Pattern Decomposition Algorithm for Data Mining Frequent Patterns

Qinghua Zou, Wesley Chu, David Johnson, Henry Chiu
Computer Science Department
University of California – Los Angeles

Abstract

Efficient algorithms to mine frequent patterns are crucial to many tasks in data mining. Since the Apriori algorithm was proposed in 1994, there have been several methods proposed to improve its performance. However, most still adopt its candidate set generation-and-test approach. In addition, many methods do not generate all frequent patterns, making them inadequate to derive association rules. We propose a pattern decomposition (PD) algorithm that can significantly reduce the size of the dataset on each pass making it more efficient to mine all frequent patterns in a large dataset. The proposed algorithm avoids the costly process of candidate set generation and saves time by reducing dataset. Our empirical evaluation shows that the algorithm outperforms Apriori by one order of magnitude and is faster than FP-tree.

1. Introduction

A fundamental problem in data mining is the process of finding frequent patterns in large datasets. This problem is further exasperated when dealing with datasets which contain highly frequent, yet often meaningful patterns (e.g., free text). While many different algorithms have been proposed, the fact remains that finding frequent patterns enables essential data mining tasks such as discovering associations relationships, correlations between data as well as finding sequential patterns [8].

Two main classes of algorithms have been proposed. The first class uses a process of candidate generation and testing to find frequent patterns. The second class of algorithms transform the original data into a representation better suited for frequent pattern mining.

1.1 Generate and Test Algorithms

Several different algorithms have been proposed to find all frequent patterns in a dataset [1, 5, 6, 7, 8]. The Apriori algorithm [1] accomplishes this by employing a bottom-up search. The algorithm generates candidate sets to limit pattern counting to only those patterns which can possibly meet the minimum support requirement. At each pass, the algorithm determines which candidates are frequent by counting their occurrence. Due to combinatory explosion, this leads to poor performance when frequent pattern sizes are large.

To avoid this problem, some algorithms output only maximal frequent patterns [2, 3, 4]. Pincer-Search [4] uses a bottom-up search along with top-down pruning to find maximal frequent patterns. Max-Miner [2] uses a heuristic bottom-up search to identify frequent sets as early as possible. Even though performance improvements may be substantial, maximal frequent sets have limited use in association rule mining. A complete set of rules cannot be extracted without support information of the subsets of those maximal frequent sets.

Algorithm in [13] partition the initial dataset into several partitions and then uses candidate set generate-and-test approach to calculate local frequent sets for each partition. The global frequent sets can be generated from counting for all local frequent sets in the whole dataset.

Other techniques have used sampling methods to select random subsets of a dataset to calculate candidate sets and then test those sets to identify frequent patterns [14, 15]. Given that the method uses sampling techniques, it is possible that some frequent patterns are not included in the candidate sets, thus the algorithm may not find all frequent patterns. In general, the accuracy of this approach is highly dependant on the data characteristic and the specific sampling technique used.

1.2 Data Transform

Most previous algorithms have used the candidate set generate-and-test approach and have mined patterns directly from an original dataset. Researchers are now exploring transforming the original data into data representations optimized for data mining. FP-tree-based mining [8] is a such an approach which first builds a compressed data representation from a dataset and then all mining tasks are performed on the FP-tree rather than on the dataset. It has performance improvements over Apriori since it uses FP-tree and does not need to generate candidate sets. However, FP-tree-based mining uses a complicated data structure and performance gains are sensitive to the support threshold.

1.3 Pattern Decomposition [16]

This paper introduces an innovative algorithm which uses pattern decomposition (PD) to mine frequent patterns. Pattern decomposition provides three significant improvements. First, by decomposing transactions into short itemsets, it is possible to combine regular patterns together, thus significantly reducing the dataset in each pass. Second, the algorithm does not need to generate candidate sets since the reduced dataset does not contain any infrequent patterns found before. Finally, using a reduced dataset greatly saves the time for counting pattern occurrence.

Pattern decomposition transforms the dataset, similar to the FP-tree algorithm. However, unlike the FP-tree algorithm, pattern decomposition does not pre-calculate the new data representation, instead the dataset is transformed only when the changes may shorten subsequent passes (e.g., decrease the number of data items to count).

2. The Method

Using candidate set generate-and-test approach, it is time consuming to count pattern occurrence since original datasets are often of huge number of transactions. The intuition of our approach is that the huge dataset need to be dramatically reduced in order to give better performance. Our algorithm is shrinking the dataset itself when new infrequent itemsets are discovered. More specifically, the PD algorithm finds frequent sets by employing a bottom-up search. For a given transaction dataset D_1, the first pass of the algorithm have two phrases. First, the algorithm counts for item occurrences to determine the frequent 1-itemsets L_1 and the infrequent 1-itemsets $\sim L_1$. Second, PD-decompose algorithm in Section 2.2 is used to decompose D_1 to get D_2 such that D_2 contains no items in $\sim L_1$. Similarly, in a subsequent pass, say pass k, consists of two phases. First, frequent itemsets L_k and $\sim L_k$ are generated by counting for all k-itemsets in D_k. Next, D_{k+1} is generated by decomposing D_k using $\sim L_k$ such that D_{k+1} contains no itemsets in $\sim L_k$.

2.1 Definitions

The terms *item* and *transaction* keep the same meaning as used in [1] where items are literals and a transaction is a set of literals in one basket. Let us define other terms as follows:

1) A **pattern** p is a pair of a set of itemsets and its occurrence, denoted by <*p.IS*, *p.Occ*>; in *p.IS*, an itemset can not be a subset of another. For example,
   p_1=<{abdef},3>, p_1.*IS*={abdef}, p_1.*Occ*=3. For short, we write p_1=abdef:3.
   p_2=<{abcd,cde},3>, p_2.*IS*={abcd,cde}, p_2.*Occ*=3. Similarly, p_2=abcd,cde:3.
   A pattern p is a **simple pattern** if *p.IS* contains only one itemset. A pattern p is a **composite pattern** if *p.IS* contains at least 2 itemsets. The size of p is the maximal size of its itemsets, denoted by *p.Len*. In the example, p_1 is a simple pattern, p_1.*Len*=5; p_2 is a composite pattern, p_2.*Len*=4.

2) A **dataset** D is a set of patterns.
   D={p: p is a pattern}
   For example, D_1 = {abc:1, abd:2, abe:1, ace:1, ade:1, bce:1, bde:1, cde:1}.
   Note that the dataset we redefined here includes pattern occurrence. The reason is that in our algorithm, we only need to consider a specific pattern once which saves computation for repeat patterns.

3) The **support** of an itemset I in a dataset D is
   Sup(I|D) = ? *p.Occ*, if p? D and (? R? *p.IS* and I? R).
   For above D_1, *Sup*(*abd*|D_1)=2, *Sup*(*ab*|D_1)=4.

The PD algorithm uses a dataset D_k on the kth pass to determine frequent itemsets L_k and infrequent itemsets $\sim L_k$. For every pattern p? D_k, PD needs to decompose its *p.IS*.

4) The **decomposition** of an itemset I with L_k and $\sim L_k$ is to find all maximal subsets S of I which does not contain any infrequent itemset in $\sim L_k$. In other words, all k-itemset of those maximal subsets are frequent in L_k. The **decomposition** of a set of itemsets R is the union of the decomposition of each itemset of R, and then removing all non-maximal itemsets that are subsets of another itemset.

5) Itemset S is said ***k*-item independent** with itemset R if the number of their common items is less than k. For example, *{1,2,3}* and *{2,3,4}* has a common set of *{2,3}*, so they are *3*-item independent, but not *2*-item independent.

Pattern decomposition rule: Given D_k, L_k, $\sim L_k$, for a pattern p? D_k, if the decomposition of *p.IS* is R={S_1,S_2,…,S_m}, and D'_k= (D_k–{p})? {(R, *p.Occ*)}, then for any frequent itemset S, *Sup*(S|D_k)=*Sup*(S|D'_k). It follows from the definition of decomposition. The rule means if we replace *p* by its decomposition result (*R*, *p.Occ*), then D_k and D'_k have the same support for any frequent itemset. This process reduces the pattern length in D_k.

After decomposition, a long pattern becomes a composite pattern with several smaller itemsets. In order to merge identical itemsets in different patterns, PD need to separate composite patterns.

Pattern separation rule: For a composite pattern *p*? D_k, $p.IS=\{S_1,S_2,\ldots,S_m\}$, if S_1 is *k*-item independent with $S_2,\ldots,S_m$ and $D'_k= (D_k-\{p\})$? $\{(\{S_1\},p.Occ)\}$? $\{(\{S_2,\ldots,S_m\}, p.Occ)\}$, then for every *k* or longer itemset *S*, $Sup(S|D_k) = Sup(S|D'_k)$. This means if we replace *p* by the two patterns $\{(\{S_1\},p.Occ)\}$ and $\{(\{S_2,\ldots,S_m\},p.Occ)\}$, then D_k and D'_k have the same support for *k* or longer itemsets. This increases the opportunity of merging identical patterns.

There are two reasons to decompose a pattern if it contains infrequent itemsets: 1) to use the infrequent itemsets $\sim L_k$ to reduce long patterns to short patterns which contain only frequent *k*-item sets, thus eliminating the need to generate candidates since PD in *k+1*th simply counts for all *k+1* itemsets in patterns of D_k; 2) to shorten a long pattern to increase the chance of merging identical patterns, and thus reducing the size of the dataset.

Let us illustrate how a pattern in the dataset is decomposed on a specific pass:

1) Suppose we are given a pattern *p*= *abcdef:1*? D_1 where *a,b,c,d,e*? L_1 and *f*? $\sim L_1$. To decompose *p* with $\sim L_1$, we simply delete *f* from *p*, leaving us with a new pattern *abcde*:1 in D_2.
2) Suppose a pattern *p*= *abcde:1*? D_2 and *ae*? $\sim L_2$. Since *ae* cannot occur in a future frequent set, we decompose *p*= *abcde:1* to a composite pattern *q*= *abcd,bcde:1* by removing *a* and *e* respectively from *p*.
3) Suppose a pattern *p*= *abcd,bcde:1* ? D_3 and *acd*? $\sim L_3$. Since *acd* ? *abcd*, *abcd* is decomposed into *abc*, *abd*, *bcd*. Their sizes are less than 4, so they are not qualified for D_4. Itemset *bcde* does not contain *acd*, so it remains the same and is included in D_4.

Now let us illustrate the complete process for mining frequent patterns. In Figure 1, we show how PD is used to find all frequent patterns in a dataset. Suppose the original data set is D_1 and minimal support is 2. We first count the support of all items in D_1 to determine L_1 and $\sim L_1$. In this case, frequent 1-

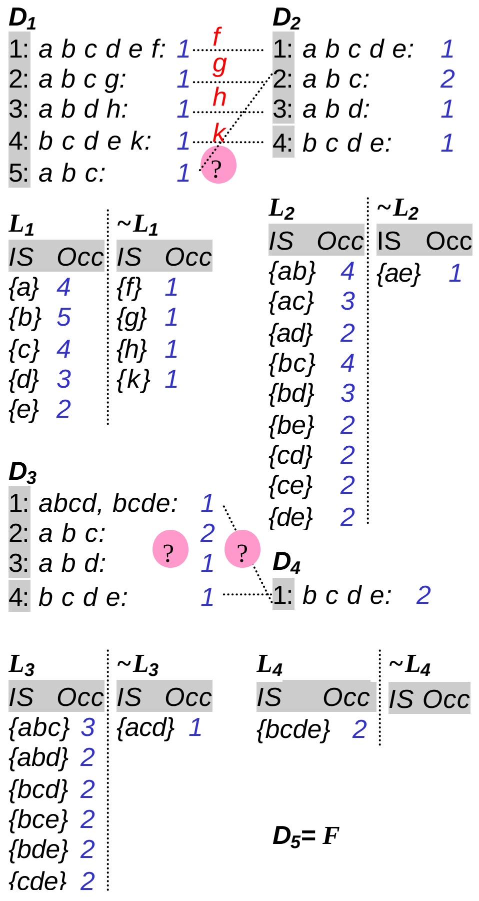

Figure 1. Pattern Decomposition Example

itemset L_1={a,b,c,d,e} and infrequent 1-itemset ~L_1={f,g,h,k}. Then we decompose each pattern in D_1 using ~L_1 to get D_2. In the second pass, we generate and count all 2-item sets contained in D_2 to determine L_2 and ~ L_2, as shown in the figure. Then we decompose each pattern in D_2 to get D_3. This continues until we determine D_5 from D_4, which is the empty set and we terminate. The final result is the union of all frequent sets L_1 through L_4.

The example illustrates three ways to reduce the dataset as denoted by ?, ?, ? in Figure 1.

In ?, when patterns after decomposition yield the same itemset, we combine them by summing their occurrence. Here, *abcg* and *abc* reduce to *abc*. Since both their occurrences are 1, the final pattern is *abc:2* in D_2.

In ?, we remove patterns if their sizes are smaller than the required size of the next dataset. Here, patterns *abc* and *abd* with sizes of 3 cannot be in D_4 and are deleted.

In ?, when a part of a given pattern has the same itemset with another pattern after decomposition, we combine them by summing their occurrence. Here, *bcde* is the itemset of pattern 4 and part of pattern 1's itemset after decomposition, so the final pattern is *bcde:2* in D_4.

Notably, the algorithm first counts for L_k and ~L_k and then decomposes patterns in each pass. It differs fundamentally from previous algorithms in that it avoids candidate set generation and reduces the dataset on each pass. Counting time is thus also reduced.

2.2. The PD-decompose Algorithm

There could be many ways to decompose a pattern. As shown above, to decompose a pattern *p* with ~L_1, we simply remove the items in ~L_1 from *p*. For a *p*? D_2, suppose *q* is its frequent 2-item sets, maximal clique techniques discussed in [4, 9, 10] can be used to calculate the decomposition result from q_2. Since the number of items in p is small and possible maximal cliques are few, those algorithms are very efficient. For k>2, no results are available to efficiently decompose a pattern. Thus a novel algorithm PD-decompose is proposed for this task.

```
PD-decompose(itemset s, ~q_k)
1: if(k=1)
2:     t = remove items in ~q_k from s
3:  else {
4:     build ordered frequency tree r;
5:     Sbs = Quick-split( r );
6:     t = mapping Sbs to itemsets;
   }
7: return t
```

Figure 2. PD-decompose

The PD-decompose algorithm is shown in Figure 2. Here, *s* is an itemset; ~q_k is the infrequent *k*-itemsets of *s*. In other words, ~q_k are *k*-item subsets of *s* that are in ~L_k. When k=1, PD-decompose simply removes the infrequent items in ~q_1 from itemset *s*. When k=2, we first build up a frequency tree from the itemsets in ~q_k. Then in step 5, we call quick-split to perform a calculation on the tree. The result is stored in *Sbs*. In step 6, we map *Sbs* back to itemsets. We give details in the following paragraphs.

One simple way to decompose the itemset *s* by an infrequent *k*-item set *t*, as explained in [4], is to replace *s* by *k* itemsets, each obtained by removing a single item in *t* from *s*. For example, for *s* = *abcdefgh* and *t* = *aef*, we decompose *s* by removing *a*, *e*, *f* respectively to obtain {*bcdefgh*, *abcdfgh*, *abcdegh*}. We call this method simple-split. When the infrequent sets are large, simple-split is not efficient. The main objective of PD-decompose is to decompose an itemset *s* by its infrequent *k* itemsets. It consists mainly of two parts: 1) building the frequency tree; 2) splitting itemsets using the tree via a method called Quick-split and returning the resulting itemsets.

A frequency tree is a tree whose nodes are items. In the tree, items at each level are ordered by the frequency of their occurrence at the level. The most frequent item at each level is placed first. A frequency tree can be constructed for a given set of infrequent *k*-item sets, $\sim q_k$. More specifically, the frequency tree for a set *t* can be built recursively as follow: 1) identifying the most frequent item *x* in *t*, let *t'*={*i*-{*x*}: *i* ? *t*, *x*? *i*} and *t*?={*i*: *i* ? *t*, *x*? *i*} 2) building a tree *r* with *x* as root item and trees from *t'* as *x*'s subtrees 3) building trees from *t*? as *x's* sibling.

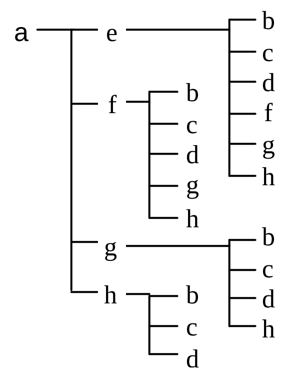

Figure 3. A frequency tree example

Example Suppose we are given a pattern *p*? D_3 where *p.IS* = *abcdefgh*. In the third pass, we find infrequent 3-itemsets {*aef*, *aeg*, *aeh*, *afg*, *afh*, *agh*, *abe*, *abf*, *abg*, *abh*, *ace*, *acf*, *acg*, *ach*, *ade*, *adf*, *adg*, *adh*}. First, we build up a frequency tree, as shown in Figure 3. The first level consists of only *a*'s. The second level consists of items *e*, *f*, *g*, and *h*, with *e* occurring the most at its level. The third level is constructed in similar fashion.

After we built a frequency tree, we then use the Quick-split technique to calculate the maximal frequent sets. The main purpose of Quick-split is to find all possible maximal frequent sets of an itemset given its infrequent *k*-itemset $\sim q_k$. In other words, Quick-split is used to find the decomposing results for an itemset.

The Quick-split algorithm is given in Figure 4. To speed up calculation, an itemset is represented by a bitset with 0 and 1 for specifying the absence or presence of an item at a corresponding position respectively. Step 1 in the Figure 4 is the exit condition. In steps 2 to 3, the subtrees of *r* are calculated and stored in an array of sub results (*subres*). The new bitset of ~x (*newBS(~x)*) returns a bitset of which all bits are 1s except that the bit corresponding to x is 0. Step 4 initializes *result* to an all 1's bitset. The results of *r*'s subtree are logically AND together to yield the final results in steps 5 to 6. Step 7 removes non-maximal itemsets and thus yields the maximal ones.

```
Quick-split(Tree r) // returns an array of BitSet
1: if(r is leaf) return Ø ;
2: forall x? r.subs do
3:     subres[x] = Quick-split(x) ?  newBS (~x)
4: result =newBS();
5: forall x? r.subs do
6:     result = result & subres[x];
7: remove b?  result, b.size= k
8: return result;
```

Figure 4. Quick-split algorithm

Quick-split performs a calculation on a frequency tree and returns an array of bitsets, which represent a group of decomposed itemsets. Splitting is accomplished by calculating bitset results in a bottom-up fashion in the tree. In the above example, we have 8 items *a, b, … , h* corresponding to positions 0-7 in a 8-bit *bitset*. So *p.IS* = *abcdefgh* = {*11111111*}; *abcd* = {*11110000*}; *bcdefgh* = {*01111111*}. The size of the bitset is the number of items in *p.IS* which is usually much smaller than the total item size in the dataset.

Table 1 shows the Quick-split splitting operations for the frequency tree in Figure 3.

Table 1: Quick-split example

Step	Results	Remarks
1	a-–e: ~b~c~d~f~g~h	From Figure 3, the leaf trees are translated into a list of ~Items. The meaning of "a-e: ~b~c~d~f~g~h" is that if a set contains **a** and **e**, then it may not contain b, c, d, f, g, and h.
	+–f: ~b~c~d~g~h	
	+–g: ~b~c~d~h	
	+–h: ~b~c~d	
2	a-–e: Ø	Total # of items is 8; the results requires # of item>=4. So maximal ~Item is 8-4=4. Thus we replace first two with Ø.
	+–f: Ø	
	+–g: ~b~c~d~h	
	+–h: ~b~c~d	
3	a: ~e (1)	The "a–e: Ø" contains two cases: "a: eØ" and "a: ~e". The first case is deleted since it contains "Ø". Other branches can be computed in the same way.
	: ~f (2)	
	: ~g	
	g~b~c~d~h	
	: ~h	
	h~b~c~d	
4	a: ~e~f (3)	From step 3, (1)&(2) yields (3).
	: ~g (4)	
	g~b~c~d~h	
	: ~h	
	h~b~c~d	
5	a: ~e~f~g (5)	From step 4, (3)&(4) yields (5).
	~e~fg~b~c~d~h	
	: ~h	
	h~b~c~d	
6	a: ~e~f~g (6)	From step 5, "~e~fg~b~c~d~h" has 6 ~Item and thus is removed.
	: ~h (7)	
	h~b~c~d	
7	a: ~e~f~g~h (8)	From step 6, (6)&(7) yields (8).
	~e~f~gh~b~c~d	
8	a: ~e~f~g~h (8)	"~e~f~gh~b~c~d" is removed.
9	:~a (9)	(8) contains two cases: "~a" and "a~e~f~g~h".
	a~e~f~g~h (10)	
10	bcdefgh	From step 9,(9)✍ bcdefgh; (10)✍ abcd.
	abcd	

As we can see from Table 1, for the itemset *abcdefgh* and infrequent 3-itemsets {*aef*, *aeg*, *aeh*, *afg*, *afh*, *agh*, *abe*, *abf*, *abg*, *abh*, *ace*, *acf*, *acg*, *ach*, *ade*, *adf*, *adg*, *adh*}, Quick_split returns the possible maximal frequent sets {*abcd*, *bcdefgh*}.

2.3. The PD Algorithm

In this section we will show the PD algorithm that uses PD-decompose to find all frequent patterns in a transaction dataset T.

As shown in Figure 5, PD is the top-level function that accepts a transaction dataset as its input and returns the union of all frequent sets as the result. At the k^{th} pass, steps 3-6 count for every k itemset of each pattern in D_k and then determine the frequent and infrequent sets, L_k and $\sim L_k$; step 7 uses D_k, L_k and $\sim L_k$ to rebuild D_{k+1}. PD stops when D_k is empty.

The PD-rebuild shown in Figure 6 is to determine D_{k+1} by D_k, L_k and $\sim L_k$. For each pattern p in D_k, step 3 computes its q_k and $\sim q_k$; step 4 calls PD-decompose algorithm to decompose p by $\sim q_k$. Note that q_k is not used here for decomposing p. As we will discuss in section 6, in some situations, using q_k to decompose p will be more efficient than using $\sim q_k$. We leave this for future research. In steps 5 to 9, we use pattern separation rule to separate p. In steps 7 to 9, PD-rebuild merges the patterns separated from p with their identical ones via a hash table ht. Since PD follows the pattern decomposition rule to decompose patterns and the pattern separation rule for merging identical patterns that yield same support, the answers generated by PD are correct.

```
PD ( transaction-set T )
1: D_1 = {<t, 1>| t ? T }; k=1;
2: while (D_k ? F ) do begin
3:     forall p ? D_k do       // counting
4:         forall k-itemset s ? p.IS do
5:             Sup(s|D_k) += p.Occ;
6:     decide L_k and ~L_k ;
                               //build D_k+1
7:     D_k+1 = PD-rebuild(D_k, L_k, ~L_k);
8:     k++;
9: end
10:Answer = ? L_k
```

Figure 5. PD

```
PD-rebuild (D_k, L_k, ~L_k)
1: D_k+1 = F ;  ht = an empty hash table;
2: forall p ? D_k do begin
3:     // q_k, ~q_k can be taken from previous counting
       q_k={s|s? p.IS n L_k }; ~q_k={t|t? p.IS n ~L_k }
4:     u = PD-decompose(p.IS, ~q_k);
5:     v ={s? u| s is k-item independent in u}
6:     add <u-v, p.Occ> to D_k+1;
7:     forall s? v do
8:         if s in ht then ht.s.Occ+= p.Occ;
9:         else put <s,p.Occ> to ht;
10: end
11: D_k+1 = D_k+1 ? {p? ht};
```

Figure 6. PD-rebuild

3. Performance Study

We compare PD with Apriori and FP-tree since the former is widely cited and the latter claims the best performance in the literature.

Our experiments were performed on a 330MHz Pentium PC machine with 128 MB main memory, running on Microsoft Windows 2000. PD algorithms were written in Java JDK1.2.2. The test data sets were generated in the same fashion as the IBM Quest project [1]. We used two data sets T10.I4.D100K denoted as D1, and T25.I10.D100K as D2. In the datasets, the number of distinct items N was set to 1000. The corruption level for a seed large itemset was fixed, obtained from a normal distribution with mean 0.5 and variance 0.1. In the first dataset, all items in a seed large itemset were corruptible while in the latter datasets half were corruptible. In the dataset D1, the average transaction size |T| and average maximal potentially frequent itemset size |I| are set to 10 and 4, respectively, while the number of transactions |D| in the dataset is set to 100K. In the dataset D2, |T|=25, |I|=10, |D|=100K.

For the comparison of PD with FP-tree, since PD was written in Java and FP-tree in C++ and we don't have time to implement PD in C++, their results are adjusted by a coefficient about 10.

3.1 Comparison of PD with Apriori

Figures 7 and 8 display our test results for datasets T10.I4.D100K and T25.I10.D100K respectively. Figure 7 shows the execution times for different minimum support. We can see that PD is about 30 times faster than Apriori with minimal support at 2% and about 10 times faster than Apriori at 0.25%.

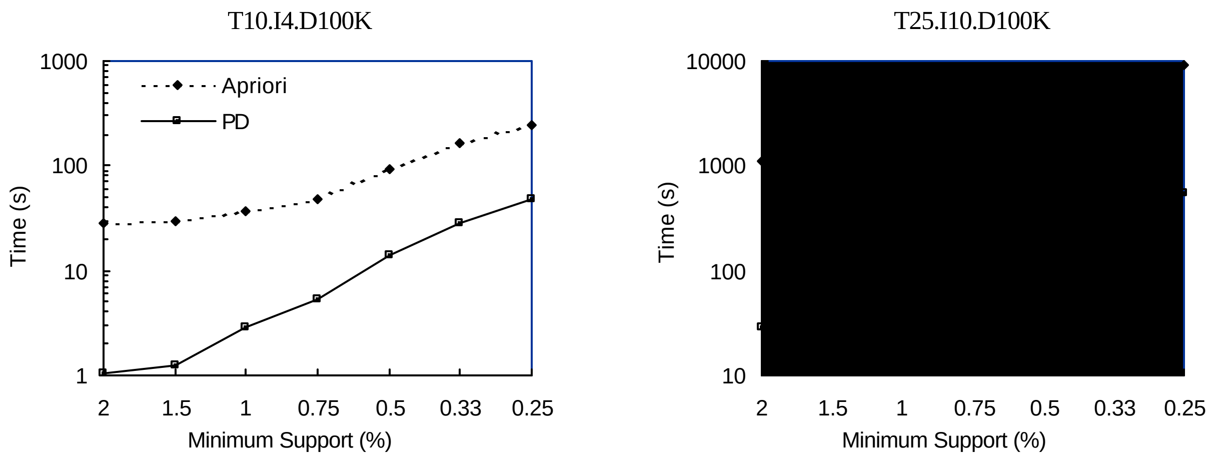

Figure 7. Execution times comparison between Apriori and PD vs. minimum support

Figure 8 shows execution times for each pass given minsup=0.25%. Initially, execution times of Apriori and PD are comparable. In later passes, when frequent sets become numerous and longer, PD outperforms Apriori. Apriori counts candidates support in the original dataset with 100K transactions with average size |T|; while PD counts in a reduced dataset with only about 5K patterns with average size much less than |T|.

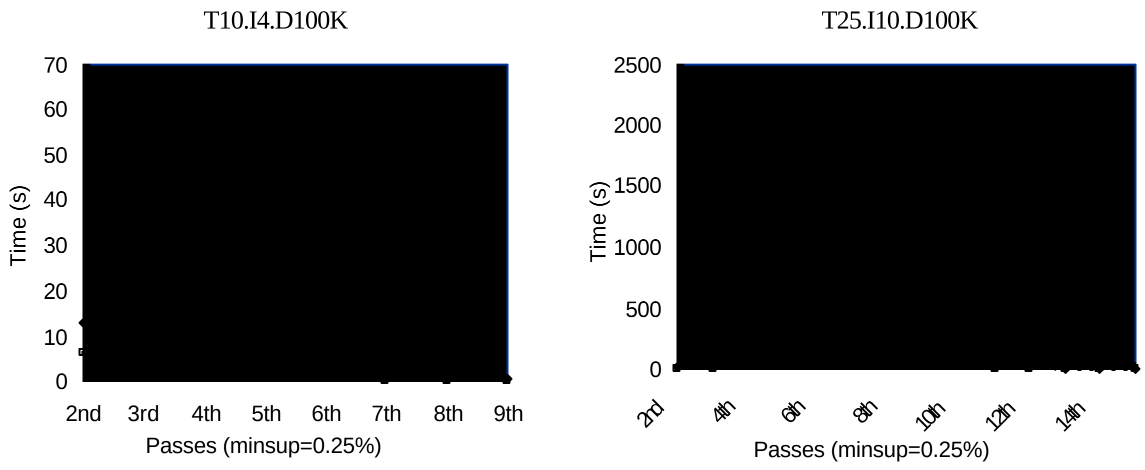

Figure 8. Execution times comparison between Apriori and PD vs. passes

To test the scalability with the number of transactions, experiments on dataset D2 are used. The support threshold is set to 0.75%. The results are presented in Figure 9. The execution time for Apriori linearly increases with the number of transactions from 50K to 250K. However, the execution time for PD does not necessarily increase as the number of transactions increases. This is due to the fact that as number of transaction |D| increases, the possibility of patterns after decomposition can combine with others increases. Suppose two datasets *D'* and *D?* have different numbers of transactions with |*D'*|>>|*D?*|; it is possible after decomposition to have $|D_1'|<|D_1?|$, i.e. a much bigger dataset after decomposition may become smaller. This means increasing the number of transactions may decrease the time for PD to mine all frequent patterns. Thus PD has a better scalability in terms of number of transactions than that of Apriori.

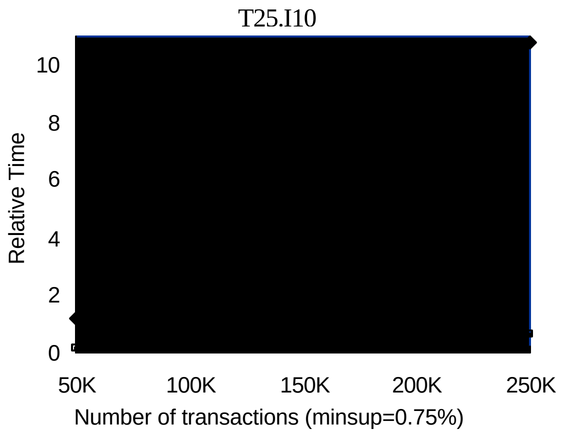

Figure 9. Scalability comparison between Apriori and PD

3.2 Comparison of PD with FP-tree

FP-tree algorithm is an efficient algorithm recently proposed in [8]. The novel idea is to build up frequent pattern trees to store data and mine frequent patterns using the trees. We note that: (1). FP-trees is substantially smaller than the original data and saves costly database scans; (2) It avoids candidate set generation and testing. For comparison, we ran the PD and FP-tree algorithms on the same machine using the same dataset as input and generated the same output. For each test point, we determined four values: (a) t_{FP} the running time for FP-tree (in C++); (b) t_{PD} the running time for PD (in Java); (c) t_{AC} the running time for Apriori (in C++); (d) t_{AJ} the running time for Apriori (in Java). To calculate the language time difference between C++ and Java, we adjusted t_{PD} to $t_{PD}*$ (t_{AC}/t_{AJ}), where t_{AC}/t_{AJ} ˜ 10. According to our experiments, both FP tree and PD were faster than Apriori, especially when the minimum support was relatively low. However, PD ran two times faster than FP-tree.

As shown in Figure 10, both FP-tree and PD have good performance on D1. But FP-tree takes substantially more time when minimum support in the range from 0.5% to 2%. When minsup less than 0.5%, the number of frequent patterns increased quickly and thus the execution times are comparable. For D2, FP-tree takes nine times longer than PD at minsup=2% and the gap reduces to 2 times faster at minsup=0.25%

In Figure 11, we compared the scalability of PD with FP-tree on the dataset D2 with minimum support=0.75%. When the number of transactions ranged from 60k-80k, both methods took almost constant time (most likely due to overhead). When we scaled up to 200K, FP-tree required more than 1884M of virtual memory and could not run on our machine while PD finished the computation within 64M main memory.

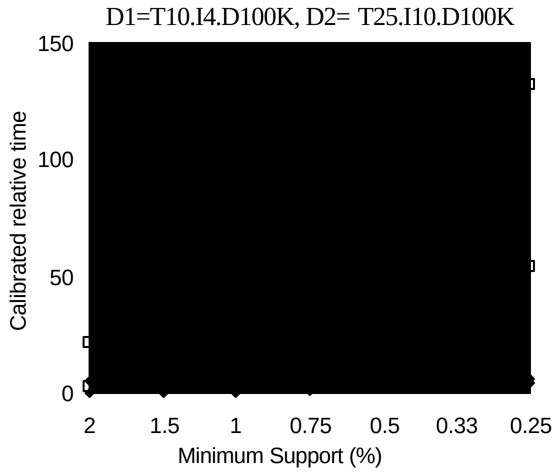

Figure 10. Performance comparison between FP-tree and PD for selective minimum support

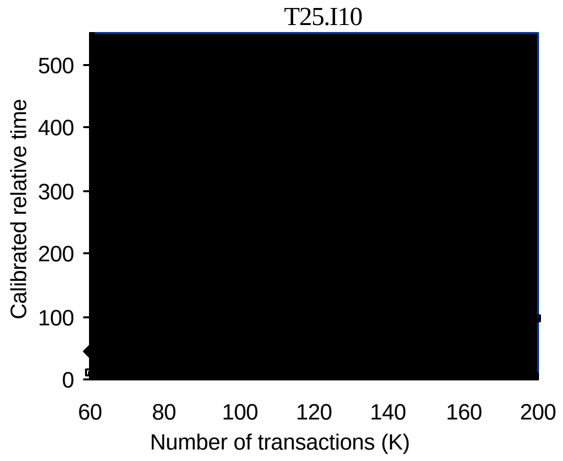

Figure 11. Scalability comparison between FP-tree and PD

The main costs in FP-tree-based mining involve recursively building conditional FP-trees. The number of conditional FP-trees can be enormous and run out of virtual memory space on our machine when we run 200K transaction dataset. Further, the complicated data structure of FP-tree requires large number of pointers. In order to build the conditional FP-tree efficiently, each node needs three pointers. Suppose the item, counter, and pointer is encoded in 4 bytes; the storage overhead of pointers in a node will be 60% of the data storage.

PD, like the FP-tree-based algorithm, uses a compressed data representation to find the frequent patterns. However, PD uses a very simple and flat data structures and significantly shrinks the dataset in each pass. PD keeps only the current dataset D_k and a hash table for pattern decomposition. Thus it requires much less storage space than FP-tree and thus yields better scalability.

4. Further Discussions

4.1 Comparison with Pincer-search

The idea of using a newly discovered infrequent set to split its supersets was discussed in Pincer-search [4]. It was reported to have performance improvements up to several orders of magnitude compared to the best algorithms at that time. Pincer-search uses both the bottom-up and top-down searches. Its primary search direction is still bottom-up, but a restricted search is also conducted in the top-down direction.

However, there are several differences between PD and Pincer-search. First, the quick-split algorithm is more efficient than the MFCS-gen used by pincer-search [4] which we call simple-split in section 3. Intuitively in quick-split, using a frequency tree saves much computation on shared items than using simple split. Second, we use quick-split to decompose a pattern of the dataset while Pincer-search uses simple-split to split candidate sets. In addition, it discovers only maximal frequent sets which do not provide enough information for generating association rules.

4.2 Further Improvements

First, we note that quick-split is not the only technique we can use for pattern decomposition. For an itemset s, suppose q_k is its frequent k-item sets and $\sim q_k$ is its infrequent k-item sets, if $| q_k | << | \sim q_k |$, one can follow that it would be more efficient to calculate decomposition results from q_k rather than from $\sim q_k$.

Second, PD is flexible in that it can be extended in various ways or applied with other algorithms. We can extend PD to output maximal frequent patterns whenever the support of a pattern in the dataset satisfies the given requirement of minimal support.

5. An Application

The motivation of our work originates from the problem of finding multi-word combinations in a group of medical report documents, where sentences can be viewed as transactions and words can be viewed as items. The problem is to find all multi-word combinations that occur at least in 2 sentences of a document.

As a simple example, Figure 12(f) shows a sample medical report. Its topic is "Aspirin greatly underused in people with heart disease". After stemming and removing stop words, there are 135 distinct words. The 34 frequent words are shown in Figure 12(a) in decreasing order of frequency. Frequent 2-word, 3-word, 4-word, 5-word combinations are listed in Figures 12(b)-(e).

Aspirin greatly underused in people with heart disease

DALLAS (AP) -- Too few heart patients are taking aspirin despite its widely known ability to prevent heart attacks, according to a study released Monday.

The study, published in the American Heart Association's journal Circulation, found that only 26 percent of patients who had heart disease and could have benefited from aspirin took the pain reliever.

"This suggests that there's a substantial number of patients who are at higher risk of more problems because they're not taking aspirin," said Dr. Randall Stafford, an internist at Harvard's Massachusetts General Hospital who led the study. "As we all know, this is a very inexpensive medication -- very affordable."

The regular use of aspirin has been shown to reduce the risk of blood clots that can block an artery and trigger a heart attack. Experts say aspirin can also reduce the risk of a stroke and angina, or severe chest pain.

Because regular aspirin use can cause some side effects -- such as stomach ulcers, internal bleeding and allergic reactions -- doctors are too often reluctant to prescribe it for heart patients, Stafford said.

"There's a bias in medicine toward treatment and within that bias we tend to underutilize preventative services -- even if they've been clearly proven," said Marty Sullivan, a professor of cardiology at Duke University in Durham, N.C.

Stafford's findings were based on 1996 data from 10,942 doctor visits by people with heart disease. The study may underestimate aspirin use; some doctors may not have reported instances in which they recommended patients take over-the-counter medications, he said.

He called the data "a wake-up call" to doctors who focus too much on acute medical problems and ignore general prevention.

(f) A sample medical report

heart, aspirin, patient, doct, study, they, risk, prevent, take, diseas, stafford, use, too, may, thi, we, attack, ther, intern, bia, gener, peopl, problem, call, know, not, pain, some, reduc, medicat, very, becaus, data, regul

(a) Frequent 1-word table (total 34)

aspirin patient, heart aspirin, aspirin use, aspirin take, aspirin risk, aspirin study, patient take, patient study, heart diseas, heart patient, diseas peopl, prevent too, they not, they ther, they take, doct data, doct some, doct too, doct use, doct stafford, aspirin regul, aspirin becaus, aspirin reduc, aspirin some, aspirin pain, aspirin not, aspirin attack, aspirin too, aspirin diseas, use regul, aspirin they, aspirin doct, stafford intern, take not, risk reduc, study take, patient becaus, patient some, patient not, patient too, patient use, patient they, patient doct, heart regul, heart peopl, heart attack, heart too, heart use, heart stafford, use some, heart study, heart doct

(b) Frequent 2-word table (total 52)

aspirin patient take, aspirin patient study, heart aspirin patient, aspirin doct some, aspirin patient some, heart aspirin use, doct use some, aspirin take not, aspirin they not, aspirin patient not, aspirin they take, aspirin study take, patient doct use, heart aspirin diseas, heart use regul, heart aspirin regul, aspirin patient too, heart aspirin attack, aspirin risk reduc, patient take not, patient they not, heart patient too, heart aspirin too, patient use some, patient doct some, patient they take, patient study take, aspirin doct use, heart doct stafford, aspirin patient use, heart diseas peopl, aspirin use regul, aspirin patient they, heart patient study, heart aspirin study, aspirin patient becaus, aspirin patient doct, aspirin use some, they take not

(c) Frequent 3-word table (total 39)

heart aspirin use regul, aspirin they take not, aspirin patient take not, patient doct use some, aspirin patient study take, patient they take not, aspirin patient use some, aspirin doct use some, aspirin patient they not, aspirin patient they take, aspirin patient doct some, heart aspirin patient too, aspirin patient doct use, heart aspirin patient study

(d) Frequent 4-word table (total 14)

aspirin patient they take not, aspirin patient doct use some

(e) Frequent 5-word table (total 2)

Figure 12. An example of multi-word combination

Multi-word combinations are effective for document indexing and summarization. The work in [12,11] shows that multi-word combinations can index documents more accurately than using single-word indexing terms. Multi-word combinations can delineate the concepts or content of a domain specific document collection more precisely than single word. For example, from the frequent 1-word table in Figure 12(a), we may infer that "*heart*", "*aspirin*", and "*patient*" are the most important concepts in the text since they occur more often than others. For the frequent 2-word table in Figure 12(b), we see a large number of 2-word combinations with "aspirin", i.e. "*aspirin patient*", "*heart aspirin*", "*aspirin use*", "*aspirin take*", etc. This infers that the document emphasizes "*aspirin*" and "*aspirin related*" topics more than any other words.

6. Conclusion

In this paper, we propose a pattern decomposition algorithm to find frequent patterns for large datasets. The PD algorithm significantly shrinks the dataset in each pass. It avoids the costly candidate set generation procedure and greatly saves counting time by using reduced datasets. Our experiments show that the PD algorithm has an order of magnitude improvement over the Apriori algorithm on standard test data and is faster than FP-tree. Since PD reduces the dataset, mining time does not necessary increase as the number of transactions increases. Experimental results reveal that PD has better scalability than both Apriori and FP-tree. We are using PD to mine multi-word combinations from medical report documents. Without an efficient technique, we otherwise need to limit the length of sentences as well as the size of multi-word combinations.

Acknowledgment

The authors wish to thank Professor Jaiwei Han, Mr. Jian Pie and Mr. Runying Mao of Simon Fraser University for providing their FP-tree software for the performance study.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In VLDB'94, pp. 487-499.

[2] R. J. Bayardo. Efficiently mining long patterns from databases. In SIGMOD'98, pp. 85-93.

[3] Zaki, M. J.; Parthasarathy, S.; Ogihara, M.; and Li, W. 1997. New Algorithms for Fast Discovery of Association Rules. In Proc. of the Third Int'l Conf. on Knowledge Discovery in Databases and Data Mining, pp. 283-286.

[4] Lin, D.-I and Kedem, Z. M. 1998. Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Set. In Proc. of the Sixth European Conf. on Extending DatabaseTechnology.

[5] Park, J. S.; Chen, M.-S.; and Yu, P. S. 1996. An Effective Hash Based Algorithm for Mining Association Rules. In Proc. of the 1995 ACM-SIGMOD Conf. on Management of Data, pp. 175-186.

[6] Brin, S.; Motwani, R.; Ullman, J.; and Tsur, S. 1997. Dynamic Itemset Counting and Implication Rules for Market Basket Data. In Proc. of the 1997 ACM-SIGMOD Conf. On Management of Data, 255-264.

[7] J. Pei, J. Han, and R. Mao. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets, Proc. 2000 ACM-SIGMOD Int. Workshop on Data Mining and Knowledge Discovery (DMKD'00), Dallas, TX, May 2000.

[8] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation, Proc. 2000 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'00), Dallas, TX, May 2000.

[9] Bomze, I. M., Budinich, M., Pardalos, P. M., and Pelillo, M. The maximum clique problem, Handbook of Combinatorial Optimization (Supplement Volume A), in D.-Z. Du and P. M. Pardalos (eds.). Kluwer Academic Publishers, Boston, MA, 1999.

[10] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph. In Communications of the ACM, 16(9):575-577, Sept. 1973.

[11] Johnson D.B., Chu W.W., Dionisio J.D.N., Taira R.K., Kangarloo H., Creating and Indexing Teaching Files from Free-text Patient Reports. Proc. AMIA Symp 1999; pp. 814-818.

[12] Johnson D.B., Chu W.W., Using n-word combinations for domain specific information retrieval, Proceedings of the Second International Conference on Information Fusion – FUSION'99, San Jose, CA, July 6-9,1999.

[13] A. Savasere, E. Omiecinski, and S. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. In Proceedings of the 21st VLDB Conference, 1995.

[14] Heikki Mannila, Hannu Toivonen, and A. Inkeri Verkamo. Efficient algorithms for discovering association rules. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, Proc. of the AAAI Workshop on Knowledge Discovery in Databases, pp. 181-192, Seattle, Washington, July 1994.

[15] H. Toivonen. Sampling Large Databases for Association Rules. In Proceedings of the 22nd International Conference on Very Large Data Bases, Bombay, India, September 1996.

[16] Q. Zou, W. Chu, D. Johnson, H. Chiu. A Pattern Decomposition (PD) Algorithm for Finding All Frequent Patterns in Large Datasets. Proc. of the IEEE International Conference on Data Mining, San Jose, California, November 2001.