Using Pattern Decomposition Methods for Finding All Frequent Patterns in Large Datasets

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 quickly reduces 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 a great amount of counting time with reduced datasets. Our empirical evaluation shows that the algorithm outperforms Apriori by one order of magnitude and is more scalable.

1. Introduction

A fundamental component in data mining tasks is finding frequent patterns in a given dataset. Frequent patterns are ones that occur at least a user-given number of times (minimum support) in the dataset. They allow us to perform essential tasks such as discovering association relationships among items, correlation, sequential pattern mining, and much more [7].

Algorithms proposed in [1, 5, 6, 9] find all frequent sets in a dataset. The Apriori algorithm [1] accomplishes this by employing a bottom-up search. It generates candidate sets starting at size 2 up to the maximal frequent set size. At each pass, it 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 sets [2, 3, 4]. Pincer-Search [4] uses a bottom-up search along with top-down pruning. Max-Miner [2] uses a bottom-up search with a heuristic to try to identify frequent sets as early as possible. Even though performance improvements may be substantial, maximal frequent sets have limited use in terms of association rule mining. A complete set of rules cannot be extracted without support information of frequent subsets.

Almost all previous algorithms use the candidate set generate-and-test approach. FP-tree-based mining [9] is an exception. It has performance improvements over Apriori since it uses a compressed data representation 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.

In this paper we propose an innovative algorithm called PD (Pattern Decomposition) that generates all frequent sets. It dynamically reduces the dataset in each pass by reducing the number of transactions and their size to give better performance. Counting time is

clearly less in a reduced dataset. In addition, the algorithm does not need to generate candidate sets; the reduced dataset contains only itemsets whose subsets are all frequent. Intuitively, a transaction that contains infrequent itemsets can be decomposed to smaller itemsets since together they do not meet the minimum support threshold. The smaller itemsets after separation are often identical with others and can be combined, thus reducing the size of the dataset.

The remainder of the paper is organized as follows. In Section 2 we introduce the PD algorithm. We show details of the algorithm in Section 3. Section 4 gives top-level algorithms for frequent pattern mining using decomposition. Section 5 compares the performance between the PD and Apriori algorithms. Section 6 discusses comparison with other techniques. Section 7 gives an application for PD. Section 8 summarizes the important points of this paper.

2. Introducing Pattern Decomposition (PD)

We begin by providing some basic definitions. The terms *transaction, itemset* and *item* keep the same meaning as in literature. We define others as follows:

1) A **pattern** p consists of an itemset and its occurrence, denoted by $p.IS$ and $p.Occ$ respectively. The size of p is the number of items in its itemset, denoted by $p.Len$. For example, if p = <{a,b,d,e,f}, 3>, then $p.IS$ = {a,b,d,e,f}, $p.Occ$ = 3, and $p.Len$ = 5. For simplicity, we write p = abdef:3.

2) A **dataset** D is a set of patterns.
   D = { p: p is a pattern }
   For example, D_0 = {abc:1, abd:2, abe:1, ace:1, ade:1, bce:1, bde:1, cde:1}

Note that the term *pattern* in the dataset differs from *transaction* in that a pattern refers to both an itemset and its occurrence.

3) The **support** of an itemset I in a dataset D is
   $Sup(I \mid D) = \Sigma p.Occ$, for all $p \in D$ and $I \subseteq p.IS$
   We call the pair (I, $Sup(I \mid D)$) the **candidate pattern** of D. If q is a candidate pattern of D, then $q.Occ = Sup(q.IS \mid D)$. For a given minimum support m, if $q.Occ \geq m$, then q is a **frequent pattern** of $\langle D, m\rangle$.
   For example, patterns *abd:2, ab:4, abcd:0* are candidate patterns of dataset D_0; *abd:3, ace:0, ad:2* are not candidate patterns of D_0. The pattern *ab:4* is a frequent pattern of $(D_0, 4)$.
4) A **composition pattern** is the pair <set of itemsets, occurrence>.
   For example, p = <{{a,b,c,d }, {b,c,d,e}}, 3>, w.r.t. p = abcd, bcde:3.

The PD algorithm uses a dataset D_k on the kth pass to determine the frequent set L_k and the infrequent set $\sim L_k$. D_k is decomposed with $\sim L_k$ to determine D_{k+1}.

There are two reasons to decompose a pattern if it contains infrequent itemsets: 1) to remove the infrequent itemsets from the pattern, thus eliminating the need to generate candidates 2) to reduce the size of the dataset. The decomposed patterns are used to build the next dataset.

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

1) Suppose we are given a pattern $p= abcdef{:}1 \in D_1$ where $a,b,c,d,e \in L_1$ and $f \in \sim L_1$. To decompose pattern 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*$\in D_2$ and $ae \in \sim L_2$. Since *ae* cannot occur in a future frequent set, we decompose $p=$ *abcde:1* to a composition pattern $q=$ *abcd,bcde:1* by removing *a* and *e* respectively from *p*.
3) Suppose a pattern $p=$ *abcd,bcde:1* $\in D_3$ and $acd \in \sim L_3$. Since $acd \subseteq$ *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 given dataset. Suppose the original data set is D_1 with minimal support of 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-itemset L_1 = {{a}, {b}, {c}, {d}, {e}} and infrequent 1-itemset $\sim L_1$ = {{f}, {g}, {h}, {k}}. Then we decompose each pattern in D_1 using $\sim L_1$ 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, shown as 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.

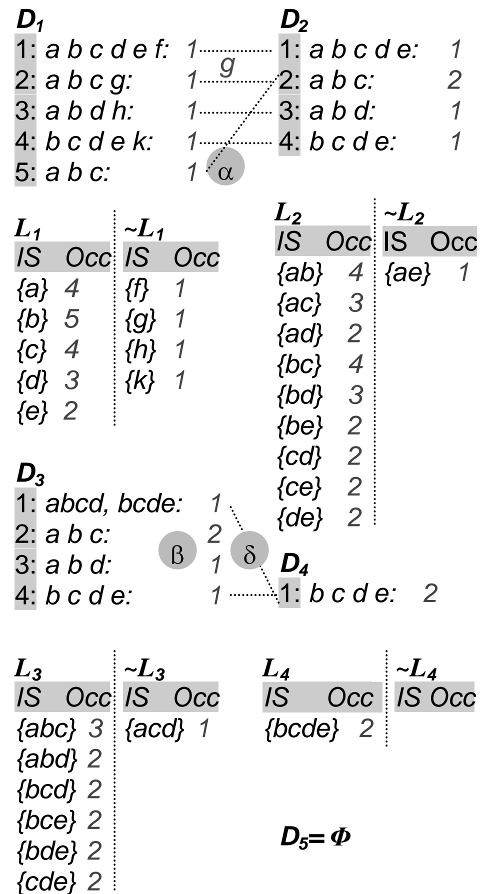

Figure 1. Pattern Decomposition Example

We notice that there are three ways to reduce the size of the dataset denoted by α, β, δ in Figure 1.
In α, 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 β, we remove patterns if their sizes are smaller than the required size of next dataset. Here, patterns *abc* and *abd* with sizes of 3 cannot be in D_4 and are deleted.

In δ, 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 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.

3. The PD-decompose Algorithm

```
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; $\sim q_k$ is the infrequent *k*-itemsets of *s* in $\sim L_k$. When k=1, PD-decompose simply removes the infrequent items in $\sim q_1$ from itemset *s*. When k≥2, we first build up a frequency tree from the itemsets in $\sim q_k$. 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.

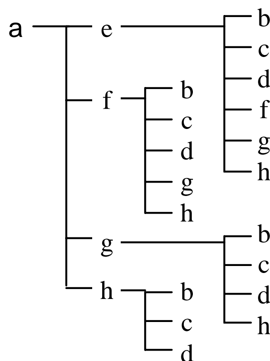

Figure 3. Frequency Tree

A frequency tree is a tree structure whose nodes on each level are ordered by the number of occurrences in the itemset. Nodes that are the most common at each level are placed first. We construct a frequency tree from the itemsets in $\sim q_k$. The trees from a set *t* is built by: 1) identifying the most common item *x* in *t*, $t'=\{i-\{x\}: i \in t, x \in i\}$, $t''=\{i: i \in t, x \notin i\}$ 2) building tree *r* with *x* as root and trees from *t′* as its subtrees 3) building trees from *t″* as *r's* brothers. The quick-split technique is then applied to the tree to return results of the split.

Example Suppose we are given a pattern $p \in D_3$ 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.

To speed up calculation, an itemset is represented by a bitset with 0 and 1 specifying the absence or presence of an item at a corresponding position respectively. 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*}.

```
Quick-split(Tree r) // returns an array of BitSet
  1: if(r is leaf) return ∅;
  2: forall x∈r.subs do
  3:     subres[x] = Quick-cal(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 is given in Figure 4. It performs a calculation on a frequency tree to get 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. We illustrate how it works with the example below.

Note here that *newBS()* is a function to get a new bitset with default value of all *1*'s. In our example, *newBS()* returns *{11111111}*. *newBS(~x)* returns a bitset of *1*'s except at the position of *x* i.e. *newBS(~b) = {10111111}*. We now begin to calculate bitsets in a bottom-up fashion. Since leaf nodes return empty bitsets, we begin at node *e* and finish at *a*.

- ***Node e***

At node *e*, there are six subtrees, each returning the empty set ∅ since they are leaves (step 1). In steps 2 and 3, we must determine the bitset *subres[x]* (subtree-result[x]) for each subtree x. For *subres[b]*, ∅ union *newBS(~b)={10111111}* is still *{10111111}*. Similary, *subres[c]={11011111},* and so on for each subtree.
At step 4, result is initialized to *{11111111}*.
Let us denote & as *bitwise AND*. In steps 5 and 6, we must perform this operation for each subtree. At the first iteration we have *{11111111} & {10111111} = {10111111}*. We take this result and perform this operation for each *subres[x]*. The final result after step 6 is *{10000000}*.
At step 7, we notice that the size (number of 1's) of *result={10000000}* is less than the frequent item size of 3. Therefore the resulting bitset is removed, and ∅ is returned.

- ***Node f***

Simlar to node *e*, ∅ is returned.

- ***Node g***

The result at step 7 is *{10001110}*. Since its size (4) is greater than the frequent item size, the result is returned.

- ***Node h***

The result *{10001111}* is returned.

- ***Node a***

We can finally compute the final resulting bitset since we have results from *a*'s subtrees.
Steps 2 and 3

$$subresult[e] = \varnothing \text{ union } newBS(\sim e) = \{11110111\}$$
$$subresult[f] = \varnothing \text{ union } newBS(\sim f) = \{11110111\}$$
$$subresult[g] = \{10001110\} \text{ union } newBS(\sim g) = \{10001110,\ 11111101\}$$
$$subresult[h] = \{10001111\} \text{ union } newBS(\sim h) = \{10001111,\ 11111110\}$$

Steps 5 and 6

$$result = \{11111111\} \ \&\ subresult[e] = \{11110111\}$$
$$result = \{11110111\} \ \&\ subresult[f] = \{11110011\}$$
$$result = \{11110011\} \ \&\ subresult[g] = \{11110011\} \ \&\ \{10001110,\ 11111101\} = \{10000010,\ 11110001\} = \{11110001\}$$
$$result = \{11110001\} \ \&\ subresult[h] = \{11110001\} \ \&\ \{10001111,\ 11111110\} = \{10000001,\ 11110000\} = \{11110000\}$$

At the final iteration, this result is combined with *newBS(~a),* giving a final result of *{11110000, 01111111}* from Quick-split. We map the bitsets back to itemsets by having *1* represent that the item at a certain position is present and 0 if it is not. The quick split result is {*abcd*, *bcdefgh*}.

4. 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.

PD in Figure 5 is the top-level function that does the counting to determine frequent and infrequent sets. It calls PD-rebuild in Figure 6 to determine D_{k+1} at each pass. PD accepts a transaction dataset as its input and returns the union of all frequent sets as the result.

```
PD ( transaction-set T )
1: D1 = {<t, 1>| t ∈ T }; k=1;
2: while (Dk≠ Φ) do begin
3:     forall p ∈ Dk do    // counting
4:         forall k-itemset s ⊆ p.IS do
5:             Sup(s|Dk) += p.Occ;
6:     decide Lk and ~Lk ;
                                  //build Dk+1
7:     Dk+1= PD-rebuild(Dk, Lk, ~Lk);
8:     k++;
9:  end
10:Answer = ∪ Lk
```

Figure 5. PD

```
PD-rebuild (Dk, Lk, ~Lk)
1: Dk+1 = Φ;  ht = an empty hash table;
2:  forall  p ∈ Dk do  begin
3:     // qk, ~qk can be taken from previous counting
       qk={s|s∈ p.IS ∩ Lk }; ~qk={t|t∈ p.IS ∩ ~Lk }
4:     u = PD-decompose(p.IS, ~qk);
5:     v ={s∈ u| s is k-item independent in u}
6:     add <u-v, p.Occ> to Dk+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: Dk+1 = Dk+1 ∪ {p∈ ht};
```

Figure 6. PD-rebuild

* itemset I_1 is **k-item independent** with I_2 if the number of their common items is less than k. e.g. {1,2,3} and {2,3,4} has a common set of {2,3}, so they are 3(and above)-item independent, but not 2-item independent. This helps avoiding duplicate counts in the hash table after decomposition.

5. Performance Comparisons with Apriori

Our experiments were performed on a 330MHz Pentium PC machine with 128 MB main memory, running on Microsoft Windows 2000. All 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 three data sets T10.I4.D100K, T20.I6.D100K, and T25.I10.D100K to compare PD with the Apriori algorithm. In the datasets, the number of items N was set to 1000. The corruption level [4] 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 two datasets half were corruptible. In the dataset T10.I4.D100K, 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 T20.I6.D100K, |T|=20, |I|=6, |D|=100K. In T25.I10.D100K, |T|=25, |I|=10, |D|=100K.

5.1 Relative Performance

Figures 7-9 show our test results for datasets T10.I4.D100K, T20.I6.D100K, and T25.I10.D100K respectively. The left graphs show the execution times for different minimum support. We can see that PD is about an order of magnitude faster than Apriori for all given minimal support. The right graphs show execution times for each pass given minsup=0.25%. Initially, execution times are comparable. In later passes when frequent sets become numerous and longer, PD clearly outperforms Apriori. The main reason is that Apriori counts candidates support in the original dataset with 100K transactions whose average size is |T|; while PD counts in a reduced dataset with about 5K patterns whose average size is much less than |T|.

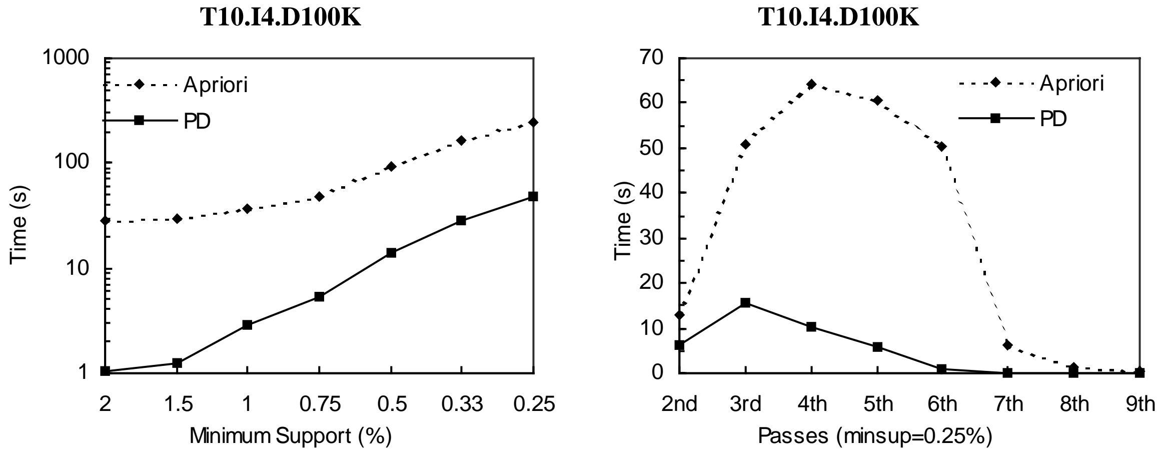

Figure 7. Execution times

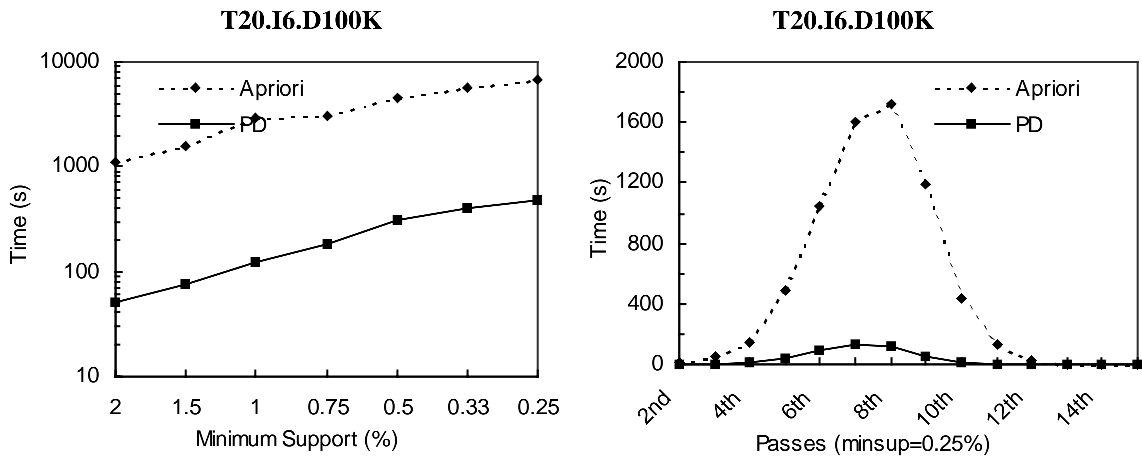

Figure 8. Execution times

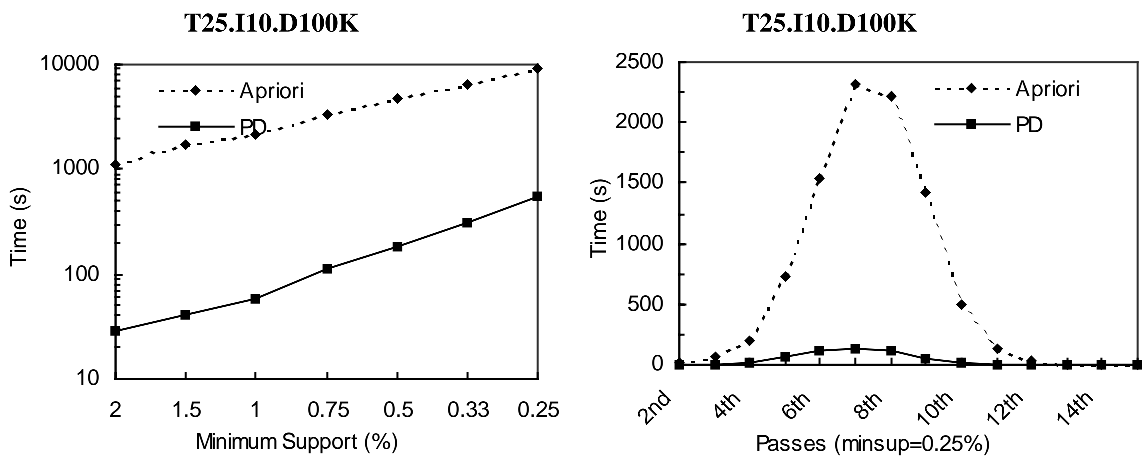

Figure 9. Execution times

5.2 Scale-up Experiment

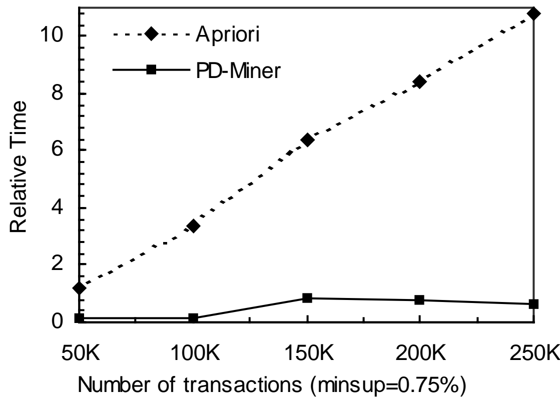

Figure 10. Number of transactions scale-up

To test the scalability with the number of transactions, experiments on dataset T25.I10.D100K are used. The support threshold is set to 0.75%. The results are presented in Figure 10. The time for Apriori algorithm linearly increases with the number of transactions from 50K to 250K. However, the execution time for PD doesn't necessary increase as the number of transactions increases. To better understand this point, one can think of many people buying the same set of items in a supermarket.

With a larger number of transactions |D|, there is a greater chance patterns after decomposition can combine with others. 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. For this reason we say PD has better scalability than Apriori.

6. Comparison and Discussion

6.1 Comparison with FP-tree-based mining

FP-tree-based mining is reported in [9] to be faster than other recent techniques in literature including TreeProjection [18]. It first builds up FP-tree and then recursively builds conditional FP-trees to mine frequent patterns. It has performance gains since it uses a compressed data representation and does not need to generate candidate sets. From [9] however, the performance improvement over Apriori is sensitive to the support threshold.

The main costs in FP-tree-based mining involve recursively building conditional FP-trees; the number of conditional FP-trees could be in the same order of magnitude as the number of frequent itemsets. Secondly, FP-tree is a complicated data structure in terms of its 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 would be 60%. Moreover, the performance gain of FP-tree-based mining could be greatly reduced if there exist a large number of random and short frequent sets since the number of common prefixes are few.

PD, like the FP-tree-based algorithm, uses a compressed data representation to find the frequent patterns. However, it uses a very simple data structure and dynamically shrinks the dataset in each pass. As shown in Figure 7-9, performance gains over Apriori are not very sensitive to support threshold on our three test datasets.

6.2 Comparison with Pincer-search

The idea of using a newly discovered infrequent set to split its supersets was independently proposed 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 in [4] which we called 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. It addition, it discovers only maximal frequent sets and cannot provide enough information to generate association rules.

6.3 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 $\sim q_k$. For k=2 when infrequent sets are often large, we use maximal clique techniques discussed in [4, 10, 11] to get s decomposition result from q_2.

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 is bigger than or equal to minimal support.

7. 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.

The work in [17] shows multi-word combinations can more accurately index documents better than using single-word indexing terms because they more precisely delineate the concepts or content of a domain specific document collection. Note that before mining multi-word combinations we need to: 1) stem, i.e. delete suffixes from each word; 2) remove stop words [17].

Figure 11(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 11(a) in decreasing order of frequency. Frequent 2-word, 3-word, 4-word, 5-word combinations are listed in Figures 11(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 11. An example of multi-word combination

Multi-word combinations have interesting properties. For example, for the frequent 1-word table in Figure 11(a), we may infer that "heart", "aspirin", and "patient" are the most important concepts in the text since they occur more often than others. When we study the frequent 2-word table in Figure 11(b), we see a large number of 2-word combinations with "aspirin", i.e. "*aspirin patient*", "*heart aspirin*", "*aspirin use*", "*aspirin take*", etc. We may infer that this document emphasizes "*aspirin*" more than any other word. Frequent 3-word, 4-word, and 5-word combinations in Figures 11(c)-(e) give increasingly better impressions of the central idea. We believe that using multi-word combinations is better than using only single words to cluster and summarize text, but we are still investigating how to use them effectively.

Using the proposed PD algorithm, we notice a performance improvement of more than two orders of magnitude over Apriori when mining multi-word combinations with more than 500 frequent words in the document and a multi-word combination size greater than

10. The reason could be that words in a sentence occur logically and not at random; and long sentences are effectively decomposed to short patterns.

8. Conclusion

In this paper, we propose a pattern decomposition algorithm to find frequent patterns for large datasets. The PD algorithm dynamically shrinks the dataset in each pass. It is efficient because 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. Since PD reduces the dataset, mining time does not necessary increase as the number of transactions increases, and experiments do show that PD has better scalability than Apriori. We are successfully using PD to mine multi-word combinations from medical report documents. Without an efficient technique, we would otherwise need to limit the length of sentences.

Acknowledgment

Removed for double blind review.

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, 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, 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 (PDF) '', Proc. 2000 ACM-SIGMOD Int. Workshop on Data Mining and Knowledge Discovery (DMKD'00)}, Dallas, TX, May 2000.
[8] K. Wang, Y. He and J. Han, `` Mining Frequent Itemsets Using Support Constraints '', Proc. Int. Conf. on on Very Large Data Bases (VLDB'00), Cairo, Egypt, Sept. 2000.
[9] J. Han, J. Pei, and Y. Yin, `` Mining Frequent Patterns without Candidate Generation (PDF)'', (Slides), Proc. 2000 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'00), Dallas, TX, May 2000.
[10] 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.

[11] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph. In Communications of the ACM, 16(9):575-577, Sept. 1973.
[12] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. Depth-first generation of large itemsets for association rules. IBM Tech. Report RC21538, July 1999.
[13] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. In J. Parallel and Distributed Computing, 2000.
[14] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo, "The maximum clique problem", in D.-Z. Du and P. M. Pardalos (Eds.), Handbook of Combinatorial Optimization (Supplement Volume A), Kluwer Academic Publishers, Boston, MA, 1999.
[15] *Removed for double blind review.*
[16] T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, The MIT Press 1993.
[17] *Removed for double blind review.*
[18] R.Agarwal, C.Aggarwal, and V. V. V. Prasad. A Tree projection algorithm for generation of frequent itemsets. In J. Parallel and Distributed Computing, 2000