Mining Frequent Itemsets from Secondary Memory

Gösta Grahne and Jianfei Zhu
Concordia University, Montreal, Canada
{grahne, j_zhu}@cs.concordia.ca

Abstract

Mining frequent itemsets is at the core of mining association rules, and is by now quite well understood algorithmically for main memory databases. In this paper, we investigate approaches to mining frequent itemsets when the database or the data structures used in the mining are too large to fit in main memory. Experimental results show that our techniques reduce the required disk accesses by orders of magnitude, and enable truly scalable data mining.

1 Introduction

Mining frequent itemsets is a fundamental problem for mining association rules [2, 3, 5]. It also plays an important role in many other data mining tasks such as sequential patterns, episodes, multi-dimensional patterns and so on [4, 9, 8]. In addition, frequent itemsets are one of the key abstractions in data mining.

The description of the problem is as follows. Let $I = \{i_1, i_2, \ldots, i_n\}$, be a set of *items*. Items will sometimes also be denoted by $a, b, c, \ldots$. An I-*transaction* τ is a subset of I. An I-transactional *database* $\mathcal{D}$ is a finite bag of I-transactions. The *support* of an itemset $S \subseteq I$ is the proportion of transactions in $\mathcal{D}$ that contain S. The task of mining frequent itemsets is to find all S such that the support of S is greater than some given *minimum support* ξ, where ξ either is a fraction in $[0, 1]$, or an absolute count.

Most of the algorithms, such as Apriori [3], DepthProject [1], and dEclat [12] work well when the main memory is big enough to fit the whole database or/and the data structures (candidate sets, FP-trees, etc). When a database is very large or when the minimum support is very low, either the data structures used by the algorithms may not be accommodated in main memory, or the algorithms spend too much time on multiple passes over the database. In the *First IEEE ICDM Workshop on Frequent Itemset Mining Implementations, FIMI '03* [5], many well known algorithms were implemented and independently tested. The results show that "*none* of the algorithms is able to gracefully scale-up to very large datasets, with millions of transactions" [5].

At the same time very large databases do exist in real life. In a medium sized business or in a company big as Walmart, it's very easy to collect a few gigabytes of data. Terabytes of raw data are ubiquitously being recorded in commerce, science and government. The question of how to handle these databases is still one of the most difficult problems in data mining.

In this paper we consider the problem of mining frequent itemsets from *very* large databases. We adopt a divide-and-conquer approach. First we give three algorithms, the general divide-and-conquer algorithm, then an algorithm using basic projection (division), and an algorithm using aggressive projection. We also analyze the disk I/O's required by these algorithms.

In a detailed divide-and-conquer algorithm, called *Diskmine*, we use the highly efficient *FPgrowth** method [6] to mine frequent itemsets from an FP-tree for the main memory part of data mining. We describe several novel techniques useful in mining frequent itemsets from disks, such as the *FP-array* technique, and the *item-grouping* technique.

We also present experimental results that demonstrate the fact that our *Diskmine*-algorithm can outperform previous algorithms by orders of magnitude, and scales up to terabytes of data.

2 Mining from disk

How should one go about when mining frequent itemsets from very large databases residing in a secondary memory storage, such as disks? Here "very large" means that the data structures constructed from the database for mining frequent itemsets can not fit in the available main memory.

One approach is *sampling* [11]. Unfortunately, the results of sampling are probabilistic, some critical frequent itemsets could be missing. Besides the sampling, there are basically two strategies for mining frequent itemsets, the datastructures approach, and the divide-and-conquer approach.

The *datastructures* approach consists of reading the database buffer by buffer, and generate datastructures (i.e. candidate sets or FP-trees). Since the datastructure do not fit into main memory, additional disk I/O's are required. The number of passes and disk I/O's required by the approach depend on the algorithm and its datastructures. For examples, if the algorithm is Apriori [3] using a hash-tree

for candidate itemsets, disk based hash-trees have to be used. If the algorithm is FP-growth method, as suggested in [7], FP-trees have to be written to the disk. Then the number of disk I/O's for the trees depends on the size of the trees on disk. Note that the size of the trees could be the same as or even bigger than the size of the database.

The basic strategy for the *divide-and-conquer* approach is shown in the procedure *diskmine*. In the approach, $|\mathcal{D}|$ denotes the size of the data structures used by the mining algorithm, and M is the size of available main memory. Function *mainmine* is called if candidate frequent itemsets (not necessary all) can be mined without writing the data structures used by a mining algorithm to disks. In *diskmine*, a very large database is decomposed into a number of smaller databases. If a "small" database is still too large, i.e, the data structures are still too big to fit in main memory, the decomposition is recursively continued until the data structures fit in main memory. After all small databases are processed, all candidate frequent itemsets are combined in some way (obviously depending on the way the decomposition was done) to get all frequent itemsets for the original database.

Procedure *diskmine*($\mathcal{D}, M$)
if $|\mathcal{D}| \leq M$ **then return** *mainmine*($\mathcal{D}$)
else decompose $\mathcal{D}$ into $\mathcal{D}_1, \ldots \mathcal{D}_k$.
return *combine diskmine*($\mathcal{D}_1, M$), ... , *diskmine*($\mathcal{D}_k, M$).

The efficiency of *diskmine* depends on the method used for mining frequent itemsets in main memory and on the number of disk I/O's needed in the decomposition and combination phases. Sometimes the disk I/O is the main factor. Since the decomposition step involves I/O, ideally the number of recursive calls should be kept small. The faster we can obtain small decomposed databases, the fewer recursive call we will need. On the other hand, if a decomposition cuts down the size of the projected databases drastically, the trade-off might be that the combination step becomes more complicated and might involve heavy disk I/O.

In the following we discuss two decomposition strategies, namely decomposition by partition, and decomposition by projection.

Partitioning [10] is an approach in which a large database is decomposed into cells of small non-overlapping databases. The cell-size is chosen so that all frequent itemsets in a cell can be mined without having to store any data structures in secondary memory. However, since a cell only contains partial frequency information of the original database, all frequent itemsets from the cell are local to that cell of the partition, and could only be *candidate* frequent itemsets for the whole database. Thus the candidate frequent itemsets mined from a cell have to be verified later to filter out false hits. Consequently, those candidate sets have to be written to disk in order to leave space for processing the next cell of the partition. After generating candidate frequent itemsets from all cells, another database scan is needed to filter out all infrequent itemsets. The partition approach therefore needs only two passes over the database, but writing and reading candidate frequent itemsets will involve a significant number of disk I/O's, depending on the size of the set of candidate frequent itemsets.

We can conclude that the partition approach to decomposition keeps the recursive levels down to one, but the penalty is that the combination phase becomes expensive.

To get an easier combination phase, we adopt another decomposition strategy, which we call *projection*. This approach projects the original database on several databases, each determined by one or more frequent item(s). One advantage of this approach is that any frequent itemset mined from a projected database is a frequent itemset in the original database. To get *all* frequent itemsets, we only need to take the union of the frequent itemsets discovered in the small projected databases. The biggest problem of the projection approach is that the total size of the projected databases could be too large, and there could be too many disk I/O's for the projected databases. Thus, there is a tradeoff between the easier combination phase and possible too many disk I/O's.

To analyze the recurrence and required disk I/O's of the general divide-and-conquer algorithm when the decomposition strategy is projection, let us suppose that:

- The original database size is D bytes.
- The data structure is an FP-tree.
- The FP-tree constructed from original database $\mathcal{D}$ is T, and its size is $|T|$ bytes.
- If a conditional FP-tree T' is constructed from an FP-tree T, then $|T'| \leq c \cdot |T|$, for some constant $c < 1$.
- The main memory mining method is the *FP-growth* method [7]. Two database scans are needed for constructing an FP-tree from a database.
- The block size is B bytes.
- The main memory available is M bytes.

In the first line of the algorithm *diskmine*, if T can not fit in memory, then projected databases will be generated. We assumed that the size of the FP-tree for a projected database is $c \cdot |T|$. If $c \cdot |T| \leq M$, function *mainmine* can be called for the projected database, otherwise, the decomposition goes on. At pass m, the size of the FP-tree constructed from a projected database is $c^m \cdot |T|$. Thus, the number of passes needed by the divide-and-conquer projection algorithm is $1 + \lceil \log_c M/T \rceil$. Based on our experience and the analysis in [7], we can say that for all practical purposes the number of passes will be at most two. For example, Let $D = 100$ gigabytes, $T = 10$ gigabytes, $M = 1$ gigabytes, and $c = 10\%$. Then the number of passes is $1 + \lceil \log_{0.1} 2^{30}/(10 \times 2^{30}) \rceil = 2$. In five passes we can handle databases up to 100 Terabytes. Namely, we get $1 + \lceil \log_{0.1} 2^{30}/(10 \times 2^{40}) \rceil = 5$.

Assume that there are two passes, and that the sum of the sizes of all projected databases is D'. After the first

database scan for finding all frequent single items, the second database scan attempts to construct an FP-tree from the database. If the main memory is not big enough, the scan will be aborted. We assume on average half of $\mathcal{D}$ is read at this stage, which means $1/2 \cdot D/B$ disk I/O's. The third scan is for decomposition. Totally, there are $5/2 \times D/B$ disk I/O's. The projected databases have to be written to the disks first, then later each scanned twice for building the FP-tree. This step needs $3 \times D'/B$ disk I/O's. Thus, the total disk number of disk I/O's for the general divide-and-conquer projection algorithm is at least

$$5/2 \cdot D/B + 3 \cdot D'/B. \quad (1)$$

Obviously, the smaller D', the better the performance.

One of the simplest projection strategies is to project the database on each frequent item, which we call *basic projection*. First we need some formal definitions.

Definition 1 Let I be a set of items. By I^* we will denote *strings* over I, such that each symbol occurs at most once in the string. If α, β are strings, then $\alpha.\beta$ denotes the concatenation of the string α with the string β.

Let $\mathcal{D}$ be an I-database. Then $freqstring(\mathcal{D})$ is the string over I, such that each frequent item in $\mathcal{D}$ occurs in it exactly once, and the items are in decreasing order of frequency in $\mathcal{D}$. ■

As an example, consider the $\{a,b,c,d,e\}$-database $\mathcal{D} = \{\{a,c,d\},\{b,c,d,e\},\{a,b\},\{a,c\}\}$. If the minimum support is 50%, then $freqstring(\mathcal{D}) = acbd$.

Definition 2 Let $\mathcal{D}$ be an I-database, and let $freqstring(\mathcal{D}) = i_1 i_2 \cdots i_k$. For $j \in \{1,\ldots,k\}$ we define $\mathcal{D}_{i_j} = \{\tau \cap \{i_1,\ldots,i_j\} : i_j \in \tau, \tau \in \mathcal{D}\}$.

Let $\alpha \in I^*$. We define $\mathcal{D}_\alpha$ inductively: $\mathcal{D}_\epsilon = \mathcal{D}$, and let $freqstring(\mathcal{D}_\alpha) = i_1 i_2 \cdots i_k$. Then, for $j \in \{1,\ldots,k\}$, $\mathcal{D}_{\alpha.i_j} = \{\tau \cap \{i_1,\ldots,i_j\} : i_j \in \tau, \tau \in \mathcal{D}_\alpha\}$. ■

Obviously, $\mathcal{D}_{\alpha.i_j}$ is an $\{i_1,\ldots,i_j\}$-database. The decomposition of $\mathcal{D}_\alpha$ into $\mathcal{D}_{\alpha.i_1},\ldots,\mathcal{D}_{\alpha.i_k}$ is called the *basic projection*.

To illustrate the basic projection, let's consider the above example, starting from the least frequent item in the *freqstring*, we obtain $\mathcal{D}_d = \{\{a,c,d\},\{b,c,d\}\}$, $\mathcal{D}_b = \{\{c,b\},\{a,b\}\}$, $\mathcal{D}_c = \{\{a,c\},\{c\},\{a,c\}\}$, and $\mathcal{D}_a = \{\{a\},\{a\},\{a\}\}$.

Definition 3 Let $\alpha \in I^*$, $i_j \in I$, and let $\mathcal{D}_{\alpha.i_j}$ be an I-database. Then $freqsets(\xi, \mathcal{D}_{\alpha.i_j})$ denotes the subsets of I that contain i_j and are frequent in $\mathcal{D}_{\alpha.i_j}$ when the minimum support is ξ. Usually, we shall abstract ξ away, and write just $freqsets(\mathcal{D}_{\alpha.i_j})$. ■

Lemma 1 *Let $\mathcal{D}_\alpha$ be an I-database, and $freqstring(\mathcal{D}_\alpha) = i_1 i_2 \cdots i_k$. Then*

$$freqsets(\mathcal{D}_\alpha) = \bigcup_{j \in \{1,\ldots,k\}} freqsets(\mathcal{D}_{\alpha.i_j}) \quad ■$$

In the previous example, for $\mathcal{D}_d$, *freqsets*($\mathcal{D}_d$)=$\{\{d\},\{c,d\}\}$. Note though $\{c\}$ is also frequent in $\mathcal{D}_d$, it is not listed since it does not contain d. It will be listed in *freqsets*($\mathcal{D}_c$). Similarly, *freqsets*($\mathcal{D}_b$)=$\{\{b\}\}$, *freqsets*($\mathcal{D}_c$)=$\{\{c\},\{a,c\}\}$ and *freqsets*($\mathcal{D}_a$)=$\{\{a\}\}$. We also can notice that $\mathcal{D}_d$ and $\mathcal{D}_c$ are not that much smaller than the original database. The upside is though that the set of all frequent itemsets in $\mathcal{D}$ now simply is the union of *freqsets*($\mathcal{D}_d$), *freqsets*($\mathcal{D}_b$), *freqsets*($\mathcal{D}_c$) and *freqsets*($\mathcal{D}_d$). This means that the combination phase is a simple union.

The following procedure *basicdiskmine* gives a divide-and-conquer algorithm that uses basic projection. A transaction τ in $\mathcal{D}_\alpha$ will be partly inserted into $\mathcal{D}_{\alpha.i_j}$ if and only if τ contains i_j. The parallel projection algorithm introduced in [7] is an algorithm of this kind.

Procedure *basicdiskmine*($\mathcal{D}_\alpha, M$)
if $|\mathcal{D}_\alpha| \le M$ **then return** *mainmine(* $\mathcal{D}_\alpha$*)*
else let $freqstring(\mathcal{D}_\alpha) = i_1 i_2 \cdots i_n$,
return *basicdiskmine*($\mathcal{D}_{\alpha.i_1}, M$) $\cup$
$\ldots \cup$
basicdiskmine($\mathcal{D}_{\alpha.i_n}, M$).

Let's analyze the disk I/O's of the algorithm *basicdiskmine*. As before, we assume that there are two passes, that the data structure is an FP-tree, and that the main memory mining method is *FP-growth*. If in $\mathcal{D}_\epsilon$, each transaction contains on the average n frequent items, each transaction will be written to n projected databases. Thus the total length of the associated transactions in the projected databases is $n+(n-1)+\cdots+1 = n(n+1)/2$, the total size of all projected databases is $(n+1)/2 \cdot D \approx n/2 \cdot D$.

Still there are two full database scans and a incomplete database scan for $\mathcal{D}_\epsilon$, as explained for formula (1). The number of total disk I/O's is $5/2 \cdot D/B$. The projected databases have to be written to the disks first, then later scanned twice each for building an FP-tree. This step needs at least $3 \cdot n/2 \times D/B$. Thus, the total disk I/O's for the divide-and-conquer algorithm with basic projection is

$$5/2 \cdot D/B + n \cdot 3/2 \cdot D/B \quad (2)$$

The recurrence structure of *basicdiskmine* is shown in Figure 1. The reader should ignore nodes in the shaded area at this point, they represent processing in main memory.

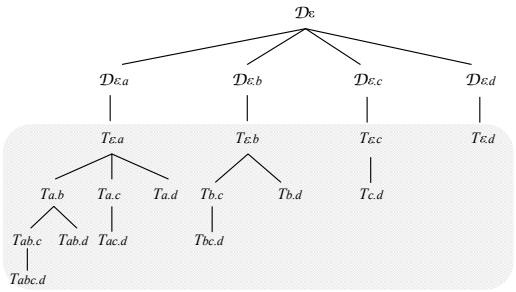


Figure 1. Recurrence of Basic Projection

In a typical application n, the average number of frequent items could be hundreds, or thousands. It therefore makes sense to devise a smarter projection strategy. Before we go further, we introduce some definitions and a lemma.

Definition 4 Let $\mathcal{D}_\alpha$ be an I-database, and let $freqstring(\mathcal{D}_\alpha) = \beta_1.\beta_2.\cdots.\beta_k$, where each β_j is a string in I^*. We call $\beta_1.\beta_2.\cdots.\beta_k$ a *grouping* of $freqstring(\mathcal{D}_\alpha)$. Let $\beta_j = i_{j_1}.\ldots.i_{j_m}$, for $j \in \{1,\ldots,k\}$. We now define $\mathcal{D}_{\alpha.\beta_j} =$

$$\{\tau \cap \{i_{1_1},\ldots,i_{j_m}\}, \tau \in \mathcal{D}_\alpha, \tau \cap \{i_{j_1},\ldots,i_{j_m}\} \neq \emptyset\}.$$

In $\mathcal{D}_{\alpha.\beta_j}$, items in β_j are called *master items*, items in $\beta_1,\ldots,\beta_{j-1}$ are called *slave items*. ■

For the previous example, $freqstring(\mathcal{D}_\alpha) = acbd$, $\beta_1 = ac$, $\beta_2 = bd$ gives the grouping $ac.bd$ of $acbd$. Now $\mathcal{D}_{bd} = \{\{a,c,d\},\{b,c,d\},\{a,b\}\}$ and $\mathcal{D}_{ac} = \{\{a,c\},\{c\},\{a\},\{a,c\}\}$.

Definition 5 Let $\{\alpha,\beta\} \subset I^*$, and let $\mathcal{D}_{\alpha.\beta}$ be an I-database. Then $freqsets(\mathcal{D}_{\alpha.\beta})$ denotes the subsets of I that contain at least one item in β and are frequent in $\mathcal{D}_{\alpha.\beta}$. ■

Lemma 2 *Let $\alpha \in I^*$, $\mathcal{D}_\alpha$ be an I-database, and $freqstring(\mathcal{D}_\alpha) = \beta_1\beta_2\cdots\beta_k$. Then*

$$freqsets(\mathcal{D}_\alpha) = \bigcup_{j\in\{1,\ldots,k\}} freqsets(\mathcal{D}_{\alpha.\beta_j})$$ ■

By following the above example, we can get *freqsets*($\mathcal{D}_{bd}$)=$\{\{d\},\{b\},\{c,d\}\}$, and *freqsets*($\mathcal{D}_{ac}$)= $\{\{c\},\{a\},\{a,c\}\}$.

Based on Lemma 2, we can obtain a more aggressive divide-and-conquer algorithm for mining from disks. The following shows the algorithm *aggressivediskmine*. Here, $freqstring(\mathcal{D}_\alpha)$ is decomposed into several substrings β_j, each of which could have more than one item. Each substring corresponds to a projected database. A transaction τ in $\mathcal{D}_\alpha$ will be partly inserted into $\mathcal{D}_{\alpha.\beta_j}$ if and only if τ contains at least one item a in β_j. Since there will be fewer projected databases, there will be fewer disk I/O's. Compared with the algorithm *basicdiskmine*, we can expect that a large amount of disk I/O will be saved by the algorithm *aggressivediskmine*.

Procedure *aggressivediskmine*($\mathcal{D}_\alpha, M$)
if $|\mathcal{D}_\alpha| \leq M$ **then return** *mainmine($\mathcal{D}_\alpha$)*
else let $freqstring(\mathcal{D}_\alpha) = \beta_1\beta_2\cdots\beta_k$,
return *aggressivediskmine*($\mathcal{D}_{\alpha.\beta_1}, M$) $\cup$
$\ldots \cup$
aggressivediskmine($\mathcal{D}_{\alpha.\beta_k}, M$).

Let's analyze the recurrence and disk I/O's of the aggressive divide-and-conquer algorithm. The number of passes needed is still $1 + \lceil \log_c M/T \rceil \approx 2$, since grouping items does not change the size of an FP-tree for a projected database. However, for disk I/O, suppose in $\mathcal{D}_\epsilon$, each transaction contains on average n frequent items, and that we can group them into k groups of equal size. Then the n items will be written to the projected databases with total length $n/k + 2\cdot n/k + \ldots + k\cdot n/k = (k+1)/2\cdot n$. Total size of all projected databases is $(k+1)/2\cdot D \approx k/2\cdot D$. The total disk I/O's for the aggressive divide-and-conquer algorithm is then

$$5/2\cdot D/B + k\cdot 3/2\cdot D/B \quad (3)$$

The recurrence structure of algorithm *aggressivediskmine* is shown in Figure 2. Compared to Figure 1, we can see that the part of the tree that corresponds to decomposition (the nonshaded part) is much smaller in Figure 2. Although the example is very small, it exhibits the general structure of the two trees.

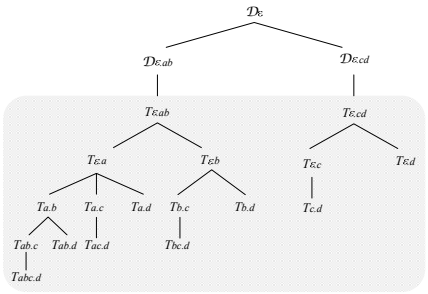


Figure 2. Recurrence of Aggressive Projection

If $k \ll n$, we can expect the aggressive divide-and-conquer algorithm will significantly outperform the basic one.

3 Algorithm Diskmine

The algorithm *Diskmine* is shown below. In the algorithm, $\mathcal{D}_\alpha$ is the original database or a projected database, and M is the maximal size of main memory that can be used by *Diskmine*.

Procedure *Diskmine*($\mathcal{D}_\alpha, M$)
scan $\mathcal{D}_\alpha$ and compute *freqstring*($\mathcal{D}_\alpha$)
call $trialmainmine(\mathcal{D}_\alpha, M)$
if $trialmainmine(\mathcal{D}_\alpha, M)$ aborted **then**
compute a grouping $\beta_1\beta_2\cdots\beta_k$ of $freqstring(\mathcal{D}_\alpha)$
Decompose $\mathcal{D}_\alpha$ into $\mathcal{D}_{\alpha.\beta_1},\ldots,\mathcal{D}_{\alpha.\beta_k}$
for j = 1 **to** k **do begin**
if β_j is a singleton **then** $Diskmine(\mathcal{D}_{\alpha.\beta_j}, M)$
else $mainmine(\mathcal{D}_{\alpha.\beta_j})$
end
else return *freqsets*($\mathcal{D}_\alpha$)

Diskmine uses the FP-tree [7] as data structure and *FP-growth** [6] as main memory mining algorithm. Since the FP-tree encodes all frequency information of the database, we can shift into main memory mining as soon as the FP-tree fits into main memory.

Since an FP-tree usually is a significant compression of the database, our *Diskmine* algorithm begins optimistically, by calling *trialmainmine*, which starts scanning the database and constructing the FP-tree. If the tree can be successfully completed and stored in main memory, we have reached the bottom level of the recursion, and can obtain the frequent itemsets of the database by running *FP-growth** on the FP-tree in main memory.

Procedure *trialmainmine*$(\mathcal{D}_\alpha, M)$
start scanning $\mathcal{D}_\alpha$ and building the FP-tree
  T_α in main memory.
if $|T_\alpha|$ exceeds M **then return** the incomplete T_α
 else call *FPgrowth** (T_α) and **return** *freqsets*$(\mathcal{D}_\alpha)$.

If, at any time during *trialmainmine* we run out of main memory, we abort and return the partially constructed FP-tree, and a pointer to where we stopped scanning the database. We then resume processing *Diskmine*$(\mathcal{D}_\alpha, M)$ by computing a grouping $\beta_1, \ldots, \beta_k$ of *freqstring*$(\mathcal{D}_\alpha)$, and then decomposing $\mathcal{D}_\alpha$ into $\mathcal{D}_{\alpha.\beta_1}, \ldots, \mathcal{D}_{\alpha.\beta_k}$. We recursively process each decomposed database $\mathcal{D}_{\alpha.\beta_j}$. During the first level of the recursion, some groups β_j will consist of a single item only. If β_j is a singleton, we call *Diskmine*, otherwise we call *mainmine* directly, since we put several items in a group only when we estimate that the corresponding FP-tree will fit into main memory.

In computing the grouping $\beta_1, \ldots, \beta_k$ we assume that transactions in a very large database are evenly distributed, i.e., if the size of the FP-tree is n for $p\%$ of the database, then the size of the FP-tree for whole database is $n/p \cdot 100$. Most of the time, this gives an overestimation, since an FP-tree increases fast only at the beginning stage, when items are encountered for the first time and inserted into the tree. In the later stages, the changes to the FP-tree will be mostly counter updates.

Procedure *mainmine*$(\mathcal{D}_{\alpha.\beta})$
build a modified FP-tree $T_{\alpha.\beta}$ for $\mathcal{D}_{\alpha.\beta}$
for each i in β **do begin**
  construct the FP-tree $T_{\alpha.i}$ for $\mathcal{D}_{\alpha.i}$ from $T_{\alpha.\beta}$
  call *FPgrowth** $(T_{\alpha.i})$ and **return** *freqsets*$(\mathcal{D}_{\alpha.i})$.
end

In *basicdiskmine*, since there is only one master item in each projected database (for $\mathcal{D}_\epsilon$, no master item at all), an FP-tree can be constructed without considering the master item. In procedure *mainmine*, since $\mathcal{D}_{\alpha.\beta}$ is for multiple master items, the FP-tree constructed from $\mathcal{D}_{\alpha.\beta}$ has to contain those master items. However, the item order is a problem for the FP-tree, because we only want to mine all frequent itemsets that contain master items. To solve this problem, we simply use the item order in the partial FP-tree returned by the aborted *trialmainmine*$(\mathcal{D}_\alpha)$. This is what we mean by a "modified FP-tree" on the first line in the algorithm *mainmine*.

The entire recurrence structure of *Diskmine* can be seen in Figure 2. Compared to the basic projection in Figure 1 we see that since the aggressive projection uses main memory more effective, and that the decomposition phase is shorter, resulting in fewer disk I/O's.

In Figure 2, the shaded area shows the recursive structure of FP-growth*. Comparing with the shaded area in Figure 1 which shows the recursive structure of the FP-growth method, we can see that the main difference is the extra shaded level in Figure 2. This level is for the FP-trees of groups. For each group, since the total size of all FP-trees for its master items may be greater than the size of main memory, a "modified FP-tree" is constructed. This FP-tree will fit in main memory. From the FP-tree, smaller FP-trees can be constructed one by one, as shown in both figures. As an example, in Figure 1, *basicdiskmine* enters the main memory phase for instance for the conditional database $\mathcal{D}_{\epsilon.a}$. Then FP-growth first constructs the FP-tree $T_{\epsilon.a}$ from $\mathcal{D}_{\epsilon.a}$ (in Figure 2, $T_{\epsilon.a}$ is constructed from $T_{\epsilon.ab}$). The tree rooted at $T_{\epsilon.a}$ shows the recursive structure of FP-growth, assuming for simplicity that the relative frequency remains the same in all conditional pattern bases.

Theorem 1 *Diskmine*$(\mathcal{D})$ returns *freqsets*$(\mathcal{D})$ ■.

Applying the FP-array Technique. In *Diskmine*, the Frequent Pairs Array (FP-array) technique developed for FP-growth* [6] is also applied to save one tree traversal for each recursive call. Furthermore, when projected databases are generated, the FP-array technique can save a great number of disk I/O's.

Recall that in *trialmainmine*, if an FP-tree can not be accommodated in main memory, the construction stops. Suppose now we decided to stop scanning the database. Then later, after generating all projected databases, two database scans are required to construct an FP-tree from a projected database. To save one scan, in *Diskmine* we calculate an FP-array for each FP-tree. When constructing the FP-tree from $\mathcal{D}_\alpha$, if it is found that the tree can not fit in main memory, the construction of the FP-tree T_α stops, but the scan of the database $\mathcal{D}_\alpha$ continues so that we finish filling the cells of the array A_α. Later, only one database scan is needed to construct an FP-tree from a projected database because of the existence of the array A_α.

Grouping items. In *Diskmine*, the fourth line computes a grouping $\beta_1\beta_2\cdots\beta_k$ of $freqstring(\mathcal{D}_\alpha)$. For each β, a new projected database $\mathcal{D}_{\alpha.\beta}$ will be computed from $\mathcal{D}_\alpha$, then written to disk and read from disk later. Therefore, the more groups, the more disk I/O's. In other words, there should be as many items in each β as possible. To group items, two questions have to be answered.

1. If β currently only has one item i_j, after projection, is the main memory big enough for accommodating $T_{\alpha.i_j}$ constructed from $\mathcal{D}_{\alpha.i_j}$ and running the *FP-growth** method on $T_{\alpha.i_j}$?
2. If more items are put in β, after projection, is the main memory big enough for accommodating $T_{\alpha.\beta}$ constructed from $\mathcal{D}_{\alpha.\beta}$ and running *FPgrowth** on $T_{\alpha.\beta}$ only for items in β?

To answer the questions, algorithm *Diskmine* collects statistics on the partial FP-tree T_α and the rest of database $\mathcal{D}_\alpha$.

For the first question, for each item i_j, by counting the number of nodes in the FP-tree $T_{\alpha.i_j}$ constructed from the partial FP-tree T_α, we can use the number to estimate the size of FP-tree $T_{\alpha.i_j}$ constructed from $\mathcal{D}_\alpha$. We write the number as $\mu[j](T_\alpha)$ for each i_j, and suppose the number of transactions in $\mathcal{D}_\alpha$ is $t(\mathcal{D}_\alpha)$ and the number of transactions used for constructing the partial FP-tree T_α is $t(T_\alpha)$. By the assumption that the transactions in $\mathcal{D}_\alpha$ are evenly distributed and that the partial T_α represents the whole FP-tree for $\mathcal{D}_\alpha$, the estimated size of FP-tree $T_{\alpha.i_j}$ is $\mu[j](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha)$.

Before answering the second question, we introduce the *cut point* from which the first group can be easily found.

Finding the cut point. Notice that in *FPgrowth**, when mining frequent itemsets for i_k, all frequency information about $i_{k+1}, \ldots, i_n$ is useless. Thus, though a complete FP-tree T_α constructed from $\mathcal{D}_\alpha$ could not fit in main memory, we can find many k's such that the trimmed FP-tree containing only nodes for items $i_k, \ldots, i_1$ will fit into main memory. All frequent itemsets for $i_k, \ldots, i_1$ can be then mined from one trimmed tree. We call the biggest of such k's the *cut point*. At this point, main memory is big enough for storing the FP-tree containing only $i_k, \ldots, i_1$, and there is also enough main memory for running *FPgrowth** on the tree. Obviously, if the cut point k can be found, items $i_k, \ldots, i_1$ can be grouped together. Only one projected database is needed for $i_k, \ldots, i_1$.

There are two ways to estimate the cut point. One way is to get cut point from the value of $t(\mathcal{D}_\alpha)$ and $t(T_\alpha)$. Figure 3 illustrates the intuition behind the cut point. In the figure, $l = t(T_\alpha)$, and $m = t(\mathcal{D}_\alpha)$. Since the partial FP-tree for $t(T_\alpha)$ of $t(\mathcal{D}_\alpha)$ transactions can be accommodate in main memory, we can expect that the FP-tree containing $i_k, \ldots, i_1$, where $k = \lfloor n \cdot t(T_\alpha)/t(\mathcal{D}_\alpha) \rfloor$, also will fit in main memory.

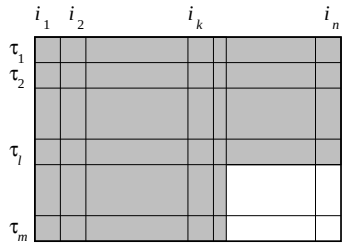


Figure 3. Cut Point

The above method works well for many databases, especially for those databases whose corresponding FP-trees have plenty of sharing of prefixes for items from i_1 to the cut point. However, if the FP-tree constructed from a database does not share prefixes that much, the estimation could fail, since now the FP-tree for items from i_1 to the cut point could be too big. Thus, we have to consider another method. Let $\nu[j](T_\alpha)$ be the size of the FP-tree after the partial FP-tree T_α is trimmed and only contains items $i_1, \ldots, i_j$. Based on $\nu[j](T_\alpha)$ the number of nodes in the complete FP-tree for item i_j can be estimated as $\nu[j](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha)$. Now, suppose $\nu(T_\alpha)$ is the number of nodes in T_α, finding the cut point becomes finding the biggest k such that $\nu[k](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) \leq \nu(T_\alpha)$, and $\nu[k+1](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) > \nu(T_\alpha)$.

Sometimes the above estimation only guarantees that the main memory is big enough for the FP-tree which contains all items between i_1 and the cut point, while it does not guarantee that the descendant trees from that FP-tree can fit in main memory. This is because the estimation does not consider the size of descendant trees correctly. Actually, from $\mu[j](T_\alpha)$ we can get a more accurate estimation of the size of the biggest descendant tree. To find the cut point, we need to find the biggest k, such that $(\nu[k](T_\alpha) + \mu[j](T_\alpha)) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) \leq \nu(T_\alpha)$, and $(\nu[k+1](T_\alpha) + \mu[m](T_\alpha)) > \nu(T_\alpha)$, where $j \leq k$, $\mu[j](T_\alpha) = max_{j \in \{1,\ldots,k\}} \mu[j](T_\alpha)$, and $m \leq k+1$, $\mu[m](T_\alpha) = max_{m \in \{1,\ldots,k+1\}} \mu[m](T_\alpha)$.

Grouping the rest of the items. Now we answer the second question, how to put more items into a group? Here we still need $\mu[j](T_\alpha)$. Starting with $\mu[cutpoint+1](T_\alpha)$, we test if $\mu[cutpoint+1](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) > \nu(T_\alpha)$. If not, we put next item cutpoint+2 into the group, and test if $(\mu[cutpoint+1](T_\alpha) + \mu[cutpoint+2](T_\alpha)) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) > \nu(T_\alpha)$. We repeatedly put next item in $freqstring(\mathcal{D})$ into the group until we reach an item i_j, such that

$$\sum_{m=cutpoint+1}^{j} \mu[m](T_\alpha) \cdot t(\mathcal{D}_\alpha)/t(T_\alpha) > \nu(T_\alpha).$$

Then starting from i_j, we put items into next group, until all items find its group.

Why can we put items $i_j, \ldots, i_k$ together into group β? This is because even if we construct $T_{\alpha.i_j}, \ldots, T_{\alpha.i_k}$ from the projected databases $\mathcal{D}_{\alpha.i_j}, \ldots, \mathcal{D}_{\alpha.i_k}$ and put all of them into main memory, the main memory is big enough according to the grouping condition. At this stage, $T_{\alpha.i_j}, \ldots, T_{\alpha.i_k}$ all can be constructed by scanning $\mathcal{D}_\alpha$ once. Then we mine frequent itemsets from the FP-trees. However, we can do better. Obviously $T_{\alpha.i_j}, \ldots, T_{\alpha.i_k}$ overlap a lot, and the total size of the trees is definitely greater than the size of $T_{\alpha.\beta}$. It also means that we can put more items into each β, only if the size of $T_{\alpha.\beta}$ is estimated to fit in main memory. To estimate the size of $T_{\alpha.\beta}$, part of T_α has to be traversed by following the links for the master items in T_α.

The disk I/O's. Let's re-count the disk I/O's used in *Diskmine*. The first scan is still for obtaining all frequent items in $\mathcal{D}_\epsilon$, and it needs D/B disk I/O's. In the second scan we construct a partial FP-tree T_ϵ, then continue scanning the rest database for statistics. The second scan is a full scan, which needs another D/B disk I/O's.

Suppose then that k projected databases have to be computed. According to Section 2, the total size of the projected databases is approximately $k/2 \cdot D$. For computing the projected databases, the frequency information in T_ϵ is reused, so only part of $\mathcal{D}_\epsilon$ is read. We assume on average half of $\mathcal{D}_\epsilon$ is read at this stage, which means $1/2 \cdot D/B$ disk I/O's. By using of the FP-array technique [6], writing and later reading k projected databases now only take $2 \cdot k/2 \cdot D/B = k \cdot D/B$ disk I/O's. Suppose all frequent itemsets can be mined from the projected databases without going to the third level. Then the total disk I/O's is

$$5/2 \cdot D/B + k \cdot D/B \quad (4)$$

Compared with formula 3, *Diskmine* saves at least $k/2 \cdot D/B$ disk I/O's, thanks to the various techniques used in the algorithm.

4 Performance Study

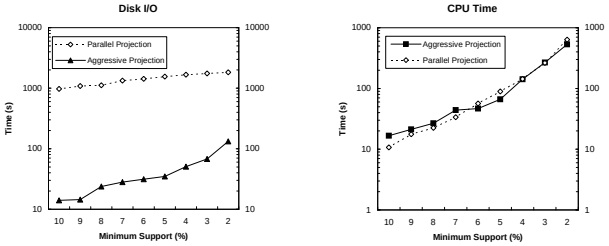
In this section, we present the results from a performance comparison of *Diskmine* with the *Parallel Projection* algorithm in [7] and the *Partitioning* algorithm in [10]. The scalability of *Diskmine* is also analyzed, and the accurateness of our memory size estimations are validated.

As mentioned in Section 2, the *Parallel Projection* algorithm is a basic divide-and-conquer algorithm, since for each item a projected database is created. For performance comparison, we implemented *Parallel Projection* algorithm, by using *FP-growth* as main memory method, as introduced in [7]. The Partitioning algorithm is also a divide-and-conquer algorithm. We implemented the partitioning algorithm by using the Apriori implementation [1]. We chose this implementation, since it was well written and easy to adapt for our purposes.

We ran the three algorithms on both synthetic datasets and real datasets. Some synthetic datasets have millions of transactions, and the size of the datasets ranges from several megabytes to several hundreds gigabytes. Due to lack of space, only the results for some synthetic datasets and a real dataset are shown here.

All experiments were performed on a 2.0Ghz Pentium 4 with 256 MB of memory under Windows XP. For *Diskmine* and the *Parallel Projection* algorithm, the size of the main memory is given as an input. For the Partitioning algorithm, since it only has two database scans and each main-memory-sized partition and all data structures for Apriori are stored into main memory, the size of main memory is not controlled, and only the running time is recorded.

We first compared the performance of three algorithms on synthetic dataset. Dataset *T100I20D100K* was generated from the benchmark application of IBM research center [2]. The dataset has 100,000 transactions and 1000 items, and occupies about 40 megabytes of memory. The average transaction length is 100, and the average pattern length is 20. The dataset is very sparse and FP-tree constructed from the dataset is bushy. For Apriori, a large number of candidate frequent itemsets will be generated from the dataset.

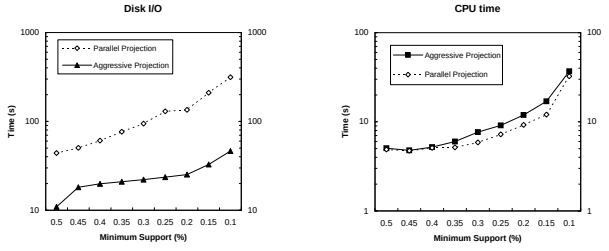


(a) Time for Disk I/O's (b) CPU time

Figure 4. Experiments on synthetic dataset

When running the algorithms, the main memory size was given as 128 megabytes. Figure 4 shows the experimental results. In the figure, "Basic" represents the *Parallel Projection* algorithm, and "Aggressive" represents the *Diskmine* algorithm. Since the Partitioning algorithm is the slowest in the group, its total running time is always an order of magnitude greater than the *Basic* algorithm and the *Aggressive* algorithm, we didn't separate its CPU time and the time for disk I/O's. Consequently the lines for Partitioning algorithm are not shown in the figures. From Figure 4 (a), as expected, we can see that the disk I/O time of the *Aggressive* algorithm is orders of magnitude smaller than that of the *Basic* algorithm. On the other hand, in Figure 4 (b) we can see that the *Basic* algorithm, however, is not slower than the *Aggressive* algorithm if we only compare their CPU time. In [6], where we were concerned with main memory mining, we found that if a dataset is sparse the boosted *FPgrowth** method has a much better performance than the original *FP-growth*. The reason here the CPU time of the *Aggressive* algorithm is not always less than that of *Basic* algorithm is that the *Aggressive* algorithm has to spend CPU time on calculating statistics. However, from Figure 4, we also can see that the CPU overhead used by the *Aggressive* algorithm now become insignificant compared to the savings in disk I/O.

We then ran the algorithms on a real dataset *Kosarak*, which is used as a test dataset in [5]. The dataset is about 40 megabytes. Since it is a dense dataset and its FP-tree is fairly small, we set the main memory size as 16 megabytes for the experiments. Results are shown in Figure 5.



(a) Time for Disk I/O's (b) CPU time

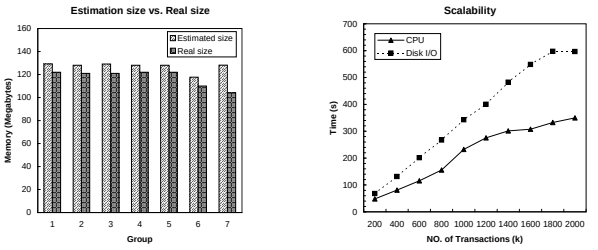
Figure 5. Experiments on real dataset

[1] www.cs.helsinki.fi/u/goethals/software

[2] www.almaden.ibm.com/software/quest/Resources

In Figure 5, for the same reason as above, results for the Partitioning algorithm is not shown. It is still the slowest comparing the total running time. This is because it generates too many candidate frequent itemsets from the dense dataset. Together with the data structures, the candidate sets use up main memory and virtual memory was used. In Figure 5 (a), the time used for disk I/O's of the *Aggressive* algorithm is still remarkably less than the time used for disk I/O's of the Basic Algorithm. We can again notice that the CPU time of the Basic Algorithm is less than that of the *Aggressive* algorithm. This is because *Kosarak* is a dense dataset so the FP-array technique does not help a lot. In addition, calculating the statistics takes an amount of time.

To test the effectiveness of the techniques for grouping items, we run *Diskmine* on *T100I20D100K* and see how close the estimation of the FP-tree size for each group is to its real size. We still set the main memory size as 128 megabytes, the minimum support is 2%. When generating the projected databases, items were grouped into 7 groups (the total number of frequent items is 826). As we can see from Figure 6 (a), in all groups, the estimated size is always slightly larger than the real size. Compared with the Basic Algorithm, which constructs an FP-tree for each item from its projected database, the *Aggressive* algorithm almost fully uses the main memory for each group to construct an FP-tree.



(a) (b)

Figure 6. Estimation Accuracy and Scalability of Diskmine

As a divide-and-conquer algorithm, one of the most important properties of *Diskmine* is its good scalability. We ran *Diskmine* on a set of synthetic datasets. In all datasets, the item number was set as 10000 items, the average transaction length as 100, and the average pattern length as 20. The number of the transactions in the datasets varied from 200,000 to 2,000,000. Datasets size ranges from 100 megabytes to 1 gigabyte. Minimum support was set as 1.5%, and the available main memory was 128 megabytes. Figure 6 (b) shows the results. In the figure, the CPU and the disk I/O time is always kept in a small range of acceptable values. Even for the datasets with 2 million transactions, the total running time is less than 1000 seconds. Extrapolating from these figures using formula (4), we can conclude that a dataset the size of the Library of Congress collection (25 Terabytes) could be mined in around 18 hours with current technology.

5 Conclusions

We have investigated several divide-and-conquer algorithms for mining frequent itemset from secondary memory. We also analyzed the recurrences and disk I/O's of all algorithms. We then gave a detailed divide-and-conquer algorithm which almost fully uses the limited main memory and saves a numerous number of disk I/O's. We introduced many novel techniques used in our algorithm.

Our experimental results show that our algorithm successfully reduces the number of disk access, sometimes by orders of magnitude, and that our algorithm scales up to terabytes of data. The experiments also validate that the estimation techniques used in our algorithm are accurate.

Future extensions of this work will include mining maximal and closed frequent itemsets, as well as exploring disk layout for various datastructures, for instance for candidate sets, since there are some situations where Apriori indeed outperforms the FP-tree based methods.

References

[1] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *KDD'00*, pages 108–118, 2000.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *ACM SIGMOD'93*, pages 207–216, Washington, D.C., 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, pages 487–499, 1994.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE'95*, pages 3–14, 1995.

[5] B. Goethals and M. J. Zaki. Advances in frequent itemset mining implementations: Introduction to fimi03. In *Prodeeding of the 1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Nov 2003.

[6] G. Grahne and J. Zhu. Efficiently using prefix-trees in mining frequent itemsets. In *1st IEEE ICDM Workshop on Frequent Itemset Mining Implementations (FIMI'03)*, Nov 2003.

[7] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8:53–87, 2004.

[8] M. Kamber, J. Han, and J. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *Knowledge Discovery and Data Mining*, pages 207–210, 1997.

[9] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.

[10] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB'95*, pages 432–444, 1995.

[11] H. Toivonen. Sampling large databases for association rules. In *VLDB'96*, pages 134–145, Sep. 1996.

[12] M. Zaki and K. Gouda. Fast vertical mining using diffsets. In *ACM SIGKDD'03*, Washington, DC, Aug. 2003.