

A hybrid method for frequent closed patterns mining in large-scale high dimensional biological data

Guangzhu Yu¹, Keqing Li², Shihuang Shao¹ and Bin Luo³

¹ Information and Technology College, DongHua University, ShangHai, CHINA

² Computer College, WuHan University, WuHan, CHINA

³ Automation College, Guangdong University of Technology, Guangzhou, CHINA

ygz@mail.dhu.edu.cn

guang216@126.com

Abstract

Large high dimensional biological data have posed great challenges to most existing algorithms for frequent patterns mining. In this paper, we propose a hybrid method to find all frequent patterns: we firstly decompose the mining task into two subproblems (discovering long frequent patterns and short frequent patterns), then we choose different algorithms to solve the subproblems respectively. To find long frequent patterns effectively, we describe a partition-based algorithm called inter-transaction. The new algorithm is based on the characteristic that long transactions usually have few common items. In addition, a new pruning strategy is adopted to cut down search space and an optimization technique is used to improve the performance of the intersection of transactions. Experiments on synthetic data show that our method achieves high performance in large high dimensional data.

1. Introduction

The dimension of biological data is often in thousands or tens of thousands, which has posed great challenges to most existing data mining algorithms. Take frequent pattern mining as an example, most of existing algorithms [1][2][3][4] are column enumeration based, which take column (item) combination space as search space. Due to the exponential number of column combination, these kinds of algorithms are not suitable for high dimensional data.

Carpenter [5] and TD-Close [6] have been proposed to mine frequent closed patterns in high dimensional gene data. They are based on the assumption that the datasets have a large number of attributes and relatively small number of rows. However, with the rapid development of studies, biological data increase dramatically. Supposing the number of samples to be small may be false in the future. Furthermore, put different biological data into a single database may lead to a larger, higher dimensional data. Mining in such a high dimensional and large-scale data may be very challenging but interesting. For example, perhaps we can discover relationships between or among

diseases by mining frequent patterns from a high dimensional and large-scale biological data which contains large volume of tissue samples of different kinds. From this point of view, an efficient algorithm for frequent patterns mining in large high dimensional data is needed.

In this paper, we propose a hybrid method to discover all frequent patterns in large high dimensional data. Firstly we decompose the mining task into two parts (discovering long frequent patterns and short frequent patterns), then we use different algorithms to discover all long frequent patterns and short frequent patterns separately. Considering the key problem is to mine long frequent closed patterns efficiently, we propose a partition-based algorithm, i.e., inter-transaction that is based on a new pruning strategy and especially suitable for large high dimensional data. Inter-transaction makes full use of the characteristic that long transactions usually have few common items, which means the intersection of multiple long transactions is usually very short. Since many algorithms can be used to discover short frequent patterns, this paper emphasizes on introducing inter-transaction.

2. Preliminaries

Let $F = \{f_1, f_2, \dots, f_m\}$ be a set of items, also called attributes. $T = \{T_1, T_2, \dots, T_n\}$ is a biological database with each row T_i corresponds to a tissue sample. For convenience of expression, we call each row in database a transaction. Each transaction T_q in database T ($T_q \in T$) is a subset of F , i.e., $T_q \subseteq F$. To simplify notation, we sometimes write a set $\{f_1, f_2, \dots, f_k\}$ as $f_1 f_2 \dots f_k$.

We refer to a set of transaction identifiers as tidlist, and the itemset obtained from the intersection of the transactions listed in the tidlist as intersection transaction of tidlist, denoted $T(\text{tidlist})$. For example, let $T_1 = ABDF$, $T_2 = ADFG$, $T_3 = ADFQ$, then one of the tidlists is $\{1, 2, 3\}$, corresponding intersection transaction $T(1, 2, 3) = T_1 \cap T_2 \cap T_3 = ADF$. If $|\text{tidlist}| = k$ ($1 \leq k \leq N$, N is the number of transactions), we refer to $T(\text{tidlist})$ as k -intersection transaction. A k -intersection transaction is the intersection of k individual

transactions, and it can also be regarded as the intersection of other two intersection transactions. For example, 4-intersection transaction $T(1,3,5,6)$ can be regarded as the intersection of $T(1,3)$ with $T(5,6)$ or other pairs of intersection transactions, i.e., $T(1,3,5,6)=T_1 \cap T_3 \cap T_5 \cap T_6 = T(1,3) \cap T(5,6) = T(1,5) \cap T(3,6) = T(1,3,5) \cap T(6)$, and so on. For convenience, in the remainder of the paper, we sometimes refer to a k -intersection transaction as a transaction ($1 \leq k \leq N$) if there is no confusion.

Although k -intersection transaction is actually an itemset, there are some differences between them. For example, a k -itemset means an itemset with k items, the term doesn't tell us any information about its support (i.e., the number of transactions containing the itemset); on the contrary, k -intersection transaction doesn't tell us how many items the itemset has, but it tells us the itemset stems from the intersection of k transactions, with support not less than k . To emphasize the difference between k and support s , we refer to k as the current support of k -intersection transaction. Obviously, $k \leq s$. If $k=s$, corresponding tidlist is called maximal tidlist

We also use $T(\text{tidlist})$, tidlist represents the set of transaction identifiers associated with $T(\text{tidlist})$. If $s(T(\text{tidlist}))$ represents the current support of intersection transaction $T(\text{tidlist})$, we have:

$$T(i, j).tidlist = T(i).tidlist \cup T(j).tidlist \quad (1)$$

$$s(T(i, j)) = |T(i).tidlist \cup T(j).tidlist| \quad (2)$$

$$T(\text{tidlist}).tidlist = \text{tidlist} \quad (3)$$

Note that i or j might be an integer or a tidlist in above equations. If it is an integer, $T(i)$ can be written as T_i , and $T_i.tidlist = \{i\}$, $s(T_i) = 1$.

Given an itemset/transaction, we call it a long itemset/transaction if it has more than minlen items. minlen is a user specified threshold. Otherwise, we call it a short itemset/transaction. Other terms such as partition, locally large (frequent) itemset can refer to [7]. If minimum support threshold is minsup , local support is defined as minsup/n , n the number of partitions.

3. A new pruning strategy

Downward closure property, which states if an itemset is frequent by support, then all its nonempty subsets must also be frequent by support, is a common feature for all datasets and widely used as pruning strategy, but it doesn't guarantee the best performance in any term when it was used to cut down search space. What special characteristics of a high dimensional data can be used to cut down search space?

Because of downward closure property, most of the frequent patterns are short, given a proper minlen . In other word, long frequent patterns are relatively "sparse" compared with short frequent patterns. Although the number of long frequent patterns is not very large, they

will take most of time in most cases if a traditional algorithm for frequent patterns mining is adopted. In addition, long transactions/itemsets usually have less common items, especially in sparse high dimensional data, which means the intersection of multiple long transactions, i.e., intersection transaction is usually very short. The characteristics are very useful for us because we can obtain long frequent itemsets directly by intersecting relevant transactions, without extending a short itemset step by step to identify a long frequent pattern.

On the contrary, short frequent itemsets are relatively dense; the speed of enumerating all intersection transaction by intersecting all short transactions will be very slow. Based on the different features, it's reasonable for us to decompose the mining task into two subproblems (discovering long frequent patterns and short frequent patterns), so that we can choose proper algorithm to solve the subproblems separately.

If we aim to find long frequent patterns directly by intersecting relevant transactions, a new pruning strategy can be used to narrow the search space: filter out all short (intersection) transactions. The rationale behind the pruning strategy is that short transactions have no effect on the support of long patterns/itemsets, and the intersection of a short transaction with another transaction (short or long) must be short. Now that the intersection transaction of two long transactions is usually very short, large amount of intersection transactions can be pruned out in time; this is why the pruning strategy is so efficient on high dimensional data

4. The inter-transaction algorithm

The support of an itemset is determined by a group of transactions listed in tidlist. If we let any two transactions intersect each other ($|\text{tidlist}|=2$), we can obtain all itemsets with support higher than two. Likewise, we can obtain all itemsets with support higher than s by intersecting any s transactions ($|\text{tidlist}|=s$, $1 \leq s \leq N$, N is the number of transactions). Theoretically, we can obtain all itemsets by intersecting transactions. In fact, like Carpenter [5] and TD-Closet [6], our method is also based on row enumeration. According to lemma 3.2 in [5], each intersection transaction is a closet pattern.

Let N be the number of transactions, there will be 2^N intersection transactions in the worst situation. In real database, N can easily reach to several millions, enumerating all intersection transactions is impractical. To solve the problem, two methods can be used. One is to use the new pruning strategy mentioned in section 3, the other is to divide database into multiple partitions with each partition containing fitting amount of

transactions, then establish a global candidate set from locally frequent itemsets, and finally test the entire candidate set, just like that described in [7].

The formal algorithm is shown in Figure 1.

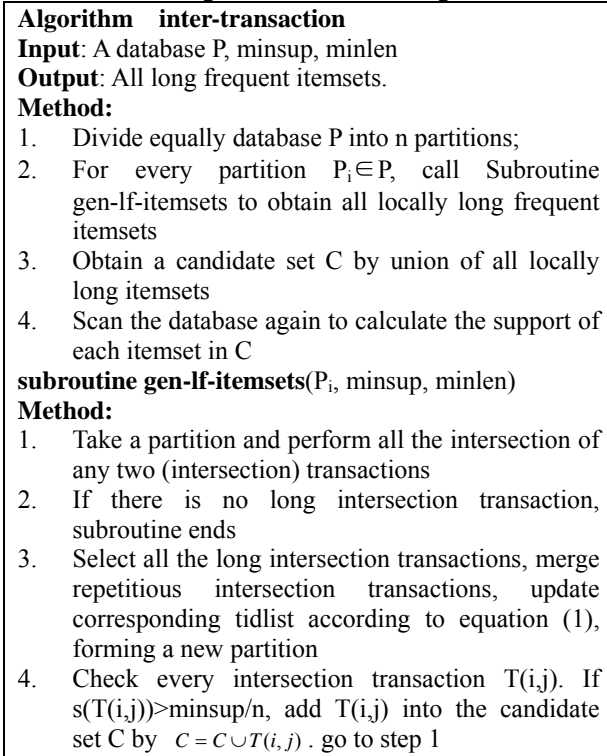


Figure 1 Algorithm inter-transaction

Inter-transaction algorithm is very similar to the partition algorithm [7], but there are two important differences between them. One is that in the partition algorithm the size of partition is chosen in term of main memory size, whereas inter-transaction seeks balance between producing less candidates and reducing the amount of intersection operation. The other difference is that in partition algorithm a certain algorithm is used to generate locally frequent itemsets of all length, whereas inter-transaction discovers only locally long frequent itemsets via enumerating intersection transactions.

Gen-lf-itemsets is responsible for generating locally long frequent itemsets. In the subroutine, the new pruning strategy is used to reduce the amount of intersection transactions, tidlist is used to record which transactions are involved in a intersection transaction.

How should we choose the size of partitions? From perspective of producing fewer candidates, the number of partitions should be small, and thus the size of partitions should be large, given the size of dataset. Otherwise, data skew may lead to too many candidates. But a too large partition contains large amount of intersection transactions and will thus lead to large amount of

intersection operations. Experiments show that it's applicable for partition size to be between $\frac{5}{a}$ and $\frac{10}{a}$ in the context of our datasets, where coefficient a is the minimum acceptable percentage of transactions containing an itemset to the total number of transactions in the database.

Note that in subroutine gen-lf-itemsets, we use the output of previous intersection as the input of next intersection operation. This method essentially is the same as the row enumeration tree described in [5]. In fact, each intersection transaction corresponds to a node in the row enumeration tree. It not only guarantees complete set of long frequent itemsets can be obtained, but also has following advantages: 1) improve the speed of enumerating all intersection transaction; 2) don't maintain complicated data structure like X-conditional table.

Another important thing to note is that redundant information is used to improve the speed of intersection operation, which will be described in section 6

5. Inter-transaction vs. Carpenter

As we have mentioned in section1, Carpenter is only suitable for small datasets with small rows, whereas inter-transaction can handle large high dimensional data. As an alternative, we can use Carpenter to obtain all locally frequent closet patterns of all length, but it's not the best choice if we aim at finding long patterns: extending X-conditional transposed table step by step is costly; it's not worth maintaining a complicated data structure in sparse high dimensional database. After each round of intersection of transactions, larger amount of short intersection transactions are pruned out, computing all the intersections of any two long intersection transactions is not a big burden. In this situation, the benefit of using X-conditional transposed table is not obvious, not to say the extra overhead entailed by this method, including increase of program complexity and storage requirements. Furthermore, this method has also to perform large amount of repetitious intersections of transactions. For example, let $\text{tidlist1}=\{1,2,3,4,5,6\}$, $\text{tidlist2}=\{2,3,4,5,6\}$, to obtain $T(\text{tidlist1})$, Carpenter will call MinePattern function five times to intersect transactions 1,2,3,4,5 and 6 step by step, Then it has to intersect transaction 2,3,4,5 and 6 again to obtain $T(\text{tidlist2})$, without using the results obtained in previous step.

In order to make full use of the intermediate results, we can obtain a new k-intersection transaction by computing the intersection of a k_1 -intersection transaction with a k_2 -intersection transaction ($k_1 \leq k$, $k_2 \leq k$, example can be seen in section 2). For convenience, we call our method "the enumeration of the intersection of two intersection transactions", or

EITIT for short. EITIT don't use an individual transaction to extend tidlist step by step, but use two intersection transactions, the intermediate results, to obtain a new intersection transaction. Compared with the method of extending X-conditional transposed table step by step, EITIT can jump over some unnecessary step that must be performed in the process of extending X-conditional transposed table. Take the same example as above, in order to obtain $T(2,3,4,5,6)$, EITIT can intersect two intersection transactions obtained in previous step, such as $T(2,3,4,5)$ and $T(3,4,5,6)$ or other pairs of intersection transactions, only one step is needed. Figure 2 is an example that can show how EITIT works.

From the example show in figure 2, we can observe that EITIT take less steps to obtain $T(1,2,3,4,5)$. The benefit brought on by the jumping method increases as the tidlist increases. But in the meantime, another problem rises: some intersection transactions will be calculated repeatedly (gray-shaded boxes in figure 2). For example, we can obtain $T(1,2,4,5)$ by intersecting $T(1,2)$ with $T(4,5)$, while we can also obtain the same itemset by intersecting $T(1,5)$ with $T(2,4)$. If no measures were taken, the repetitious computation may grow explosively in later round. In order to limit the number of repetitious intersection, we merge repetitious intersection transactions into a single one after each round of the intersection of any two long intersection transactions (step 5 in subroutine gen-LHU-itemsets).

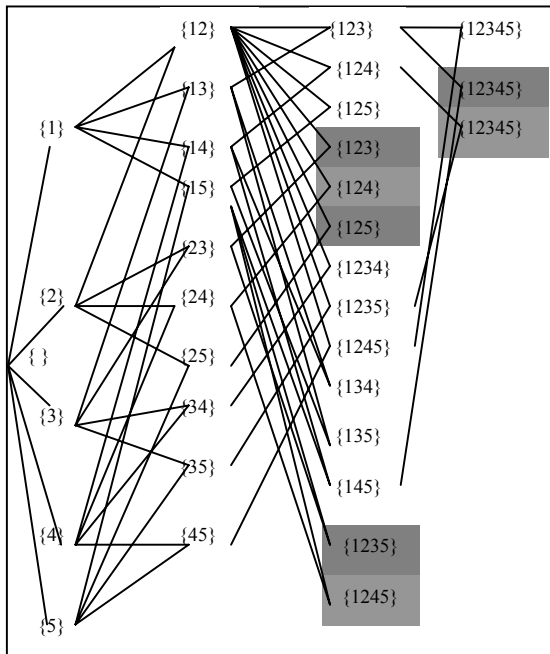


Figure 2 Intersection transactions enumeration

Here we give an example to show how to merge two

repetitious intersection transactions. Suppose there are two sets of transactions, corresponding transaction identifier lists are $tidlist1$ and $tidlist2$. If $T(tidlist1) = T(tidlist2)$ and they are long (may not be filtered out in subsequent steps), we can merge them into a new intersection transaction $T(tidlist) = T(tidlist1) = T(tidlist2)$, with $tidlist = tidlist1 \cup tidlist2$

That's to say, new intersection transaction $T(tidlist)$ has the same items as $T(tidlist1)$ or $T(tidlist2)$, but its $tidlist$ should be updated. for example, let $tidlist1 = \{1,3,4,5,7\}$, $tidlist2 = \{1,3,5,8\}$, if $T(1,3,4,5,7) = T(1,3,5,8)$, new intersection transaction should be $T(1,3,4,5,7,8) = T(1,3,4,5,7) = T(1,3,5,8)$.

Generally speaking, in sparse high dimensional database environment, EITIT has following advantages:

- 1) Improve the speed of finding long closet pattern;
- 2) Don't maintain complicated data structure like X-conditional table.

6. Data layout alternatives

Conceptually, a transaction database is a two-dimensional matrix where the rows represent individual customer purchase transactions and the columns represent the items on sale. The matrix can be implemented in the following four different ways [8][9]:

- 1) Horizontal item-vector (**HIV**): the database is organized as a set of rows with each row storing a transaction identifier (TID) and a bit-vector of 0's and 1's to represent for each of the items on sale, its presence or absence, respectively, in the transaction.
- 2) Horizontal item-list (**HIL**): this is similar to HIV, except that each row stores an ordered list of item-identifiers (IID), representing only the items actually occur in the transaction.
- 3) Vertical TID-vector (**VTV**): each column storing an IID (itemID) and a bit-vector of 1's and 0's to represent the presence or absence, respectively, of the item in the set of transactions.
- 4) Vertical TID-list (**VTL**): this is similar to VTV, except that each column stores an ordered list of only the TIDs of the transactions in which the item occur.

Each aforementioned format can be used to express data independently. If we express each transaction in horizontal item-vector format (HIV), intersection transaction can be obtained from the intersection of bit-vector. Although the bitwise and operation is efficient, the overall performance of the intersection operation of two transactions decreases dramatically with the increase of the number of items. For example, if the number of items is 8k, we have to use 1k bytes (8k bit) to express each transaction. In order to perform the intersection of two transactions, 8k bit operations are needed and this is intolerable. If two transactions are represented in HIL format, the benefit

of bitwise logical operation couldn't be shared. When the length of data in HIL format becomes long, the performance decreases dramatically.

Some optimization techniques such as run-length encoding (RLE), DIFFSET [9] and VIPER [8] have been proposed to enhance the performance of the intersection of two bit-vectors in vertical mining algorithms. VIPER uses a compressed bit-vector called "snakes" to improve the performance of intersection operation. Although VIPER can reduce the need for memories and improve the speed of intersection operation of two TID-list, compressing or uncompressing a bit-vector is still costly. DIFFSET only keeps track of differences in the tids of a candidate pattern from its generating frequent patterns. When the differences are large enough, databases in DIFFSET format may occupy more space. So DIFFSET is only suitable for dense databases. Generally speaking, existing optimization techniques have limited improvement on the speed of long transaction intersection, the reason may be most of them aim at reducing the memory requirement of algorithms, and thus adopt various compressed formats to store databases. Because transaction intersection can't be performed directly in these compressed formats, extra format converting can't be avoided. Given a certain amount of memory, our aim is to improve speed of intersection operations as soon as possible, so in our algorithm, we not only don't compress each row in main memory, but also use redundant information to reduce the amount of bitwise logical operations. This point is very important for our algorithm.

Besides the HIV format, we also store each transaction in HIL format. Although this method will waste much of memories, the cost is affordable because partition method can save lots of memories, and its benefit is tremendous. HIV format is used to perform the intersection of bit-vectors, while HIL format is used to store redundant information which guides us to choose only necessary bits in a bit-vector to perform bitwise And operation. Let $S1$ be the length of transaction T_1 in HIL format, $S2$ be the length of transaction T_2 , if $S1 > S2$, we choose T_2 (the shorter transaction) as the benchmark to determine which bits to perform bitwise And operation. The method is: if the k -th bit in T_2 is 1, bitwise And operation should be performed on the bit, all other bits should be set 0 in the result of intersection operation. For example, there are 3 transactions, corresponding data can be seen in table 1:

	HIV format	HIL format
T_1	1011,1100,1100,0110,001	1,3,4,5,6,9,10,14,15,19
T_2	1000,0000,0000,0000,011	1,18,19
T_3	1011,1100,1101,0110,001	1,3,4,5,6,9,10,12,14,15,19

Table1. Transactions in different formats

if we want to get $T(1,2,3)$, we can first get $T(1,2)$ by intersecting transactions T_1 with T_2 , then we get $T(1,2,3)$

by intersecting $T(1,2)$ with T_3 . In order to get $T(1,2)$, we choose the shorter transaction (in HILformat) T_2 as the benchmark and decide bitwise And operations should be performed only on the first bit, the eighteenth bit and nineteenth bit (gray-shaded boxes in table 4), other bits should be set zero. As a intermediate results, we get $T(1,2)=\{1,19\}$ (in HIL format). In subsequent process of intersecting $T(1,2)$ with T_3 , we choose $T(1,2)$ as the benchmark and decide bitwise And operations should be performed only on the first bit and the last bit. We get the final result $T(1,2,3)=\{1,19\}$. As shown in table 2, only five bit operations are needed for the whole process.

	HIV format	HIL format
T_1	1011,1100,1100,0110,001	1,3,4,5,6,9,10,14,15,19
T_2	1000,0000,0000,0000,011	1,18,19
$T(1,2)=T_1 \cap T_2$	1000,0000,0000,0000,001	1,19
T_3	1011,1100,1101,0110,001	1,3,4,5,6,9,10,12,14,15,19
$T(1,2,3)=T(1,2) \cap T_3$	1000,0000,0000,0000,001	1,19

Table 2. The process of computing $T(1,2,3)$

In this way, the amount of bit operations depends linearly only on the length of the short transaction in HIL format. Experiment shows this method outperforms VIPER and DIFFSET, and we also ensure it outperforms all other optimization techniques used in existing vertical algorithm, especially in the context of long bit-vector.

7. Experimental results

All the experiments were performed on a 2GHz XEON server with 2GB of memory, running windows 2003. Program was coded in Delphi 7. There are 3 groups of synthetic datasets used in our experiments. The first group is T40.I30.D8000K with items varying from 0.5K to 16K, where $T\#$ stands for average length of transactions, $I\#$ stands for average length of maximal pattern, and $D\#$ stands for the number of transactions. The second group is T60.I30.D1K, T80.I30.D1K, T100.I30.D1K, T120.I30.D1K and T140.I30.D1K, which were used to compare the performance when varying average length of transactions, between inter-transaction and carpenter. The third is T80.I10.D0.3K, T80.I20.D0.3K, T80.I30.D0.3K, T80.I40.D0.3K, T80.I50.D0.3K and T80.I60.D0.3K, used for comparing the performance when varying average length of maximal patterns. All the data were generated by IBM quest data generator [10].

Figure 3 presents the scalability of inter-transaction by increasing the number of transactions from 0.25M to 8M. Experimental result shows that our algorithm scales linearly with the number of transactions.

Figure 4 shows the performance when varying the number of items. Different from other algorithms, the performance of inter-transaction increases with the increase of the number of items. The reason is that the number of items has direct relationship with the sparseness of a dataset. The more the items, the sparser the dataset, and the faster the speed of the intersection transaction becoming short. That means inter-transaction can enumerating all long intersection transactions easily in a sparse dataset. From figure 4 we can observe that inter-transaction is suitable for those datasets with more than 1k items.

In Figure 5, minlen (a positive integer) is the minimum length of itemsets inter-transaction can discover within a reasonable time (several hours), it actually decides the task assigned to inter-transaction. Figure 5 shows minlen decreases as the number of items increases, which means inter-transaction can complete more mining tasks in a sparse database. The reason is just the same as mentioned above: The more the items, the sparser the dataset, and the shorter the intersection transaction of two transactions.

Figure 6 shows the execution time of inter-transaction when varying the utility threshold. Since the number of candidate itemsets decreases as the minimum utility threshold increases, the execution time decreases, correspondingly.

Figure 7 shows the size of partitions is very important for inter-transaction. Just as we have mentioned in section 4, a too small or a too large partition will degrade the performance of the algorithm.

Figure 8 shows the effect of minlen on the performance of inter-transaction. As we have mentioned, minlen actually assign mining tasks between inter-transaction and its cooperater such as Umining or Two-phase. The larger the minlen, the fewer the tasks assigned to inter-transaction, and the shorter the

execution time needed for inter-transaction. Although a large minlen always means a short execution time for inter-transactions, more tasks will be left for its cooperater in the meantime; the overall performance isn't necessarily high. On the other hand, a too small minlen doesn't benefit the overall performance of the hybrid model. In general, after several rounds of computing the intersection transactions, the speed of shortening intersection transaction decreases rapidly, and the number of short intersection transaction will become more and more. If minlen is too small, inter-transaction has to enumerate too many intersection transactions, the running time increases dramatically. So a proper minlen is a key parameter for the hybrid model. Figure 5 indicates that we should choose different minlen in terms of the number of items. In the meantime, our experiment also shows that data skew affects the choosing of minlen. Usually, if the dataset is too large, the possibility of one partition containing too many long patterns increases, and this may shut down our computer. In this situation, a larger minlen should be chosen.

In order to compare the performance between Carpenter and inter-transaction (in fact, compare the performance between using EITIT and X-conditional transposed table), we implement Carpenter to our best knowledge. Since inter-transaction can only find long frequent patterns, we set minlen=5 in our experiment and let Carpenter enumerate only long itemsets. That's to say, another pruning strategy was incorporated into our Carpenter: if the node in row enumeration tree is short (has less than minlen items), stop extending X-conditional transposed table. Figure 9 and figure 10 show inter-transaction outperform Carpenter and is not sensitive to average length of transactions and maximal patterns, respectively.

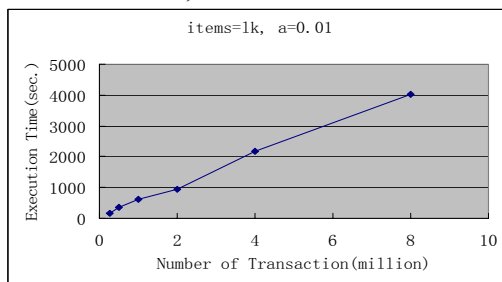


Figure 3 Scalability with the number of transactions

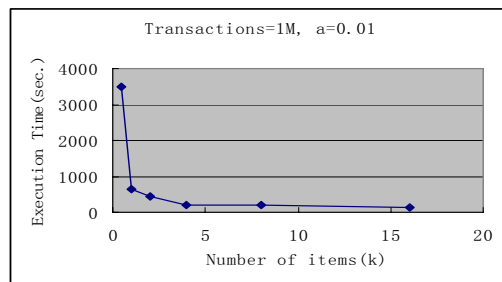


Figure 4 Scalability with the number of items

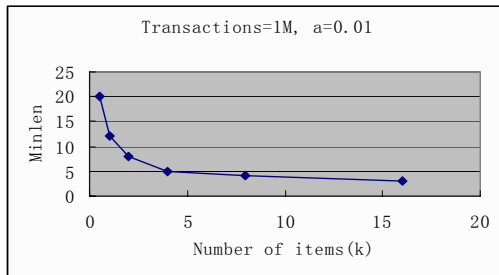


Figure 5 The effect of the number of items on minlen

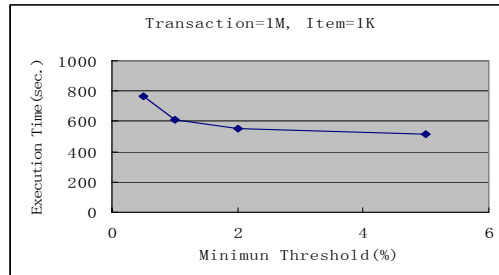


Figure 6 Scalability with threshold

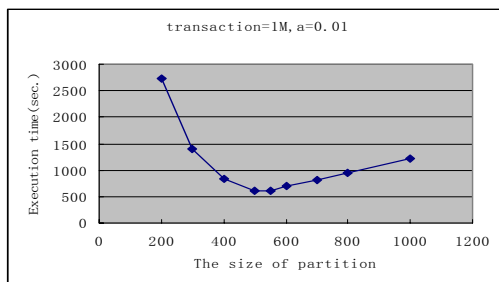


Figure 7 The effect of the size of partitions on performance

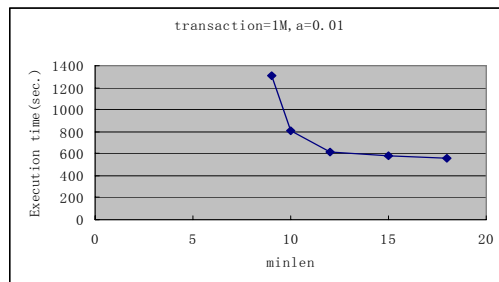


Figure 8 The effect of minlen on performance

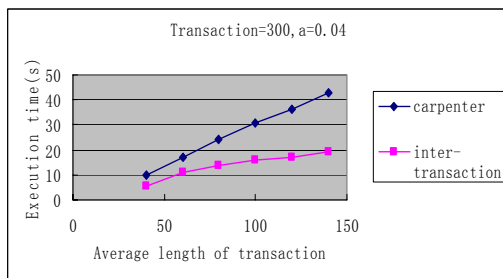


Figure 9 Scalability with the average length of transaction

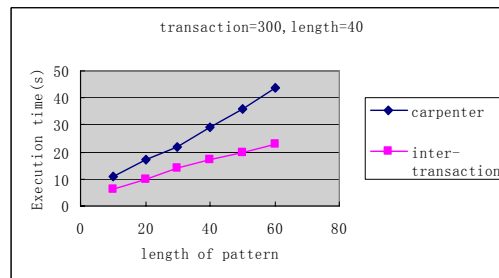


Figure 10 Scalability with the average length of pattern

8. Conclusions

The paper proposes a hybrid method to find frequent closet patterns in two directions separately. The intuition of the method is to decompose a complex problem into two sub-problems that are relatively easier to solve. To discover long frequent patterns in large high dimensional data, a partition-base algorithm called inter-transaction is proposed. Compared with existing algorithms, inter-transaction has following characteristics: 1) by making full use of the sparseness of long pattern, inter-transaction can discover long frequent closet patterns directly by intersecting relevant transactions and thus avoid the costly iteration of candidate itemsets generation and test. It shares the benefit of bitwise logical operations, which are well supported by computer hardware. No complicated data structures need to be maintained; 2) since inter-transaction is partition-based, large amount of memories can be saved if the size of partitions is not very large. This enables us to store a transaction in two different ways. Although this

redundant method wastes lots of memories, it can solve the problem that the performance of the intersection of two bit-vectors decreases as the length of the two bit-vectors increases, which has been suffered by existing vertical algorithms for frequent patterns mining; 3) a new pruning strategies is taken to cut down search space.

The features of datasets, including the number of items and data skew, and Parameters such as minimum threshold minlen, size of partitions affect the performance of inter-transaction. How to reduce the effects is our future work. In the meantime, we shall choose or develop an algorithm for short frequent itemsets mining, making an intensive study of the overall performance of the hybrid mode. In addition, we will collect as much biological data as possible and try our algorithm on these real data.

References

- [1] Agrawal R, Srikant R. Fast algorithms for mining association rules. In VLDB'94, Santiago de Chile,

- Sept. 1994, pp.487-499.
- [2] Mannila H, Toivonen H, Verkamo A I. Efficient algorithms for discovering association rules. In: Proc. AAAI'94 Workshop Knowledge Discovery in Databases (KDD'94), Seattle, WA, July 1994. 181~192
- [3] Zaki M J, Parthasarathy S, and et al. New Algorithm for Fast Discovery of Association Rules[C]. In Proc. of the 3rd Int'l Conf. on KDD and Data Mining (KDD'97), Newport Beach, California, 1997. 283-286.
- [4] Pasquier N, Bastide Y, Taouil R, Lakhal L. Discovering frequent closed itemsets for association rules. In ICDT'99, Jerusalem, Israel, Jan. 1999. 398~416
- [5] Feng, P., Gao. C., et al., CARPENTER: Finding closed patterns in long biological database, Proc. of SIGKDD'03, 2003:413-419
- [6] Hongyan L., Han J.W. and et al. Mining frequent Patterns from Very High Dimensional Data: A Top-down Row Enumeration Approach. Proceedings of the Sixth SIAM International Conference on Data Mining. Bethesda, Maryland, 2006. 20-22.
- [7] Savasere A, Omiecinsky E and Navathe S (1995). An efficient algorithm for mining association rules in large databases. 21st Int'l Conf. on Very Large Databases, 1995, pp. 432-444.
- [8] Shenoy P, Haritsa J R, Sudarshan S, et al. Turbo-charging Vertical Mining of Large Databases. Proceedings of ACM SIGMOD International Conference on Management of Data, 2000. p.22-33.
- [9] Zaki M J, Gouda K. Fast vertical mining using diffsets. In Proc. of ACM SIGKDD'03. Washington, DC: 2003.
- [10] http://www.almaden.ibm.com/cs/projects/iis/hdb/Projects/data_mining/datasets/syndata.html