Novel Data Segmentation Techniques for Efficient Discovery of Correlated Patterns using Parallel Algorithms

Amulya Kotni[1], R. Uday Kiran[2,3], Masashi Toyoda[2], P. Krishna Reddy[1] and Masaru Kitsuregawa[2,4]

[1]International Institute of Information Technology - Hyderabad, India
[2]Institute of Industrial Science, The University of Tokyo, Tokyo, Japan
[3]National Institute of Information and Communication Technologies, Tokyo, Japan
[4]National Institute of Informatics, Tokyo, Japan
kotni.amulya@research.iiit.ac.in, pkreddy@iiit.ac.in and {uday_rage, toyoda, kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract. Efficient discovery of interesting patterns using parallel algorithms is an actively studied topic in data mining. A key research issue related to this topic is data segmentation, which influences the overall computational requirements of an algorithm. This paper makes an effort to address this issue in correlated pattern mining. Two novel data segmentation techniques, 'database segmentation' and 'transaction segmentation,' have been introduced to discover the patterns efficiently. The former technique involves segmenting the database into multiple sub-databases such that each sub-database can be mined independently. The latter technique involves segmenting a transaction into multiple sub-transactions such that each sub-transaction can be processed as an individual transaction. The proposed techniques are algorithm independent, and therefore, can be incorporated into any parallel algorithm to find correlated patterns effectively. In this paper, we introduce map-reduce based pattern-growth algorithm by incorporating the above mentioned techniques. Experimental results demonstrate that the proposed algorithm is memory and runtime efficient and highly scalable as well.

Keywords: data mining, knowledge discovery in databases, parallel algorithms, correlated patterns and Map-Reduce

1 Introduction

Frequent pattern mining is an important model in data mining. Its mining algorithms discover all patterns in the data that satisfy the user-specified *minimum support* (*minSup*) constraint [1,2]. The popular adoption and successful industrial application of this model has been hindered by the following obstacle: *the frequent pattern model involves exponential mining space and often generates a huge number of patterns.* To confront this problem, researchers have introduced correlated pattern mining [3]. It is because the users may be interested in not only the frequent occurrences of sets of items, but also their possible strong correlations implied by such co-occurrences. The usefulness of correlated patterns was demonstrated in many applications such as climate studies, public health and bioinformatics.

In the literature, researchers have discussed several measures to assess the interestingness of a pattern. Examples include *all-confidence* [4], *any-confidence* [4], *lift* [3], *bond* [4], *h-confidence* [5] and *relative support* [6]. Each measure has a selection bias that justifies the significance of a knowledge pattern. As a result, there exists no universally acceptable best measure to judge the interestingness of a pattern for any given database or application. Researchers are making efforts to suggest a right measure depending upon the user and/or application requirements [7,8].

Recently, *all-confidence* is emerging as a popular measure to discover correlated patterns [9,10,11]. It is because this measure satisfies both the *anti-monotonic* and *null-invariance* properties. The former property says that "all non-empty subsets of a correlated pattern must also be correlated." This property plays a key role in reducing the search space, which in turn decreases the computational cost of mining the patterns. In other words, this property makes the pattern mining practicable in real-world applications. The latter property discloses genuine correlation relationships without being influenced by the object co-absence in a database. In other words, this property facilitates the user to discover interesting patterns involving both frequent and rare items without generating a huge number of uninteresting patterns. In this paper, we focus on finding correlated patterns using the *all-confidence* measure.

The basic model of correlated patterns is as follows [9]: Let $I = \{i_1, i_2, \cdots, i_n\}, n \geq 1$, be a set of items, and TDB be a database that consists of a set of transactions. Each transaction T contains a set of items such that $T \subseteq I$. Each transaction is associated with an identifier, called TID. Let $X \subseteq I$ be a set of items, referred as an itemset (or a pattern). A pattern that contains k items is a k-pattern. A transaction T is said to contain X if and only if $X \subseteq T$. The *support* of a pattern X in TDB, denoted as $S(X)$, is the number of transactions in TDB containing X. The pattern X is a **frequent pattern** if $S(X) \geq minSup$, where *minSup* represents the user-specified *minimum support*. The $all\text{-}confidence$ of a pattern X, denoted as $all\text{-}conf(X)$, can be expressed as the ratio of its *support* to the *maximum support* of an item within it. That is, $all\text{-}conf(X) = \frac{S(X)}{max(S(i_j)|\forall i_j \in X)}$. A pattern X is said to be *all-confident* or *associated* or *correlated* if $S(X) \geq minSup$ and $all\text{-}conf(X) \geq minAllConf$, where *minAllConf* represents the user-specified minimum all-confidence.

Example 1. Consider the transactional database shown in Table 1. The set of items $I = \{a,b,c,d,e,f,g,h,i,j\}$. The set of items '$a$' and '$b$,' i.e., $\{a,b\}$ (or ab) is a pattern. This pattern contains 2 items. Therefore, it is a 2-pattern. The pattern ab occurs in 7 transactions. Henceforth, the support of 'ab,' i.e., $S(ab)$ = 7. If the user-specified $minSup = 3$, then 'ab' is a frequent pattern because $S(ab) \geq minSup$. The $all\text{-}confidence$ of 'ab,' i.e., $all\text{-}conf(ab) = \frac{S(ab)}{max(S(a),S(b))} = \frac{7}{\max(10,9)}$ = 0.7. If the user-specified $minAllConf = 0.7$, then the frequent pattern 'ab' is a correlated pattern because $all\text{-}conf(ab) \geq minAllConf$.

Lee et al. [9] discussed a pattern-growth algorithm, called CoMine, to find all correlated patterns in a transactional database. Uday et al. [12] have described an improved CoMine algorithm based on the property of *items' support intervals*. This property says

Table 1: Transactional Database

TID	Items	TID	Items	TID	Items
1	*abce*	5	*abcde*	9	*abfg*
2	*ace*	6	*ab*	10	*aj*
3	*abfgh*	7	*bcdi*	11	*abcd*
4	*abf*	8	*adj*	12	*bfgh*

that each item in the database can generate correlated patterns of higher order by combining with only those items that have support within a particular interval. Both CoMine and CoMine++ are sequential algorithms, and do not exploit the availability of multiple machines or the presence of multiple cores in a machine for computation.

A Map-Reduced framework to exploit the power of thousands of machines is proposed in [13]. Encouraged by the power of Map-Reduce paradigm, researchers are making efforts to propose parallel algorithms to find frequent patterns under Map-Reduce framework [14,15]. These parallel frequent pattern mining algorithms can be extended to mine correlated patterns. However, it was revealed in our investigation that such naive algorithms are inefficient because they do not take into account the properties of *all-confidence* measure. In this paper, we make an effort to develop an efficient parallel correlated pattern mining algorithm by utilizing the properties of *all-confidence* and Map-Reduce framework.

A key research issue in developing an efficient parallel algorithm is the data segment, which influences the overall computational cost of mining the correlated patterns. Two novel data segmentation techniques, 'database segmentation' and 'transaction segmentation,' have been introduced to address this issue. The former technique involves grouping the items with respect to their *supports* and splitting the whole database into multiple sub-databases such that each sub-database can be mined independently without missing any correlated pattern. This technique enables us to efficiently distribute data across multiple machines. The second technique involves splitting a transaction into multiple sub-transactions such that each sub-transaction can be treated and processed as an independent transaction. Both techniques are based on the concept of items' support intervals, and are independent of the mining algorithm. Using the proposed techniques and Map-Reduce framework, we introduce a parallel algorithm, called Parallel Correlated Pattern-growth (PCP-growth), to find correlated patterns. Experimental results demonstrate that PCP-growth is memory and runtime efficient.

The rest of the paper is organized as follows. Section 2 describes the related work on parallel algorithms and correlated pattern mining. Section 3 introduces the proposed data segmentation techniques. Section 4 describes the proposed PCP-growth algorithm. Experimental results are reported in Section 5. Finally, Section 6 concludes the paper with future research directions

2 Related Work

2.1 Frequent pattern mining

Since the introduction of frequent patterns, several algorithms have been discussed to find these patterns efficiently [1,2,14,15]. Two popular frequent pattern mining algorithms are Apriori [1] and Frequent Pattern-growth (FP-growth) [2]. Apriori employs a candidate-generate-and-test approach to find frequent patterns, while FP-growth employs pattern-growth approach to find frequent patterns. It has been argued in the literature that FP-growth is typically better than the Apriori because the latter suffers from the performance issues such as generating huge number of candidate patterns and requiring multiple scans on the database. Both Apriori and FP-growth are sequential algorithms that suffer from memory issues when employed on very large databases.

Pramudiono et al. [16] proposed a distributed FP-Growth algorithm by taking into account a cluster of machines. Li et al. [14] proposed a Parallel FP-Growth algorithm (PFP-growth) using Map-Reduce framework. Xun et al. [15] improved the performance of PFP-growth by compressing the data into FUI-tree rather than the FP-tree. This algorithm is known as FiDoop. It has to be noted that all of the above mentioned algorithms focuses on finding frequent patterns using *minSup*. These algorithms can be extended to find correlated patterns using all-confidence measure. However, such naive algorithms are inefficient as they do not take into account the properties of all-confidence measure.

2.2 Correlated pattern mining

Brin et al. [3] introduced correlated pattern mining using $lift$ and χ^2 as the interestingness measures. Lee et al. [9] have shown that correlated patterns can be effectively discovered with all-confidence measure as it satisfies both null-invariance and downward closure properties. An FP-growth-like algorithm, called CoMine, was discussed to find the patterns. A variant of CoMine, CCMine [10], was proposed to discover confidence-closed correlated patterns. Kim et al. [11] have made an effort to discover top-k correlated patterns using the measures that satisfy the null-invariance property. Since some of the null-invariant measures (e.g. cosine) do not satisfy the anti-monotonic property, an Apriori-like algorithm was discussed to find the patterns. Uday et al. [12] discussed CoMine++ based on the concept of items' *support* intervals. This concept will be discussed in latter parts of this paper. All of the above mentioned algorithms are sequential algorithms. This paper focuses on developing parallel correlated pattern mining algorithm.

In the next section, we describe novel data segmentation techniques for distributing data across multiple machines in a network efficiently.

3 Proposed Data Segmentation Techniques

3.1 Items' Support Intervals

Uday et al. [12] introduced the property of items' support intervals to find correlated patterns effectively. This property is based on Apriori property (See Property 1) [1] and is defined as follows:

Property 1. **Apriori Property.** If $X \subset Y$, then $S(X) \geq S(Y)$.

Definition 1. ***The support interval of an item*** i_j ***(****$SI(i_j)$****).*** *An item* $i_j \in I$ *can generate correlated patterns of higher order by combining with only those items that have support within the interval* $\left[max\left(S(i_j) \times minAllConf, minSup\right), max\left(\frac{S(i_j)}{minAllConf}, minSup\right)\right]$. *Thus, the support interval of* i_j*, denoted as* $SI(i_j) = \left[max\left(S(i_j) \times minAllConf, minSup\right),\right.$ $\left.max\left(\frac{S(i_j)}{minAllConf}, minSup\right)\right]$.

The correctness of this property is given in [12]. Example 2 illustrates this property.

Example 2. In Table 1, the item 'c' has support of 5. If the user-specified $minSup = 3$ and $minAllConf = 0.7$, then the support interval of 'c' is $[4,7]$ $(= [max(3.5,3), max(7,3)])$. It means '$c$' can generate correlated patterns of higher order by combining with only those items that have *support* within the interval [4,7]. If 'c' combines with items having support not within its support interval, then Apriori property [1] causes the resulting pattern to have $allConf < minAllConf$ or $sup < minSup$ or both. For instance, 'c' cannot generate correlated pattern by combining with 'b', whose support of 9 doesn't lie within the support interval of 'c'. The reason is, $S(bc) \leq minSup$ (Apriori Property), and thus $allConf(bc) < minAllConf$. The support intervals of all frequent items in Table 1 are shown in Table 2.

Table 2: The support intervals (SI) and group ids (GI) of all frequent items in Table 1

Item	Support	SI	GI	Item	Support	SI	GI
a	10	[7,14]	1	*f*	4	[3,5]	2
b	9	[7,12]	1	*e*	3	[3,4]	2
c	5	[4,7]	2	*g*	3	[3,4]	2
d	4	[3,5]	2				

We now introduce data segmentation techniques based on the above mentioned items' support intervals. Please note that the proposed techniques are novel to this paper and have not been used in CoMine++ algorithm [12].

3.2 Data segmentation techniques

Data segmentation plays a key role in parallel algorithms. It influences: (*i*) overall memory and runtime requirements of an algorithm and (*ii*) communication cost across the machines. We now describe two data segmentation techniques to discover correlated patterns effectively.

1. Database segmentation. In the database segmentation, we initially group the items such that items within a group will not generate correlated patterns by combining with the items in other groups. Next, we split the given database into sub-databases such that each sub-database contains items of a specific group. This approach of splitting the database facilitates us to mine each sub-database independently to find correlated patterns. If there are m groups of items, then there will be m sub-databases as each sub-database contains items of a particular group. The m value depends on the distribution of items' *support* values and the user-specified *minAllConf* and *minSup* values. We now describe grouping of items.

Algorithm 1 DatabaseSegmentation(TDB, *minSup*, *minAllConf*)

```
1: Scan the database to determine the support and support interval of items. Next, sort all items
   in descending order of their support. Let $I$ denote the sorted list of items. Let $S(i_p)$ denote
   the support of an item $i$ and $p$ representing its position (or rank) in the list. $SI_{i_p}[a,b]$, $a,b \in \mathtt{R}$,
   denote the support interval of item $i_p$. Let $G(i_p)$ denote the group id of item $i_p$.
2: Set $group\_id = 1$ and $G(i_0) = 1$.
3: for $p = 0$; $p < I.size() - 1$; $++p$ do
4:     if $S(i_p) < SI_{i_{p+1}}.b$ then
5:         set $G(i_{p+1}) = group\_id$.
6:     else
7:         $++group\_id$;
8:         $G(i_{p+1}) = group\_id$.
9: Split the database with respect to items' group.
```

Definition 2. ***Group of items*** *(GI). Let* $I = \{i_1, i_2, \cdots, i_n\}$, $n \geq 1$, *be the set of sorted items such that* $S(i_1) \geq S(i_2) \geq \cdots \geq S(i_n)$. *Let* $GI \subseteq I$ *be a maximal set of items such that the support of an item* $i_p \in GI$ *lies within the support interval of its subsequent item in GI. That is,* $GI \subseteq I$, *such that if* $i_p, i_{p+1} \in GI$, *then* $S(i_p) \in SI(i_{p+1})$, $1 \leq p < (|GI| - 1)$.

Example 3. In Table 2, it can be observed that the *support* of 'a' lies within the support interval of 'b', whereas b's *support* does not lie within the support interval of c. So we consider 'a' and 'b' as one group of items. Similarly, we consider 'c', 'd','e','f' and 'g' as another group of items. Thus, the set of frequent items in Table 1 can be divided into two disjoint groups, i.e., $GI_1 = \{a, b\}$ and $GI_2 = \{c, d, e, f, g\}$. (For brevity, we have not considered infrequent items in Table 1.)

Definition 3. ***Sub-database*** *(SDB). Let* $GI_1, GI_2, \cdots, GI_m$ *be the set of disjoint group of items such that* $GI_1 \cup GI_2 \cup \cdots \cup GI_m = I$ *and* $GI_p \cap GI_q = \emptyset$, $p \neq q$ *and* $p, q \in [1, m]$. *Let* $SDB_i \subseteq TDB$, $1 \leq i \leq m$, *be the sub-database such that all items in* SDB_i *belong to a particular group* GI_i.

Example 4. Continuing with the previous example, the items' groups provide useful information that 'a' and 'b' will generate correlated patterns by combining between themselves only. The same can be said about the remaining items. Thus, the given database can be divided into two sub-databases as shown in Table 3 and Table 4. Each of these sub-databases can be mined independently to discover correlated patterns.

The two main advantages of database segmentation are: (*i*) we can eliminate mining of correlated patterns in sub-databases containing a single item and (*ii*) If a sub-database contains two items, a single scan on the entire database finds all correlated patterns. The algorithm for database segmentation is given in Algorithm 1. As the algorithm is straight forward to understand, we are not describing the algorithm.

Table 3: Sub-database 1

id	transaction	*id*	transaction	*id*	transaction
1	*ab*	5	*ab*	9	*ab*
2	*a*	6	*ab*	10	*a*
3	*ab*	7	*b*	11	*ab*
4	*ab*	8	*a*	12	*b*

Table 4: Sub-database 2

id	transaction	*id*	transaction	*id*	transaction
1	*ce*	5	*cde*	9	*fg*
2	*ce*	6	–	10	–
3	*fg*	7	*cd*	11	*cd*
4	*f*	8	*d*	12	*fg*

2. Transaction segmentation After performing database segmentation, we have distributed sub-databases to various machines to discover correlated patterns. (We used a variant of PFP-growth algorithm [14] to find correlated patterns in various machines.) During this process, we have observed that some machines were taking more time to output correlated patterns. Our investigation on the cause has revealed that the sub-databases being executed in these machines actually contained long transactions, and processing these long transactions increased the runtime. Therefore, we introduced transaction segmentation to reduce the runtime requirements of a parallel algorithm.

Definition 4. ***Transaction segmentation.*** *Let $T = \{i_1, i_2, \cdots, i_m\} \subseteq I$, $1 \leq m \leq n$, be a transaction in a (sub-)database such that $S(i_1) \geq S(i_2) \geq \cdots \geq S(i_m)$. A sub-transaction $T_1 = \{i_a, i_{a+1}, \cdots, i_b\} \subseteq T$, $1 \leq a \leq b \leq m$ is a **maximal set of items** such that $S(i_a) \in SI(i_{a+1})$. Thus, $T = T_1 \cup T_2 \cup \cdots \cup T_p$, $p \geq 1$ and $T_x \cap T_y = \emptyset$, $x, y \in [1, p]$ and $x \neq y$.*

Example 5. Consider the first transaction '*ce*' in Table 4. In this transaction, the *support* of '*c*' does not lie within the support interval of its neighboring item '*e*'. Henceforth, this transaction can be further segmented though the items in this transaction belong to the same group. In other words, the transaction '*ce*' can be split into two independent sub-transactions, '*c*' and '*e*'. Please note that the fifth transaction '*cde*' in Table 4 will not be segmented because *c*'s support lies within the support interval of its neighboring item '*d*', and *d*'s support lies within the support interval of its neighboring item '*e*'.

The procedure for transaction segmentation is given in Algorithm 2. Since the algorithm is straight forward to understand, we are not describing the algorithm.

Both techniques are algorithm independent, and therefore, can be used in any parallel algorithm (such as Apriori and FP-growth). More importantly, the proposed techniques are network independent and therefore can be used in cluster computing, grid computing and Map-Reduce framework. In this paper, we extend the proposed techniques to develop efficient algorithm for Map-Reduce framework.

4 Proposed Algorithm: PCP-growth

The PCP-growth algorithm employs Map-Reduce framework to discover correlated patterns effectively. The algorithm involves the following two steps: (*i*) finding frequent

Algorithm 2 TransactionSegmentation(*SDB*: sub-database, *SI*: support intervals of items)

```
1: Output: ST=[] be the list of all sub-transactions
2: Let $p$ = NULL
3: for each transaction $t$ in $SDB$ do
4:     Sort the items in the transaction in descending order of supports.
5:     for $q$=0; $q < t$.size()-1; ++$q$ do
6:         $p$ += $i_q$
7:         if $S(i_q) > SI_{i_{q+1}}.b$ then
8:             ST.append($p$) //Transaction $t$ is split here and the sub-transaction $p$ is added to ST
9:             $p$ = NULL
10:    $p$ += $i_{q+1}$
11:    ST.append($p$)
```

items and their support intervals and (*ii*) mining all correlated patterns by constructing FP-trees. We now discuss each of these steps.

Finding frequent items and their support intervals: First, the transactional database is divided into multiple shards (partitions) and provided to worker machines. The number of shards generally depend on the available number of machines. In the map phase, each worker machine scans the transactions one after another. For each transaction, worker machine outputs key-value pairs with key as the item and value as 1 (format is $< item, 1 >$). In the reduce phase, all the pairs with the same key obtained from all the worker machines are aggregated. Thus, supports for all the items present in the database are calculated. The final list of frequent items, called FP-list, is obtained by filtering out the items which do not satisfy the *minSup* threshold. After the master machine obtains the FP-list, the items are sorted in decreasing order of their support values and assigned ranks. The most frequent item is assigned a rank of 0, the second most frequent item is assigned a rank of 1 and so on. Next, the support intervals for the frequent items is calculated according to Definition 1. Next, items are assigned group ids such that items within a group have support within the support interval of the subsequent item. These group ids are later used for database segmentation. After this step, the master node broadcasts the items' support interval information to all the worker nodes in order to facilitate transaction segmentation in the next step.

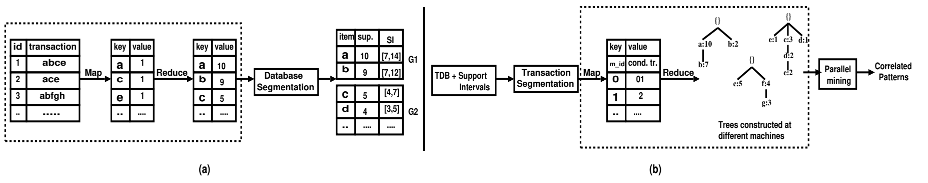

Fig. 1: Stages of PCP-Growth algorithm. (a) Assigning group ids to the items and (b) mining the patterns

Algorithm 3 Parallel FP-list Construction(TDB, $minSup$)

```
1: Procedure: Map($key = null, value = TDB_i$)
2:     for each transaction $t_{\text{cur}} \in TDB_i$ do
3:         for each item $it$ in $t_{cur}$ do
4:             Output ($it$, 1)
1: Procedure: Reduce($key = it, value = 1$)
2:     $sup = 0$
3:     for each $it$ do
4:         $sup$++
5:     Output ($it$, $sup$)
6:     if $sup \geq minSup$ then
7:         Add ($it$, $sup$) to FP-list
```

Example 6. The transactional database shown in Table 1 is divided into 3 partitions and provided to worker machines. These worker machines determine the *support* values for all items using Map-Reduce framework. Next, frequent items are generated by removing all items whose support is less than the user-specified *minSup*. The remaining frequent items are sorted in descending order of their supports. Next, the support intervals are calculated using the Definition 1. The items are now divided into groups using items' support intervals (Database Segmentation). The support intervals and group ids for all the frequent items are shown in Table 2.

Construction and Mining of FP-trees: The number of machines available in the distributed network are divided according to the ratio of number of items present in the groups. In the second database scan, for each transaction the items which are not present in the FP-list are filtered and the remaining items are sorted in descending order of supports. Now, using the transaction segmentation technique discussed in Section 3.2, we split the transaction into smaller transactions. The mining task of each item present in an item group is assigned to one of the machines assigned to that group. The sub-patterns (conditional transactions) for each sub-transaction are extracted and assigned to a machine based on a hash function. Here, item is the last item in a sub-pattern. Here, the assigned machine is stored in a hash-table for future look-up. The hash function gives a machine-id for which the pattern is responsible for further computation. The sub-patterns for every sub transaction are generated and sent to the corresponding machine. Each sub-pattern is emitted as a key-value pair, with key as the machine-id and value as sub-pattern. If a machine is responsible for many sub-patterns of the same transaction, only the longest sub-pattern is sent because the others can be derived from it. Now, the reduce function is implemented with machine-id as key, hence all the conditional transactions with same partition-id are processed at one machine. Independent local FP-trees are constructed by inserting all the sub-patterns into the tree in the same order as FP-list. The process of tree construction is same as FP-tree construction in [2]. Trees constructed on three different machines are shown in Figure 2. Since the trees are constructed from the sub-patterns itself, during conditional pattern building, communication is not required between the machines.

Algorithm 4 PCP-Growth Algorithm(TDB, FP-list, SI)

```
1: Allocate machines in the distributed network to the Item Groups.
2: **Procedure**: Map($key = null, value = TDB_i$)
3:     **for** each transaction $t_{\text{cur}} \in TDB_i$ **do**
4:         Filter the items that are not in FP-list and sort them.
5:         Segment the transaction based on SI values. Let $T'$ be the set of sub-transactions
   obtained from $t_{cur}$.
6:         **for** each sub-transaction $t'$ in $T'$ **do**
7:             **for** each item $i'$ in $t'$ **do**
8:                 machine-id = getMachineId($i'$)
1: **Procedure**: Reduce($key = machine\text{-}id, value = transactions$)
2:     Initialize FP-Tree, T
3:     **for** $t_{cur}$ in transactions **do**
4:         **for** $item$ in $t_{cur}$ **do**
5:             **if** T does not have child $item$ **then**
6:                 Create a new node and link it to the parent node
7:             Traverse to the child $item$
1: **Procedure**: Map($key = machine\text{-}id, value = FP\text{-}Tree$)
2:     **for** each suffix item $i$ in FP-Tree **do**
3:         **if** current machine-id is responsible for item $i$ **then**
4:             Generate prefix tree and conditional tree of $i$ and mine recursively.
```

Example 7. The frequent items generated in the previous step (see Table 2) are assigned ranks in descending order of their *support*. That is, the most frequent item '*a*' is assigned with rank 0, '*b*' is assigned with rank 1 and so on. Consider the first transaction '*abce*' in the first partition. Upon removing the infrequent items and sorting the remaining items in decreasing order of their supports, the transaction is converted into '*abce*'. First, we perform database segmentation and split this transaction into two sub-transactions '*ab*' and '*ce*' such that each sub-transaction belongs to a sub-database. In the sub-transaction '*ab*' the support of '*a*' lies in the support interval of '*b*'. Therefore, without performing transaction segmentation on this transaction, conditional patterns are generated and sent to the machine(s) responsible for GI_1. Now, let us consider the sub-transaction '*ce*'. In this sub-transaction although the items '*c*' and '*e*' belong to the same group, the support of '*c*' does not lie in the support interval of '*e*'(or vice versa). Therefore, we perform transaction segmentation and further split '*ce*' into sub-transactions '*c*' and '*e*'.

Let the 3 machines in the distributed network be m_0, m_1 and m_2. These machines are distributed among the item groups GI_1 and GI_2. Assume m_0 is allocated to GI_1 and m_1 and m_2 are allocated to GI_2. The sub-transactions for the transaction '*abce*' are '*ab*', '*c*' and $'e'$. Upon translating the items in the sub-transactions into their ranks, they are converted into '01', '2' and '5'. Now, for each sub-transaction, sub-patterns (conditional patterns) are generated and assigned to the machine responsible for it using a hash function. Two sub-patterns are generated for the sub-transaction '01', which are '0' and '01'. As GI_1 has only one machine allocated to it, all the sub-patterns ending with '0' or '1' are hashed to m_0. So, the key-value pairs outputted are {0:'0', 0:'01'}. Both the sub-patterns are assigned to the same machine, so only the longest sub-pattern '01' is sent as '0' can be derives from it. For the sub-transaction '2', one sub-pattern '2' is generated. As item 'c' belongs to item group GI_2 and two machines are allocated to it, the sub-pattern '2' is hashed to either m_1 or m_2 using a hash function. Similarly, for the sub-transaction '5', one sub-pattern '5' is generated and is hashed to either m_1 or m_2. Following conditional transactions are received at the m_0, $\{ab, a, ab, ab, ab, ab, b, a, ab, a, a, ab, b\}$. As per the output of the hash function used, the items 'c', 'f' and 'g' are assigned to m_1 and the items 'd' and 'e' are assigned to m_2. The following conditional transactions are received at m_1, $\{c, c, fg, f, c, c, fg, c, fg\}$. Similarly, m_2 receives the following conditional patterns, $\{e, e, cde, cd, d, cd\}$.

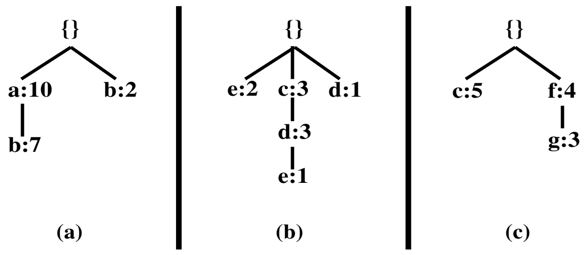

Fig. 2: (a) Tree at m_0. (b) Tree at m_1. (c) Tree at m_2

Parallel mining of correlated patterns is similar to the mining process of FP-Growth algorithm but each worker machine performs the mining process only for those suffix items for which it is responsible for computation. The prefix tree is constructed for a chosen suffix item by inserting the prefix sub paths of the nodes of the selected item. The conditional tree is constructed from the prefix tree by removing the nodes which do not satisfy the *minSup* threshold. This process is repeated for all the items assigned to each worker node. Now, all the items are translated back into their original values. Finally, the patterns extracted by all the worker nodes are gathered at the master node. Consider the mining of patterns with suffix items 'g' and 'c,' as shown in Figure 3. In the existing sequential approach, the mining process of 'c' occurs only after the mining process of 'g' and other items below 'c' in the FP-list is completed. Whereas in the proposed approach, both the processes happen in parallel.

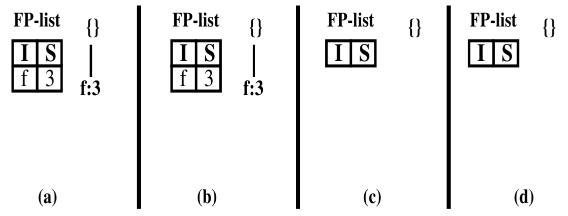

Fig. 3: Mining for suffix items 'g' and 'c' (a) Prefix tree of 'g' (b) Conditional tree of 'g' (c) Prefix tree of 'c' (d) Conditional tree of 'c'

Example 8. The mining process is started from the least frequent item, which is 'g' in the above example. The item 'g' is assigned to m_2. So, the mining of patterns with suffix 'g' is done at the tree built at m_2 i.e., Figure 2(c). The prefix tree for 'g' is constructed from this tree by removing the nodes of item 'g'. There is only one branch $< f : 3 >$ containing the item g. The final prefix tree for item 'g' is shown in Fig 3(a) . From the prefix tree, conditional tree is constructed by removing the items which do not satisfy the *minSup* threshold. The item 'f' satisfies the *minSup* threshold. The conditional tree for item 'g' is shown in Fig 3(b). Thus, from this tree the correlated patterns obtained are $\{g : 3, fg : 3\}$. Consider the item 'c'. For mining th patterns with suffix 'c', the tree at m_1 has to be mined as 'c' is assigned to it. Though the tree at m_2 has nodes containing item 'c', it need not be mined as the mining of 'c' will be done in the tree at m_1. The prefix tree of 'c' is constructed from the tree in Figure 2(b). The prefix tree of 'c' is NULL, so the conditional tree is also NULL, as shown in Fig 3(c) and (d) respectively. Therefore, the only correlated pattern generated is $\{c : 5\}$. This mining process is repeated for all the items in their respective trees and the final correlated patterns are obtained.

5 Experimental Results

In this section, we evaluate the performance of PCP-growth algorithm. The algorithm is written in Python using Apache Spark architecture and the experiments are conducted on Amazon Elastic Map-Reduce cluster, with each machine having 8GB memory. The runtime is measured in seconds and specifies the total execution time of the spark job. We conducted the experiments on both synthetic (T10I4D100K) and real-world(Retail and Online Store [17]) datasets. The T10I4D100K dataset contains 100,000 transactions with 870 items. The Retail dataset contains 88,162 transactions with 16,470 items. The Online store dataset contains 541,909 transactions with 2,603 items. For all the experiments, the *minSup* threshold for these datasets is set to 0.1%. Since there is no existing parallel algorithm for finding correlated patterns, we extend the existing PFP-growth algorithm to find correlated patterns using all-confidence measure. We call this algorithm as **naive algorithm** and compare it against the proposed PCP-growth algorithm.

Figures 4(a)-(c) show the number of correlated patterns generated in T10I4D100k, Retail and Online Store databsets respectively at different *minAllConf* values. It can be observed that with increase in *minAllConf* value, the number of correlated patterns generated decreases.

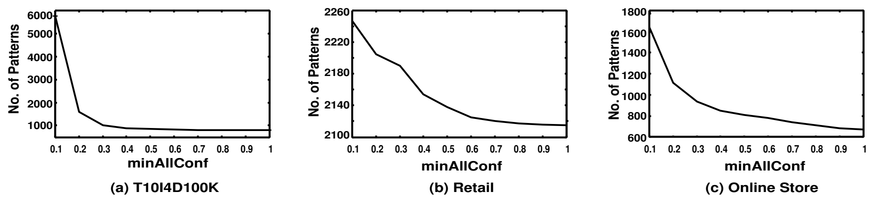

Fig. 4: Correlated patterns generated in various databases.

Fig 5 (a)-(c) show the number of sub-databases generated for various datasets at different *minAllConf* values. It can be observed that increase in *minAllConf* increases the number of sub-databases. The reason is as follows: *increase in minAllConf decreases the support-interval range for the items. The reduction of support-interval ranges for the items increases the number of sub-databases.*

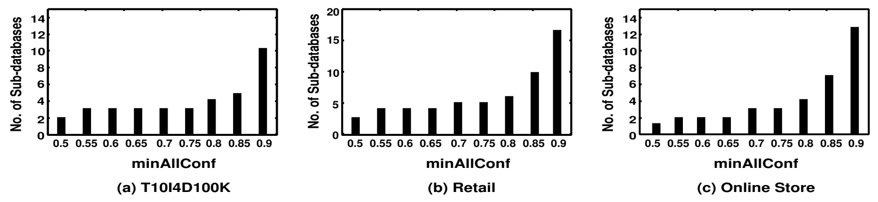

Fig. 5: Number of sub-databases formed in various databases

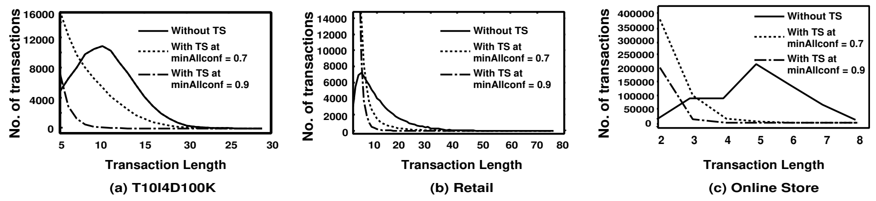

Fig. 6: Transaction length vs. Frequency of transactions for various databases

Table 5: Number of 1-length transactions. The term TS is used as an acronym for Transaction Segmentation

Dataset	Without TS	With TS at *minAllConf*=0.7	With TS at *minAllConf*=0.9
T10I4D100K	128	127,798	470,429
Retail	3016	269,114	566,261
Online Store	2635	1,238,305	1,922,804

Figures 6 (a)-(c) show the distribution of transaction lengths in various databases before and after applying transaction segmentation technique. Since number of singleton transactions (or transactions containing only one item) generated after transaction segmentation are too many, we have presented these results separately in Table 5. It can be observed that transaction segmentation has segmented many larger transactions into smaller transactions depending upon the *minAllConf* value. These shorter transactions reduce the *tree* size in PCP-growth algorithm.

Figures 7 (a)-(c) show the variation of total time consumed with the number of machines. It can be observed that increase in number of machines decreases the runtime for both naive and proposed algorithms. However, proposed algorithm was at least 50% faster than the naive algorithm.

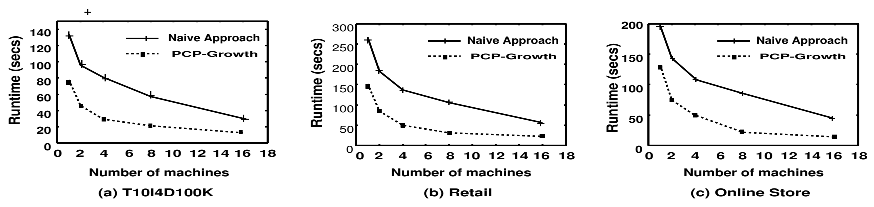

Fig. 7: Runtime vs. Number of machines for various databases

Figures 8 (a)-(c) show the amount of data shuffled among the machines. It can be observed that increase in machines increases the amount of data shuffled for both naive and proposed algorithms. However, it can be observed that the data shuffled is very less in PCP-growth algorithm.

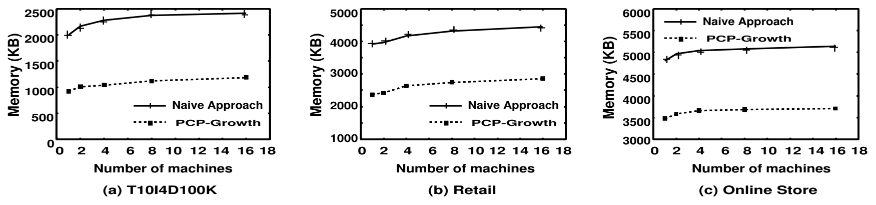

Fig. 8: Total amount of data shuffled for various databases

6 Conclusions and Future Work

Correlated pattern mining in an important model in data mining. Most of its mining algorithms are sequential algorithms. Existing parallel frequent pattern mining algorithms can be extended to mine correlated patterns. However, such a naive parallel algorithm is inadequate to discover correlated patterns effectively. It is because naive parallel algorithm cannot exploit the properties of all-confidence measure to discover the patterns effectively. In this paper, we have proposed an efficient parallel correlated pattern mining algorithm by introducing two novel data segmentation techniques. Experimental results have demonstrated that proposed algorithm is efficient.

In this paper, we have extended the proposed data segmentation techniques to Map-Reduce framework. As a part of future work, we would like extend the proposed techniques to other distributed frameworks, such as cluster and grid networks.

7 Acknowledgement

This research was partly supported by Real World Information Analytics project of National Institute of Information and Communications Technology, Japan.

References

1. R. Agarwal, R. Srikant *et al.*, “Fast algorithms for mining association rules,” in *Proc. of the 20th VLDB Conference*, 1994, pp. 487–499.
2. J. Han, J. Pei, and Y. Yin, “Mining frequent patterns without candidate generation,” in *ACM sigmod record*, vol. 29, no. 2. ACM, 2000, pp. 1–12.
3. S. Brin, R. Motwani, and C. Silverstein, “Beyond market baskets: Generalizing association rules to correlations,” in *Acm Sigmod Record*, vol. 26, no. 2. ACM, 1997, pp. 265–276.
4. E. R. Omiecinski, “Alternative interest measures for mining associations in databases,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 1, pp. 57–69, 2003.
5. H. Xiong, P.-N. Tan, and V. Kumar, “Hyperclique pattern discovery,” *Data Mining and Knowledge Discovery*, vol. 13, no. 2, pp. 219–242, 2006.
6. H. Yun, D. Ha, B. Hwang, and K. H. Ryu, “Mining association rules on significant rare data using relative support,” *Journal of Systems and Software*, vol. 67, no. 3, pp. 181–191, 2003.
7. P.-N. Tan, V. Kumar, and J. Srivastava, “Selecting the right interestingness measure for association patterns,” in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2002, pp. 32–41.
8. T. Wu, Y. Chen, and J. Han, “Re-examination of interestingness measures in pattern mining: a unified framework,” *Data Mining and Knowledge Discovery*, vol. 21, no. 3, pp. 371–397, 2010.
9. Y.-K. Lee, W.-Y. Kim, Y. D. Cai, and J. Han, “Comine: Efficient mining of correlated patterns.” in *ICDM*, vol. 3, 2003, pp. 581–584.
10. W.-Y. Kim, Y.-K. Lee, and J. Han, “Ccmine: Efficient mining of confidence-closed correlated patterns,” in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2004, pp. 569–579.
11. S. Kim, M. Barsky, and J. Han, “Efficient mining of top correlated patterns based on null-invariant measures,” in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2011, pp. 177–192.
12. R. U. Kiran and M. Kitsuregawa, “Efficient discovery of correlated patterns in transactional databases using items’ support intervals,” in *International Conference on Database and Expert Systems Applications*. Springer, 2012, pp. 234–248.
13. J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
14. H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, “Pfp: parallel fp-growth for query recommendation,” in *Proceedings of the 2008 ACM conference on Recommender systems*. ACM, 2008, pp. 107–114.
15. Y. Xun, J. Zhang, and X. Qin, “Fidoop: Parallel mining of frequent itemsets using mapreduce,” *IEEE transactions on Systems, Man, and Cybernetics: systems*, vol. 46, no. 3, pp. 313–325, 2016.
16. I. Pramudiono and M. Kitsuregawa, “Parallel fp-growth on pc cluster,” in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2003, pp. 467–473.
17. D. Chen, S. L. Sain, and K. Guo, “Data mining for the online retail industry: A case study of rfm model-based customer segmentation using data mining,” *Journal of Database Marketing & Customer Strategy Management*, vol. 19, no. 3, pp. 197–208, 2012.