other formats of data in memory). Until now, there have not been algorithms that can be called optimal (i.e., suitable for the general context of applications), due to the variety of data formats (i.e., data types and density of data), which directly affect the performance of the system. In this paper, we first introduce an algorithm for mining frequent itemsets, called the Local Power Set in Transaction Scanning (LPTScan). Then, we conduct an experiment to evaluate the performance of the LPTScan against closely related work [11] because those authors also used the concept of (global) power sets in their work. After that, we modify the LPTScan to accelerate its overall speed by taking advantage of the intersection and subtraction set operations to prune useless item sets from the dataset. The two new versions of LPTScan require scanning the dataset only twice. Finally, we compare all of our proposed methods with FP-Growth, the well-known efficient algorithm that generates all frequent item set without candidate item sets generation and requires scanning the dataset only twice.

There are three contributions provided in this paper, 1) an efficient frequent item sets generation algorithm using the concept of power set enumeration, 2) a novel method to define a minimum support threshold appropriately for each dataset, and 3) two efficient pruning techniques based on intersection and subtraction set operations to remove the item sets that do not meet the minimum support threshold.

The rest of this paper is organized as follows: in the next section, we give some background of the frequent itemsets generation task. In section 3, we discuss some related works focusing on the task improvement. In section 4, we explain how the LPTScan works and its two extensions. We also address the experiments and include an evaluation discussion. In the last section, we provide some conclusions about the work we have done in this paper.

2. BACKGROUND

The objective of this section is to provide readers with a brief introduction to related terminology and an explanation about the frequent itemsets generation task used in the classical algorithm.

2.1 Terminology and Related Symbols

To understand the process of frequent item sets generation, the following concepts, terms, and symbols must be understood:

- An "item set" refers to a set of items that share the same class. Note that items can be any type of objects, such as product items, words in a document, documents in a library, etc.
- (1)-item sets refer to the item sets that have a single member, for example 'a'. A set with k members can be called a (k)-item set, where $k > 0$.
- The support count of an item set refers to the number of transactions containing that item set.
- The minimum support count refers to the number of times an item set appeared in a dataset. It can be used as a criterion to determine whether an item can be classified into a frequent itemset or not.
- Frequent itemsets refer to sets of items whose support count reached the minimum support threshold.
- Candidate (1)-item sets refer to sets of items consisting of both frequent itemsets and infrequent itemsets.
- A global power set refers to the power set generated from a set of all unique items in a dataset.
- A local power set refers to the power set generated from a set of items in a transaction.

Table 1: *A Simple Dataset.*

Tid	Item_set
T1	[a, c, d, e]
T2	[b, c, e]
T3	[a, b, c, e]
T4	[b, e]
T5	[c, d]
T6	[a, b, e]

Consider the six transactions of the simple dataset shown in table 1. For easier understanding, one can imagine that dataset is a collection of transactions, similar to a table in a relational database system, as shown in table 1. Each transaction is composed of two attributes, a transaction identifier (tid) and a transaction value (item_set). The *tid* is used as a variable for keeping unique identifiers of transactions. Each value of the *tid* represents an identifier of an item set in that transaction. And the attribute itemset is used as a variable for keeping item sets of transactions.

Given U denotes a set of distinct items in a domain and D represents a collection of transactions containing a set of items T; $T \subseteq U$. Given X represents a set of items; $X \subseteq T$ and $T(X)$ denotes set of transactions containing X. The $|T(X)|$ is used to refer to the support count of X. The proportion of transactions containing X in the dataset is called the support value of X, denoted by $supp(X) = |T(X)|/|D|$. The support value of X implies how important the set X is in the dataset. The item set X will be said to be a frequent item set if $supp(X)$ is not less than a given minimal support threshold. For example, let us consider the size of the dataset

D shown in table 1, $|D| = 6$. Given $X = \{b, e\}$ and $T(X) = \{T2, T3, T4, T6\}$ where $|T(X)| = 4$. The support value of X is equal to $4/6$, or 67% of the total number of transactions. Suppose we define the minimal support threshold, min_supp, is equal to 50%. The item set X is considered as a frequent item set because its support value reached the minimum support threshold.

Given X and Y represent two distinct item sets appearing within the same transaction, $X \cup Y = \phi$. The proportion of transactions containing both X and Y can be written as supp$(X \cup Y)$ or $supp(XY) = |T(X \cup Y)|/|D|$. For example, suppose $X = \{a\}$ and $Y = \{b, e\}$, so $T(X \cup Y) = \{T3, T4, T6\}$. Since the number of transactions containing X and Y is equal to 3, the support value of $X \cup Y$ then becomes to $3/6$, or 50%. The new item set derived from $X \cup Y = \{a, b, e\}$ is said to be a frequent item set.

2.2 Frequent Item sets Generation

As previously mentioned in the introduction section, the process of association rule mining that applies the Apriori algorithm can be achieved by two sequential tasks (frequent item sets generation and association rule generation). However, in this work, we aim to focus only on the frequent itemsets generation task as it could greatly affect the overall efficiency of association rule mining.

To generate frequent (1)-item sets, the Apriori counts the number of times that individual items appeared within the dataset. Each counted value expresses the support count of each item. Those itemsets are called the candidate (1)-item sets and represented by C_1. Next, the algorithm removes all item sets whose support counts do not reach criterion, minimum support threshold. All the remaining item sets then become frequent (1)-item sets, represented by L_1. For finding frequent (2)-item sets, the algorithm generates new candidate (2)-item sets by performing a self-join operation over possible two item sets that appearing in L_1. Next, it enumerates each new generated item set to formulate new candidate (2)-item sets, represented by C_2. After that, the candidate (2)-item sets whose support counts do not reach the criterion are removed. All of the remaining item sets in C_2 then become the frequent (2)-item sets, represented by L_2. For other generations of frequent (k)-item sets, the algorithm performs in the same way until candidate (k+1)-item sets cannot be further generated.

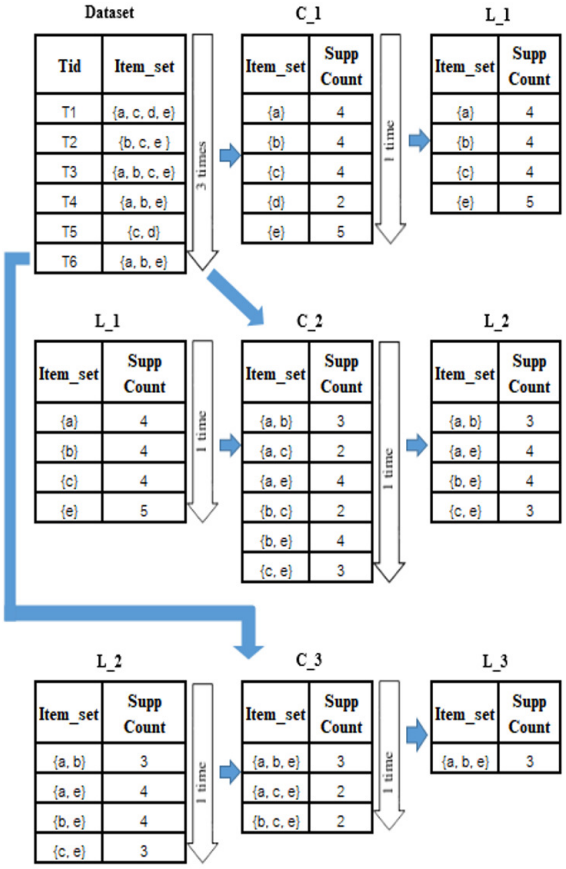

Fig. 1: *Generating Frequent item sets with 50% of minimum support..*

For example, for the dataset D shown in table 1 we define the minimum support threshold as no less than 50% of the total number of transactions. Note that typically minimum support thresholds are usually defined in terms of percentage values. In order to make it easier to understand and to reduce the computation of a support value for each item set, we suggest transforming the percentage values to numeric format by multiplying each of them by the total number of transactions as expressed in eq. (1), where $|D|$ represents the total number of transactions in dataset D.

$$min\_sup\_count = min\_sup * |D|. \quad (1)$$

Given $min\_sup = 50\%$ in this example, it implies that the item sets with support counts less than 3 cannot be promoted to be frequent item sets. To generate all frequent item sets, the Apriori runs a level-wise search approach iteratively. It generates frequent $(k+1)$-item sets based on the frequent (k)-item sets. Figure 1 shows all previous descriptions. The downward arrows in figure 1 indicate data scanning operations including the number of times.

3. RELATED WORKS

In 1994 the Apriori was first introduced and named by Agrawal et al [10]. It has become a well-known and seminal algorithm for mining confident frequent itemsets.

Since Apriori must perform the process for generating candidate (k+1)-item sets from the frequent (k)-item sets iteratively, it faces a big problem when the volume of the dataset is increased. Since a large dataset is stored in secondary storage, the input/output operation of the system must be heavily used when the dataset is scanned iteratively. In practice, Apriori is not suitable for mining frequent itemsets from a dataset containing big transactions based on a low minimum support threshold. However, there are other algorithms developed based on Apriori after it was introduced. The MSapriori [12] algorithm allows users to specify different minimum item support values for different items. Furthermore, there are some algorithms like A-Close [13] focusing on finding frequent closed itemsets which are subsets of completed frequent itemsets. The concept of this focusing is that the set of frequent closed itemsets is usually much smaller than the completed frequent item sets, while the completed frequent itemsets can be regenerated from the set of frequently closed itemsets.

The partition approach was proposed by [14]. It is based on the authors' observation. They believe that when the size of the dataset is larger than memory existing in the system, it results in expressive amounts of CPU-time consumtion. Therefore, they suggest dividing the large dataset into smaller chunks so that each chunk can be individually loaded into memory. After the partitions are loaded, the technique of Apriori algorithm can be adapted to each partition for finding frequent itemsets. Finally, it combines the results derived from each partition to formulate completed frequent itemsets. Although it can reduce the loading size of the dataset, the size of the generated candidate sets may be larger than the partitions' size of the dataset. In addition, combining frequent itemsets generated from partitions in the final step may require too much memory and too much computation time. Later, various extensions of the Apriori algorithm have been proposed continuously to overcome the big limitations (dataset scanning repeated multiple times and huge candidate item sets generation situation).

FP-Growth [15] tries to avoid multiple scanning by using compact data structures called the Frequent Pattern Tree or FP-tree, to keep frequent itemsets in memory. The Matrix Apriori [16] keeps frequent itemsets in a binary matrix. Both have claimed that they only need two passes over the dataset. Moreover, FP-Growth does not need to generate candidate itemsets to find frequent itemsets. A benchmark between them has been done by the author of [17]. Although they can reduce the number of dataset scann, in fact, they must spend time maintaining and scanning the FP-tree structures in its operations.

In 2013, the author of the paper [11] proposed an algorithm that can extract all frequent itemsets within only two passes over the dataset. In the first scan, it generates frequent (1)-item sets in the same way used in the most of other algorithms. Then the algorithm generates a power set based on the frequent (1)-item sets, called the global power set. Each item element of the global power set is assigned zero as its initial value. In the second scan, the algorithm generates a power set for each transaction based on item sets appearing within the transaction, called the local power set, represented by a hash table in memory. The algorithm also assigns an initial support count of 1 for each element of the local power set. After that, it updates support counts of item elements in the global power set by scanning for support counts of their corresponding elements that appeared in local power sets. Finally, all frequent itemsets are formulated by removing item sets if their support counts are less than the minimum support threshold. Based on our observation, the algorithm consumes too much CPU-time to update information in the global power sets. Therefore, we turn our attention to manipulate only the local power sets of transactions, instead of wasting too much time with the global power set.

Parallel and distributed computing methods have been known as valid approaches to deal with very large datasets. New parallelization techniques and distributed processing frameworks such as Hadoop in big data provide environments that let researchers easily implement and test their work in parallel. All of them are incentives for researchers to improve performance of existing algorithms. The authors in [8-9] have implemented the Apriori algorithm and the KAAL algorithm [10] on the MapReduce framework.

Recently, there are many works that try to adopt parallel and distributed processing [11] in frequent itemsets generation. However, no matter what the algorithms of the frequent itemsets generation task are, there is certainly no doubt that executing them in parallel and with distributed processing will provide better performance than running algorithms in a single process. As one considers improvements, the big challenge is to improve algorithms based on single processing. However, our proposed method can also support those parallel and distributed systems like existing algorithms do.

4. METHODOLOGIES

Since the hash table structure plays an important role and it is heavily used in our proposed algorithms, we need to briefly explain the hash table structure, which is called a dictionary in python. After that, we describe the proposed algorithm along with its asymptotic analysis by comparing it with the

related work [11] based on total complexities followed by experimental results. Next, we describe how to achieve better performance from the proposed algorithm by creating two new modified versions. Then we compare these with a highly efficient algorithm, FP-Growth.

4.1 Dictionary Format

"Dictionary" is used to refer to a data structure containing data in a set of key-value pairs in the following format; { key_1: value_1, key_2: value_2,..., key_n: value_n }. The keys serve as unique indices providing direct access to the corresponding values with which they are associated with. To apply this data structure, each item set is defined as a key and then its support count is assigned to be a value of the item set. For example, when transaction T1 in the simple dataset is scanned, each item set element in the transaction will be enumerated. The results are kept by merging item sets and their initial counting value 1. After scanning of T1 is completed, the dictionary will become {a: 1, c:1, d:1, e:1}

4.2 Local Power sets in Transaction Scanning

Basically, the complexity of power set generation grows exponentially according to the size of the original set. For example, suppose there are 30 unique items, $|U|= 30$ in the given dataset. Generating global power sets for 30 elements must loop 2^{30}, or 1073241824, times to process. It also requires 1073241824 positions in memory for keeping all elements. Then, the total complexity of this operation is around $O(2^{30})$ excluding an empty set element.

To reduce the computation time and memory usage, our approach, called Local Power sets in Transaction Scanning, or LPTScan, avoids generating the global power sets from the whole collection of unique items as it has been done in [11]. However, the LPTScan must scan the dataset three times. The first scan is for generating candidate (1)-item sets. The second scan is for updating the dataset. The last one is for generating local power sets and also for enumerating elements of the local power sets simultaneously. The pseudo-code version of the LPTScan is illustrated in figure 2.

In summary, LPTScan can be completed by performing the following three steps.

1) Candidate (1)-item sets generation: In this step, the algorithm scans the dataset for counting the number of all unique item sets and keeps the count information in the dictionary structure named C. The complexity of this processing step is $O(M \times |\bar{t}|)$, where M represents the total number of transactions and $|\bar{t}|$ denotes an average length of transactions in the dataset.

```
1   input D,min_supp
2   output : F
3   procedure : LPTScan(Dataset D)
4   begin :
5       C = { }   // hash table for candidate set
6       //generate candidate (1)-itemset
7       for all transaction t ∈ D do begin:
8          for all element e in t do begin:
9             C [e]++
10         end for
11      end for
12       // get rid of unuseful items from dataset
13      for all transaction t ∈ D do begin:
14         for e in C do begin
15            if (e in D [t]) && (C[e] < min_supp) then:
16               D[t].remove(e)
17         end for
18      end for
17      F = { } //hash table for frequent item sets
19      // frequent item set  generation
20      for all transaction t ∈ D do begin:
21         P_t= powerset(t)
22         for all subset s ∈ P_t  do begin:
23            if (s is not already in C) then:
24               C [s]++
25            if (C [s] >= min_supp) then
26               F [s] =  C [s]
27         end for
28      end for
29  end proc
```

Fig.2: *Pseudo Code for LPTScan.*

2) Eliminating unsatisfactory item sets from the dataset: In this step the algorithm scans the dataset again. It takes the user-defined minimum support threshold as a criterion to eliminate unsatisfactory item sets from the dataset. The aim of this step is to reduce the size of transactions which will also decrease the size of local power sets to be generated for transactions. The complexity of this processing step is $O(M \times N)$, where N is the total number of items in candidate (1)-item sets.

3) Extracting all frequent (k)-item sets: In the last dataset scanning phase, the algorithm generates a local power set for each transaction, then it accumulates a support count for each item element of the local power set into a dictionary structure named C. Whenever the accumulated support count of elements reaches a criterion, the algorithm will copy that item element along with its current support count to the other dictionary structure named F. As a result, all frequent (k)-item sets will be collected in F. The complexity of this step is $O(M \times 2^{|\bar{t}|})$.

The total complexity of the LPTScan is equal to $O(M \times |\bar{t}|)+O(M \times N)+O(M \times 2^{|\bar{t}|})$

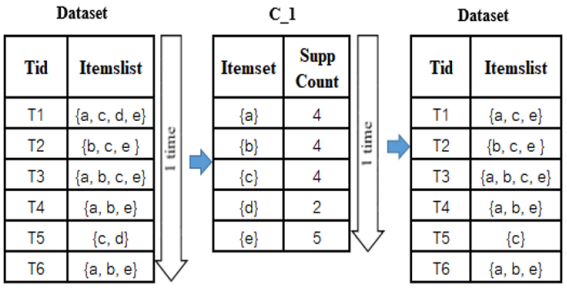

Fig.3: *Removing infrequent items from the dataset.*

For example, assume a minimum support threshold of 50%. To extract all frequent (k)-item sets from the simple dataset, in the first step, the algorithm scans the dataset for all unique items. During this scan, it calculates support counts for all unique items and stores them in a dictionary of candidate item sets, C_1. Next, it scans data in C_1 and takes advantage of it to remove all items whose support thresholds are less than the criterion from the dataset, as illustrated in figure 3. In the final step, it scans the dataset again to generate a local power set for each transaction and it accumulates a support count for each subset element in that local power set. The elements with support values equal to or exceeding the minimum support threshold will be classified as frequent itemsets, as shown in figure 4.

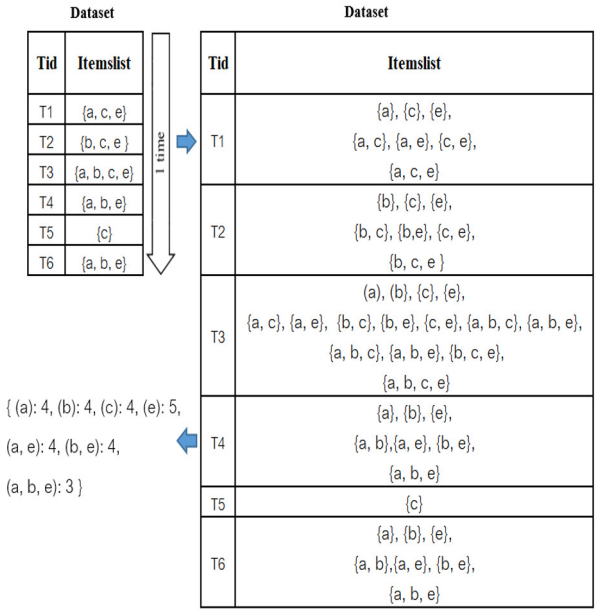

Fig.4: *Generating all frequent item sets.*

4.3 Algorithm for the work [11] and the LPTScan in asymptotic analysis

At first, we analyze the complexities of the two algorithms developed based on the same concept of power set enumeration. While one [11] applies the concept both in global and local levels, LPTScan applies simply only to the local level of power set. While the former requires two dataset scans, LPTScan needs three. In order to compare complexity of the former with the LPTScan, we divided the operation of the former into 4 steps for analysis:

1) Frequent (1)-item sets generation: the algorithm begins by extracting all unique items, called candidate (1)-item sets or C_1, from the dataset and then eliminates elements of C_1 that do not comply with the criterion, the minimum support threshold. After removal, all of the remaining values in C_1 then become the frequent (1)-item sets, L_1, by default. The complexity of this process is equal to $O(M \times |\bar{t}|) + O(N)$, where M represents the total number of transactions, $|\bar{t}|$ denotes an average size of transactions, and N is the total number of item sets in the candidate (1)-item sets, C_1.

2) Global power sets generation: The algorithm generates a power set based on the frequent (1)-item sets, called the global power set and then assigns an initial count of 0 for each item element in the global power set. The complexity of this step is equal to $O(2^{|L_1|})$.

3) Local power sets generation: The algorithm scans all transactions in the dataset again. During this scan, it removes items which do not appear in the frequent (1)-item sets from the dataset. Then it generates a local power set for each transaction based on the remaining elements of the transaction. It also assigns an initial support count of 1 for each local power set element in the local power sets simultaneously. The complexity of this step is equal to $O(M \times 2^{|\bar{t}|})$.

4) Updating global power sets: The algorithm scans all local power sets of transactions to calculate the total support count of their distinct item elements. It also updates the corresponding elements in the global power set one by one simultaneously. In the end, the algorithm eliminates the elements of the global power set that do not comply with the criterion. The remaining items in the global power set then all become frequent itemsets by default. The complexity of this step is equal to $O(M \times 2^{|\bar{t}|})$.

In summary, the total complexity of the task can be written as $O(M \times |\bar{t}|) + O(N) + O(2^{|L_1|}) + O(M \times 2^{|\bar{t}|})$. Although the LPTScan algorithm takes three passes over the dataset, when comparing its complexity with the related algorithms [11], it is much less complicated. The formula used for this comparison is quite simple and straight forward as shown in eq. (2).

$$complexity(the\ other) - complexity(\text{LPTScan}). \quad (2)$$

According to eq.(2), complexity comparison between the related algorithm [11] and our proposed can be rewritten as $[O(M \times |\bar{t}|) + O(N) +$

$O(2^{|L_1|})+O(M.2^{|\bar{t}|})+O(M.2^{|\bar{t}|})$]-[$O(M\times|\bar{t}|)+O(N\times N)+O(M\times 2^{|L_1|})$]=[$O(N)+O(2^{|L_2|})+O(M.2^{|\bar{t}|})$]-[$O(M\times N)$].

The results of the comparison show that related work [11] seems to have a higher complexity than the LPTScan. In order to get a stronger conclusion for comparison, we conducted some experiments to measure CPU-time consumption of both algorithms by executing them over two real datasets. The first one is named the "GROCERY" dataset, an open dataset collected from the French Retail Store containing 7500 transactions of product items from 120 distinct product items. The other dataset is generated by the IBM Almaden The quest research group, named the "T10I4D100K" dataset. It contains 870 distinct items over 100000 transactions of items. Table 2 shows some aspects of the two datasets.

Table 2: *Interesting attributes of datasets.*

Table	The total number of transactions	The Highest support count in Candidte 1-item sets
GROCERY	7500	1788
T10I4D100K	100000	7828

Ones can see that the size of each dataset is very large when compared to the number of transactions containing the item set with the highest support count in the candidate (1)-item sets. This information can be used to define appropriate minimum support thresholds for each dataset. Therefore, we determine a minimum support threshold by referring to the proportion between the highest support count in candidate (1)-item sets and the total number of transactions. For example, for dataset "T10I4D100K", the maximum support count of the candidate (1)-item sets is 7828 and the total number of transactions is 100000. An appropriate minimum support for the dataset should not exceed $7828/100000 = 0.078$. This implies that we cannot get any frequent (k)-item sets with support thresholds exceeding 7.8%. Therefore, an appropriate minimum support for this dataset should be less than or equal to 7.8%. Also, for the "GROCERY" dataset the appropriate minimum support should not exceed $1788/7500 = 0.23$ (around 23%). However, in order to make a better speed observation, we need to set the range of minimum support quite low, around 5.5-8% for "GROCERY" and 4-6% for "T10I4D100K". Table 3 shows the support count for each minimum support threshold used in further experiments.

Table 3: *Appropriate mininum support thresholds and supports counts for GROCERY and T1014D100k.*

GROCERY		T10I4D100k	
Minimuj Support	Support Count	Minimuj Support	Support Count
4	300	2	2000
4.5	338	2.5	2500
5	375	3	3000
5.5	413	3.5	3500
6	450	4	4000
6.5	488	4.5	4500
7	525	5	5000
7.5	563	5.5	5500
8	600	6	6000

For our experimental setting, we created two computer programs using Python 3.7 according to the algorithm proposed in [11] in addition to our proposed algorithm. Then we executed experiments on a personal computer with 32 GB of memory and a 3.60 GHz Intel Core-i7 processor. The results of running our programs on the "GROCERY" and "T10I4D100K" datasets are illustrated as bar charts in figures 5 and 6, respectively.

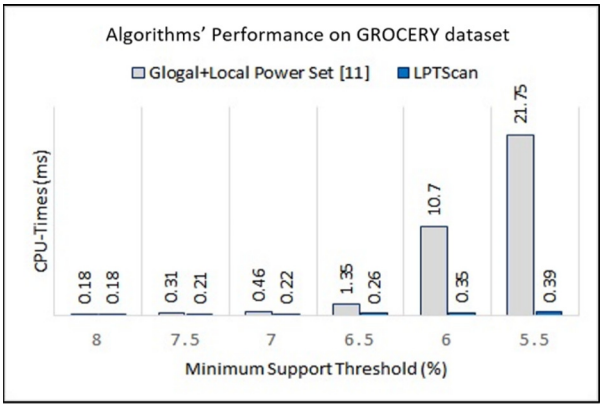

Fig.5: *CPU-times for the algorithm with global power set and the LPTScan over GROCERY.*

Figure 5 indicates that the algorithm of [11] begins to take more CPU-time than the LPTScan when the minimum support thresholds are reduced to 6.5%. Based on minimum support thresholds between 5.5 and 6.5, the LPTScan takes less CPU-time than the other (12% less on average). For the experiments on the larger dataset, the results in figure 6 indicate that both are about equal in their speed of operation. However, it appears that the other [11] tends to take more CPU-time when the minimum support thresholds are set to steadily decline. Unfortunately, when the minimum support threshold further decreases, algorithm [11] faces memory errors. The reason is that it needs to generate and maintain both global and

local power sets in memory until the process is completed. The LPTScan generates and maintains only local power sets, and does not exhaust the memory supply.

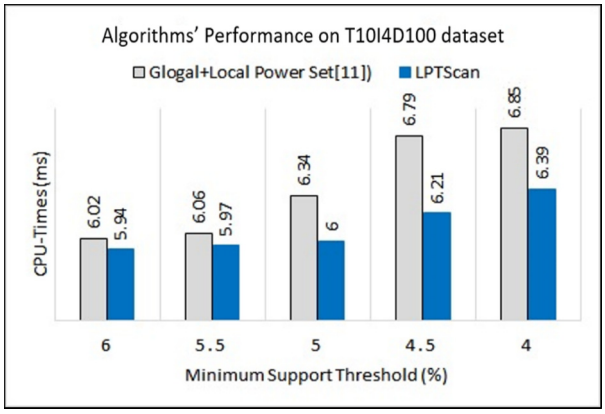

Fig.6: *CPU-times for the algorithm with global power set and the LPTScan over T10I4D100K.*

```
1   input D,min_supp
2   output : F
3   procedure : ISP_LPTScan(Dataset D)
4   begin :
5     C = { }  // hash table
6     F = { }  // hash table
7     //generate candidate (1)-itemset
8     for all transaction t ∈ D do begin:
9       for all element e in t do begin:
10        C[e]++
11        if (C[e] >= min_supp) then:
12          F[e] = C[e]
13      end for
14    end for
15    // frequent item set generation
16    for all transaction t ∈ D do begin:
17      D[t] = . D[t] ⋂ F
18      Pt= powerset(D[t])
19      for all subset s ∈ Pt  do begin:
20        if (s is not already in C) then:
21          C[s]++
22        if (C[s] >= min_supp) then
23          F[s] =  C[s]
24      end for
25    end for
26  end proc
```

Fig.7: *Pseudo Code for ISP_LPTScan.*

4.4 Accelerating of Local Power sets in Transaction Scanning

The size of transactions affects the size of local power sets being generated. We analyze the pseudo-code in figure 2 by focusing on lines numbered 13 to 17 which aim to eliminate the infrequent itemsets from the dataset. Based on our analysis, we have found that this pruning process could be improved. Then we modified the LPTScan to create the ISP_LPTScan (stands for Intersection Set Pruning in LPTScan), and the LPTScan to SSP_LPTScan (stands for the Subtraction Set Pruning in LPTScan) as shown in figures 7 and 8 respectively.

The ISP_LPTScan and the SSP_LPTScan are processed in two steps as follows:

1) Frequent (1)-item sets generation step: in this step, the algorithm scans the dataset and calculates support count for each item. The support counts are kept in the dictionary of candidate item sets C. The item sets in C with support counts greater than or equal to minimum support threshold will be copied into the dictionary of frequent item sets, F.

2) Completed frequent item sets generation: in this step, both algorithms remove some useless items from each transaction and then generate local power sets for that transaction. Finally, the algorithm enumerates each element of the local power set to accumulate its support count. When support counts of items reach the criterion, those items including their support counts will be promoted to be members of the frequent (k)-item set.

```
1   input D,min_supp
2   output : F
3   procedure : SSP_LPTScan(Dataset D)
4   begin :
5     C = { }  // hash table
6     F = { }  // hash table
7     //generate candidate (1)-itemset
8     for all transaction t ∈ D do begin:
9       for all element e in t do begin:
10        C[e]++
11        if (C[e] >= min_supp) then:
12          F[e] = C[e]
13      end for
14    end for
15    uselessC =  C[e] - F[e]
16    // frequent item set generation
17    for all transaction t ∈ D do begin:
18      Pt= powerset(D[t] - uselessC)
19      for all subset s ∈ Pt  do begin:
20        if (s is not already in C) then:
21          C[s]++
22        if (C[s] >= min_supp) then
23          F[s] =  C[s]
24      end for
25    end for
26  end proc
```

Fig.8: *Pseudo Code for SSP_LPTScan..*

4.5 Final experiment

The results of our experiments confirmed that the LPTScan can outperform the method used in [11]. The next step we need to do is compare all of our proposed algorithms to the most popular algorithm, FP-Growth. In the final experiments, we still used the same experimental settings as the previous run, the same personal computer, and same two datasets. We only created the new programs for the modified versions of the LPTScan and FP-Growth. The results are shown in figures 9 and 10.

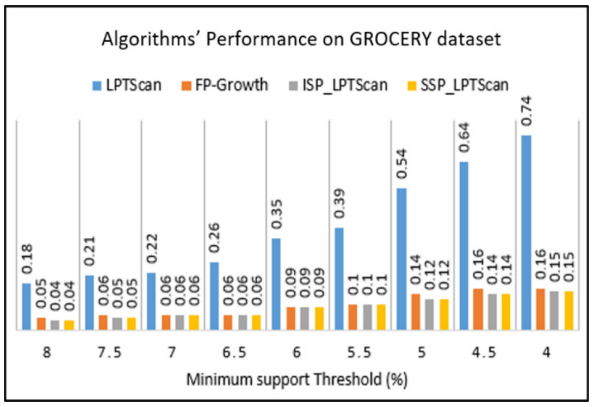

Fig.9: *CPU-times for all versions of LPTScan and FP-Growth over GROCERY.*

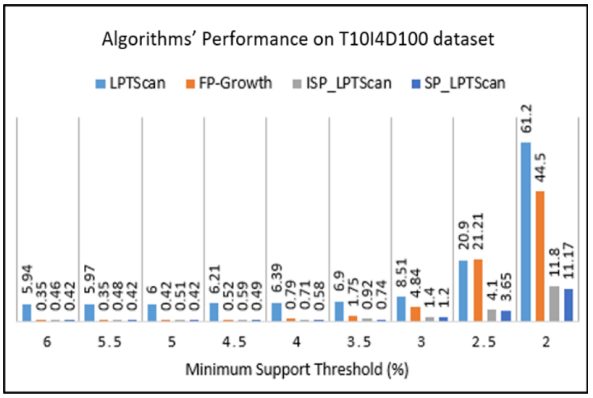

Fig.10: *CPU-times for all versions of LPTScan and FP-Growth over T10I4D100K.*

Although the bar chart shown in figure 9 clearly indicates that the LPTScan takes more CPU-time than the others, we can observe that all of them still take less than 1 millisecond. Based on the comparison among our proposed algorithms, LPTScan, ISP_LPTScan, and SSP_LPTScan, we can conclude that pruning techniques improve overall performance of the algorithms, especially the algorithms that need to remove useless item sets from the dataset before they generate candidate (1)-item sets.

In the context of execution of the algorithms over a small dataset, the four algorithm operations are not very different. However, figure 10 clearly indicates that the LPT_Scan and the FP-Growth tend to take more CPU-time when the minimum support threshold is decreased from 3% to 2%, while it gradually increased for the ISP_LPTScan and the SSP_LPTScan. Based on our experiments, we conclude that the LPTScan outperforms the work of [11], while both versions of LPTScan outperform the FP-Growth when the minimum support threshold is dropped lower and lower. However, all of the algorithms presented in our experiments still need to be further improved for more efficiency, especially in the matter of memory management.

5. CONCLUSION

The paper proposed an efficiency improvement of the Apriori algorithm because it requires scanning a dataset for multiple times in the frequent itemsets generation task. This issue leads to too much CPU-time consumtion. To speed up this task we take advantage of the power sets enumeration concept over transactions. At first, the LPTScan was created by performing (only) three repetitions of dataset scanning. We evaluated its performance against the algorithm proposed by [11] because it is the closest one to our focus/field of study. The main emphasis of our evaluation was on two important aspects of the algorithms: complexity and computational time. The results of complexity analysis indicate that the total complexity of the LPTScan is much lower than the competition. The main reason for that is that the LPTScan does not require global power sets processing. For CPU-time consumption evaluation, the results indicate that the LPTScan consumes much less CPU-time than the competition when the minimum support setting is very low. As two aspects of the evaluation were consistent, we conclude that our proposed algorithm outperforms the other. Moreover, in our study, the effectiveness of the LPTScan was extended with two new versions, the ISP_LPTScan and the SIP_LPTScan. These extensions take only three scans of the dataset and focus on the process of pruning useless item sets from the dataset before generating local power sets by transactions. Our proposed algorithms were compared with more efficient algorithms like FP-Growth. The results indicate that the LPT_Scan cannot outperform FP-Growth. However, the modified versions of it do. Although all versions of the LPTScan showed very good performance, in practice, they are not suitable for datasets with transactions composed of very large distinct item sets. The reason thay are unsuitable because the larger the transactions, the larger the local power set being generated will be. As future work, we plan to apply the extensions of the LPTSCan on larger real datasets and also to modify them so that they can use distributed and parallel computing capabilities that are currently available.

References

[1] A Rajak, MK Gupta, "Association Rule Mining: Applications in Various Areas," *Proceedings of International Conference on Data Management*, 2008.

[2] G. Serban, I. G. Czibula, and A. Campan, "A Programming Interface For Medical diagnosis Prediction," *Studia Universitatis, "Babes-Bolyai", Informatica*, LI(1), pp. 21- 30, 2006.

[3] D. Gamberger, N. Lavrac, and V. Jovanoski, "High confidence association rules for medical diagnosis," in *Proceedings of IDAMAP99*, pp. 42-51, 1999.

[4] N. Gupta, N. Mangal, K. Tiwari and P. Mitra, "Mining Quantitative Association Rules in Protein Sequences," in *Proceedings of Australasian Conference on Knowledge Discovery and Data Mining – AUSDM*, 2006.

[5] D. Malerba, F. Esposito and F.A. Lisi, Mining spatial association rules in census data, In Proceedings of Joint Conf. on "New Techniques and Technologies for Statistcs and Exchange of Technology and Know-how", 2001.

[6] G. Saporta, "Data mining and official statistics," in *Proceedings of Quinta Conferenza Nationale di Statistica*, ISTAT, Roma, 15 Nov. 2000.

[7] R. S. Chen, R. C. Wu and J. Y. Chen, "Data Mining Application in Customer Relationship Management Of Credit Card Business," in *Proceedings of 29th Annual International Computer Software and Applications Conference (COMPSAC'05)*, Volume 2, pp. 39-40,2005.

[8] H. S. Song, J. K. Kim and S. H. Kim, "Mining the change of customer behavior in an internet shopping mall," *Expert Systems with Applications*, 2001.

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

[10] R Agrawal, R Srikant, "Fast Algorithm for Mining Association Rules," *International Conference on Very large databases*, Santiago, pp. 487-499, 1994.

[11] S. Girja and B. Latita, "A New Improved Apriori Algorithm For Association rule mining," *International Journal of Engineering Research & Technology (IJERT)*, Vol. 2 (6), June 2013.

[12] B. Liu, W. Hsu, Y. Ma, "Mining association rules with multiple minimum supports," *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-99)*, San Diego, CA, USA (1999)

[13] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal, "Discovering frequent closed itemsets for association rules," in *Proc. 7th Int. Conf. Database Theory (ICDT'99)*, Jerusalem, Israel, pp.. 398–416, 1999.

[14] A. Savasere, E. Omiecinski and S. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Database," *International Conference on Very Large Data Bases*, Zurich, Switzerland, pp.. 432-443, 1995.

[15] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *ACM SIGMOD International Conference on Management of Data*, pp.1-12, 2000.

[16] J. Pavón, S. Viana & S. Gómez, "Matrix Apriori: Speeding up the Search for Frequent Patterns," in *Proceedings of the 24th IASTED international Conference on Database and Applications*, Innsbruck, Austria, pp.75-82, 2006.

[17] Yıldız, B., Ergenç, B., "Comparison of Two Association Rule Mining Algorithms without Candidate Generation," in *the 10th IASTED International Conference on Artificial Intelligence and Appliations*, Innsbruck, Austia, pp.450-457, 2010,.

[18] L. Ming-Yen L, L. Pei-Yu and H. Sue-Chen, "Apriori-based frequent item set mining algorithms on MapReduce," in *CUIMC'12*, Kuala Lumpur, Malaysia, February 20-22, 2012.

[19] C. Hemant, Y. Deepak Kumar and et, al., "MapReduce Based Frequent Item set Mining Algorithm on Stream Data," in *Global Conference on Communication Technologies*, pp. 598-603, 2015.

[20] K. Varun, D. Rajanish, "Kaal-a Real Time Stream Mining Algorithm," *43rd Hawaii International Conference on System Sciences*, 2010.

[21] S. Singh, R. Garg, and P. K Mishra, "Review of Apriori Based Algorithms on MapReduce Framework," in *Proceedings of International Conference on Communication and Computing (ICC - 2014), Elsevier Science and Technology Publications*, pp. 593–604, 2014.

Pitaya Poompuang received the B.Sc(Computer Science) from Uttaradit Rajabhat Institute, Uttaradit, Thailand in 1995, obtained the M.Sc.(Information Science) from King Mongkut's Institute of Technology Ladkrabang(KMITL), Bangkok, Thailand in 1999, and graduated the Ph.D (Information Technology in Business) from SIAM University, Bangkok Thailand in 2013. He has worked as a computer technical officer at The Comptroller General's Department Comptroller Department, Ministry of finance. Currently he is a lecturer with the Department of Computer Science, Rajamangala University of Technology Thanyaburi(RMUTT), Pathum Thani, Thailand. His researches of interest are data analytic, collective Intelligence, recommender and decision support system.

Anucha Tungkasthan received Higher Dip. Tech.(Electrical-Telecommunication) from Chiang Mai Technical College, Chiang Mai, Thailand in 1999, and the M.Sc. degree (Computer Technology) from King Mongkut's Institute of Technology North Bangkok(KMUTNB), Bangkok, Thailand in 2004. He earned the Ph.D (Information Technology in Business) from SIAM University, Bangkok Thailand in 2012. He has worked as a teacher at the Electronics Department, Suranaree Technical College. Currently he is a lecturer with the Department of Computer Technology, Rajamangala University of Technology Thanyaburi(RMUTT), Pathum Thani, Thailand. His researches of interest are digital image processing, information retrieval, CBIR system, object recognition, data analytic and machine learning.