CHAPTER 2

Review of Frequent Itemset Mining

2.1 Overview

The algorithms for discovering frequent itemset are systematized in Figure 2.1 [9]. The characterization is based on two facts. The first is the strategy to traverse the search space, which can be divided into two types: depth-first search and breadth-first search. The second is the strategy to determine the support values of each itemset, and also can be categorized into two classes: counting occurrence and transaction-ID intersecting. The algorithms for frequent itemset mining can be categorized into four types, and the representatives respectively are Apriori, Partition, FP-Growth, and Eclat.

The brute force way for discovering frequent itemsets is to generate the itemsets with every combination of items, and then determine whether they are frequent or not. However, in this chapter, we introduce three efficient algorithms, Apriori, Eclat and FP-Growth in the following three sections respectively. All of them can reduce the search space efficiently. Besides, these are three of the most significant methods to discover frequent itemset. Especially, the FP-Growth is the most efficient algorithm nowadays. We also give a brief summary in the last section.

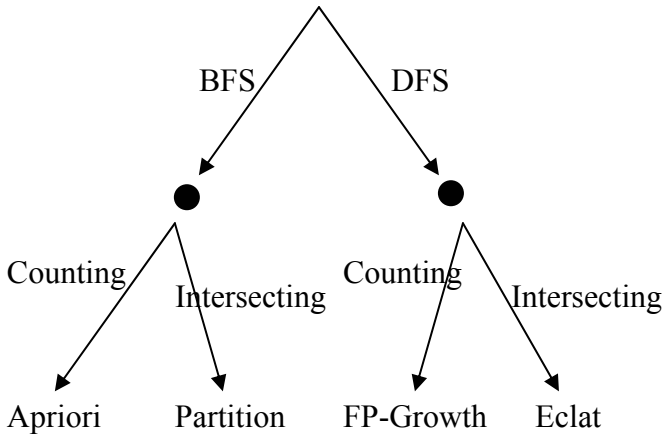


Figure 2.1: The algorithms for frequent itemset mining can be categorized into four types [9].

Before illustrating the mining algorithms, some definitions have to be presented.

Definition 2.1 Let I be a set of literals, called items, and $D=\{T_1, T_2, \ldots, T_k\}$ be a transaction database, where transaction T_i is a set of items such that $T_i \subseteq I$, and $i=1\sim k$. If there is an itemset $X \subseteq T_{i_j}$, $1 \le i_j \le k$, where $j=1, 2, \ldots, m$, the support of X is m, denoted as $support(X)=m$.

Database

TID	Items
1	1, 3, 4
2	2, 3, 5
3	1, 2, 3, 5
4	2, 5

Figure 2.2: Example for mining frequent itemset.

Example 2.1 Let I={1, 2, 3, 4, 5} be a set of items, and D be a transaction database shown in Figure 2.2. The support of the itemset X={2, 5} is three, because X is contained in transaction 2, 3 and 4.

Definition 2.2 Let D be a transaction database, and the minimum support is h. If the support of an itemset X is m, and $m \geq h$, X is a frequent itemset of D.

Example 2.2 Considering the database in Figure 2.2. If the minimum support is two, the frequent itemsets are {1}, {2}, {3}, {5}, {1, 3}, {2, 3}, {2, 5}, {3, 5} and {2, 3, 5}.

Definition 2.3 Let D be a transaction database, X_i are frequent itemsets of D, where i=1, 2,…, k. If $X_i \not\subset X_j$, j=1, 2, …, k, and $i \neq j$, X_i are maximum frequent itemsets of D [10].

Example 2.3 Considering the database in Figure 2.2. If the minimum support is two, the maximum frequent itemsets are {1, 3} and {2, 3, 5}.

Definition 2.4 Let D be a transaction database, X_i is a frequent itemset of D. If $X_i \subseteq X_j$ and $support(X_i) \neq support(X_j)$, where $i \neq j$, X_i is defined as a closed frequent itemset of D.

Example 2.4 Considering the database in Figure 2.2. If the minimum support is two, the closed frequent itemsets are {3}, {1, 3}, {2, 5}, and {2, 3, 5}.

2.2 Algorithm Apriori

Apriori algorithm is a level-wise technique to discover frequent subsets. In the category of [9], Apriori is a BFS and counting occurrence strategy. Each level is a join-and-prune process based on the property: all nonempty subsets of a frequent subset must also be frequent.

Figure 2.3 gives the algorithms for Apriori. At the *k*-th pass, Apriori joins the frequent (*k*-1)-subset to generate candidate by the function apriori-gen and stores the candidate *k*-subsets in a hash-tree. Figure 2.4 shows the function of apriori-gen. Next, Apriori counts the supports of each candidate through the subset function in line 5, Figure 2.3. For each transaction, the subset function can obtain which candidates are contained in it. And finally, prunes the infrequent subsets to determine the frequent *k*-subset. Figure 2.5 shows an example for Apriori to find frequent subset while the minimum support is two.

```
1)  L1 = {large 1-itemsets};
2)  for (k = 2; Lk-1 ≠ φ; k++) do begin
3)      Ck = apriori-gen(Lk-1);    // New candidates
4)      forall transactions t ∈ D do begin
5)          Ct = subset(Ck, t);    // Candidates contained in t
6)          forall candidates c ∈ Ct do
7)              c.count++;
8)      end
9)      Lk = {c ∈ Ck | c.count ≥ minsup}
10) end
11)     Answer = ⋃k Lk
```

Figure 2.3: Apriori algorithm [2].

```
insert into Ck
select p.item1, p.item2, …, p.itemk-1, q.itemk-1
from Lk-1 p, Lk-1 q
where p.item1 = q.item1, …, p.itemk-2 = q.itemk-2, p.itemk-1 < q.itemk-1;
forall itemset c ∈ Ck do
   forall (k-1)-subsets s of c do
      if (s ∉ Lk-1) then
         delete c from Ck;
```

Figure 2.4: Apriori-gen function in algorithm Apriori [2].

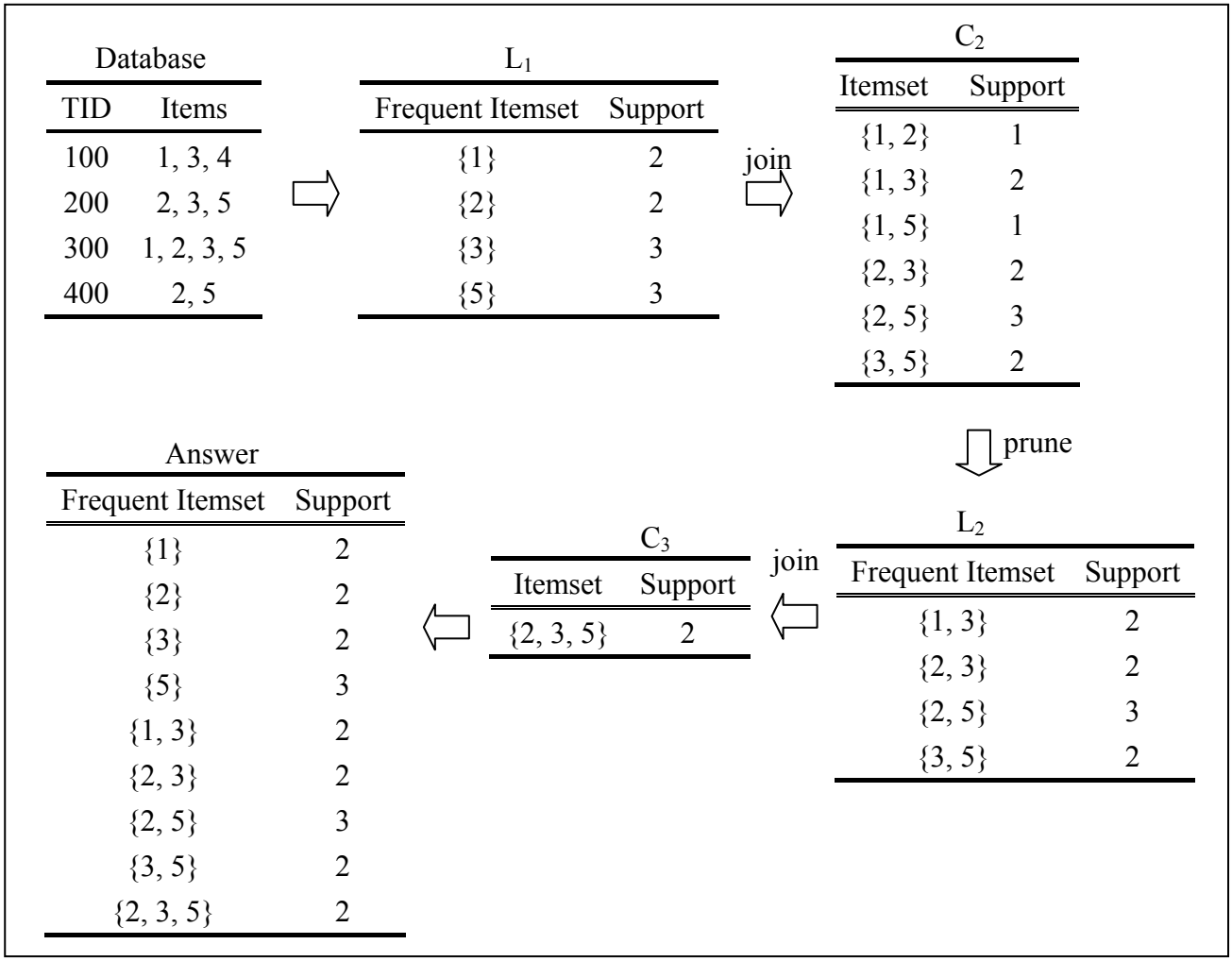


Figure 2.5: An example for Apriori [2].

Figure 2.6 depicts the hash tree that stores the candidate 2-subsets. The leaf node contains a list of itemsets, while the interior node contains a hash table. The matching method starts from root node, and finds all the candidates contained in a transaction *t* as follows. For the root, it hashes on every item in *t*, and for the interior node, it hashes on each item that comes after the item reached currently. If it is at the leaf node, it adds the corresponding itemsets into answer set. For example, when scanning database in transaction 200 ({2, 3, 5}), it starts at the root node, and hashes the items, 2, 3, and 5. Next, at the hash

table linked by the item 2 of root node, it hashes the items 3 and 5, which come after item 2. When it is at the leaf node {2, 3} and {2, 5}, it adds these two itemset into answer set. Utilizing the same rules, it can easily add the three 2-itemset {2, 3}, {3, 5}, and {2, 5} in to answer set.

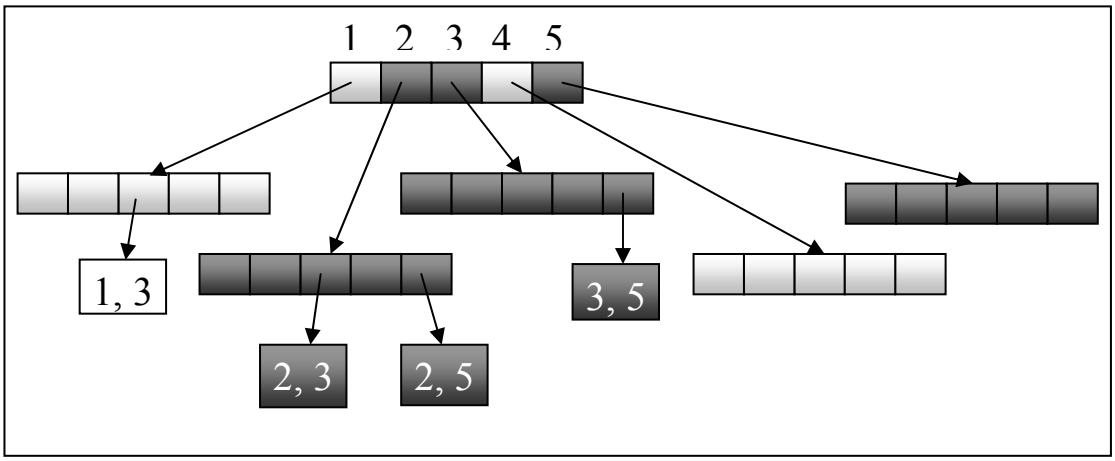


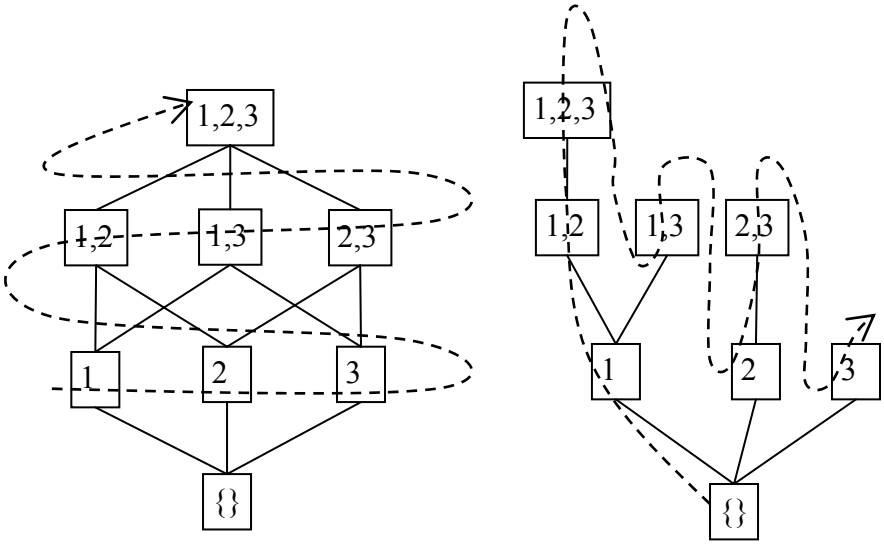
Figure 2.6: The candidate 2-itemsets in Figure 2.5 is stored in this hash tree and the dark nodes of hash tables are traversed when scanning the transaction 200, {2, 3, 5}.

To determine the maximum frequent itemset, there is an additional trick at the join step. When generating candidate k-itemset X^k from frequent (k-1)-itemsets X_1^{k-1} and X_2^{k-1}, a data structure is considered to create two links from X^k to X_1^{k-1} and to X_2^{k-1}. If the itemset X^k is determined as a frequent itemset, the itemsets linked by it, that is X_1^{k-1} and X_2^{k-1} will be removed directly. But for the determination of closed frequent itemset, the support of X_1^{k-1} and X_2^{k-1} will be considered. If $support(X_i^{k-1}) = support(X^k)$, X_i^{k-1} will be removed for i=1 and 2.

2.3 Algorithm Eclat

Eclat is categorized as a DFS and intersecting strategy by [9]. It is beneficial for mining frequent subset when the patterns are long [11], owing to its depth-first and transaction-ID list intersection mechanisms [4][9]. The search of Eclat follows a depth-first traversal of a prefix tree while the one of Apriori follows a breadth first traversal as it is shown in Figure 2.7.

The transaction-ID list is a vertical layout of transaction [12] which consists of a list of items. Eclat maintains the transaction-ID list for each frequent itemset. Each transaction-ID list records the set of transaction-ID corresponding to the itemset. Figure 2.8 depicts the horizontal and vertical layout of the example dataset shown in Figure 2.2.



(a) Apriori (b) Eclat

Figure 2.7: The traversal method for Apriori and Eclat.

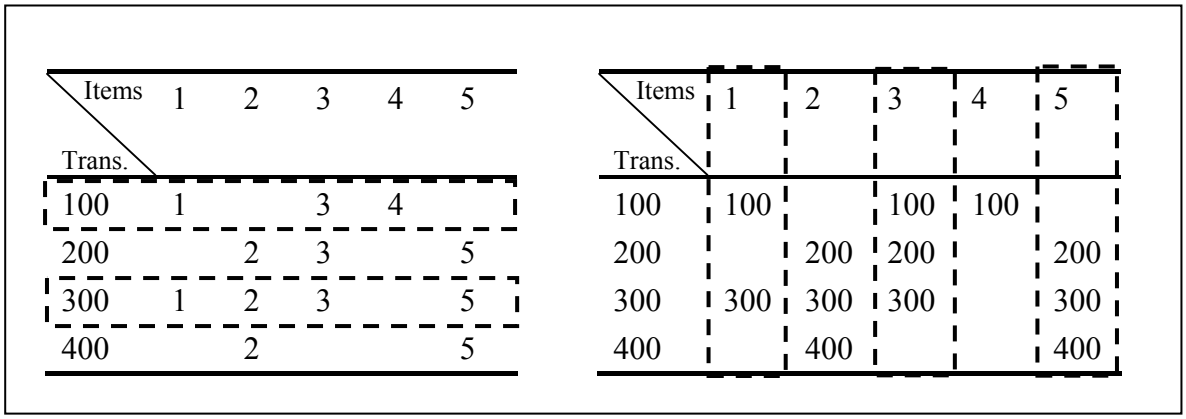


Figure 2.8: Horizontal and vertical database layout.

When generating 2-itemset, Eclat intersects two transaction-ID lists to obtain the transaction-ID list of each 2-itemset. For example, given two 1-itemsets {2} and {3}, and their transaction-ID lists are (200, 300, 400) and (100, 200, 300) respectively. When joining 2-itemset, the transaction-ID list of {2, 3} is (200, 300), and it means the support of {2, 3} is two. Figure 2.9 depicts an example for Eclat to discover frequent subset, while the minimum support is two.

Another technique to represent the transaction-ID list is using a bit matrix [13], in which each row corresponds to an item and each column to a transaction. For example, suppose the transaction-ID list of two 1-itemsets {2} and {3} are (200, 300, 400) and (100, 200, 300) respectively. The bit matrices of these two 1-itemsets are [01110] and [11100]. When generating 2-itemset {2, 3}, we just need an AND operator for the above two bit matrices and the joined bit matrix is [01100] representing the transaction-ID list of (200, 300). Figure 2.10 gives an example of Eclat which represents transaction-ID list as a bit matrix.

Database

TID	Items
100	1, 3, 4
200	2, 3, 5
300	1, 2, 3, 5
400	2, 5

Transaction-ID List for Each Item

Items	TID List	Support
1	100, 300	2
2	200, 300, 400	3
3	100, 200, 300	3
4	100	1
5	200, 300, 400	3

Answer

Frequent Itemset	TID List	Support
{1}	100, 300	2
{1, 3}	100, 300	2
{2}	200, 300, 400	3
{2, 3}	200, 300	2
{2, 3, 5}	200, 300	2
{2, 5}	200, 300, 400	3
{3}	100, 200, 300	3
{3, 5}	200, 300	2
{5}	200, 300, 400	3

Figure 2.9: An example for Eclat.

Transaction-ID List

Items	TID List	Support
1	1010	2
2	0111	3
3	1110	3
4	1000	1
5	0111	3

Answer

Frequent Itemset	TID List	Support
{1}	1010	2
{1, 3}	1010	2
{2}	0111	3
{2, 3}	0110	2
{2, 3, 5}	0110	2
{2, 5}	0111	3
{3}	1110	3
{3, 5}	0110	2
{5}	0111	3

Figure 2.10: An example for Eclat that represent transacton-ID list as a bit matrix.

2.4 Algorithm FP-Growth

FP-Growth is a divide-and-conquer algorithm and belongs to a DFS and counting occurrence strategy for the discovery of frequent itemset without candidate generation [3][8][9]. The database is compacted as a prefix tree in main memory and does not need to be scanned while the process of mining. FP-Growth contains two major phases. The first phase is to construct a tree data structure called FP-Tree, which is a prefix tree, to substitute the original database. The second phase is frequent pattern growth, which divides FP-Tree into conditional FP-Trees, and mines frequent itemsets from each conditional FP-Tree separately.

Original Database

TID	Items
100	{a, c, d, f, g, i, m, p}
200	{a, b, c, f, i, m, o}
300	{b, f, h, j, o}
400	{b, c, k, s, p}
500	{a, c, e, f, l, m, n, p}

Frequent 1-itemset

Frequent Items	Support
f	4
c	4
a	3
b	3
m	3
p	3

Processed Database

TID	Ordered Itemset
100	{f, c, a, m, p}
200	{f, c, a, b, m}
300	{f, b}
400	{c, b, p}
500	{f, c, a, m, p}

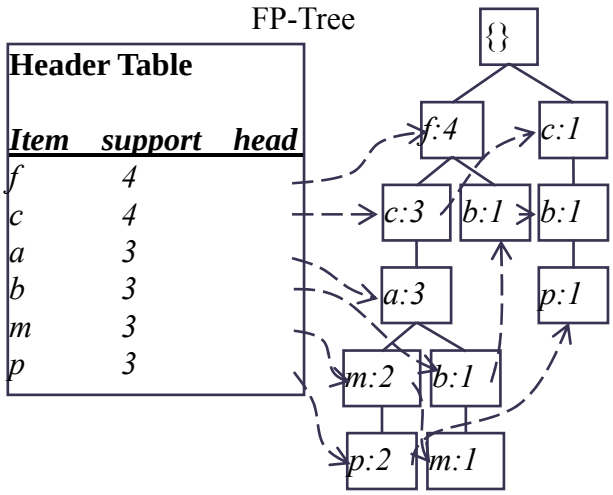


Figure 2.11: An example for FP-Tree construction.

Figure 2.11 is an example to construct the FP-Tree from database. The construction can be divided into two steps. First, it finds the frequent 1-itemset by scanning database once and then sorts them in frequency descending order. Next, it scans database again to construct the prefix tree of each transaction. Each node of the FP-Tree has a link that connects to the header table, which contains the frequent 1-itemset.

After constructing the FP-Tree, the mining process is started. First, for each frequent 1-itemset, constructing the **conditional pattern base**. The conditional pattern base is the set of prefix paths which have the same frequent 1-itemset as their suffix in FP-Tree, and the frequent 1-itemset is called **suffix pattern**. For example, the conditional pattern base of the suffix pattern “m” is {fca:2, fcab:1}, because “m” is the suffix of the two paths <f:4, c:3, a:3, m:2> and <f:4, c:3, a:3, b:1>.

For each conditional pattern base, accumulating the count for each item, and constructing the prefix tree of the frequent items. The prefix tree is the corresponding **conditional FP-Tree** to the conditional pattern base. For example, Figure 2.12 is the conditional FP-Tree of the suffix pattern “m”. The FP-Growth concatenates the suffix pattern with the frequent patterns generated from the conditional FP-Tree. Taking the conditional FP-Tree of “m” as example, given the minimum support is 3 transactions. The frequent patterns are {f}, {c}, {a}, {f, c}, {f, a}, {c, a}, and {f, c, a}, which can be generated by recursively mining the conditional FP-Tree. Finally, the frequent itemsets that have the frequent 1-itemset “m” as their suffix are {f, m}, {c, m}, {a, m}, {f, c, m}, {f, a, m}, {c, a, m}, and {f, c, a, m}.

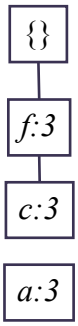


Figure 2.12: The frequent conditional FP-Tree of “m”.

The frequent itemsets are mined in each frequent conditional FP-Tree. Table 2.1 shows the results of mining frequent itemset. The first column is the frequent items (1-itemset), the second column is the conditional pattern base of these frequent items, the third column is the frequent part of the previous column, and the last column is the answer.

Table 2.1: The frequent itemset mined by FP-Growth.

Item	Conditional Pattern Base	Frequent Cond.	Frequent Itemset
f	ϕ	ϕ	ϕ
c	f:3	f:3	fc:3
a	fc:3	fc:3	fca:3,fa:3,ca:3
b	fca:1,f:1,c:1	ϕ	ϕ
m	fca:2,fcab:1	fca:3	fcam:3 ,fcm:3,fam:3,cam:3, fm:3,cm:3,am:3
p	fcam:2,cb:1	ϕ	ϕ

The elements in a set are not ordered in theory. In order to represent a set (itemset), a total order of the elements must been applied. The Apriori and Eclat algorithm discussed in previous sections are both using the lexicographic order, while the FP-Growth algorithm uses the frequency descending order as the total order relation. Figure 2.13 compares the size of FP-Trees constructed by frequency descending order and lexicographic order. The left FP-Tree uses the frequency descending order, and contains 11 nodes (except the root node) while the right one uses lexicographic order, and contains 13 nodes.

Using the frequency descending order can reduce the size of FP-Tree, because it merges the frequent items in fewer nodes effectively. For example, the "f" node is presented once in the (a) tree, but 3 times in (b) tree, in Figure 2.13.

TID	Frequency Descending Order	Lexicographic Order
100	{f, c, a, m, p}	{a, c, f, m, p}
200	{f, c, a, b, m}	{a, b, c, f, m}
300	{ f, b}	{b, f}
400	{c, b, p}	{b, c, p}
500	{f, c, a, m, p}	{a, c, f, m, p}

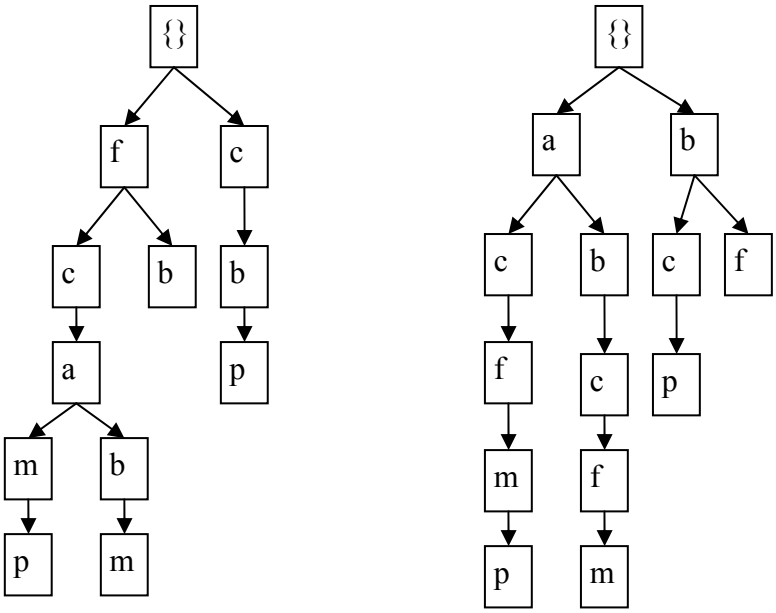


Figure 2.13: Comparing the size of FP-Tree with the order of support of items and the lexicographic. (a) is for frequency order and (b) is for lexicographic order.

2.5 Summary

The problem of association rule mining was introduced by Rakesh Agrawal et al. in 1993 [1]. They published the first algorithm, AIS, to solve the problem of frequent itemset mining. In 1994, they improved the AIS algorithm, and proposed the well-know algorithm, Apriori [2]. Apriori reduces the number of database scan into the length of maximum frequent itemset, and the search space by lessening the number of candidate itemsets in each pass. Recently, there are many algorithms successfully reduce the number of database scan, improve the strategy of traverse search space, and reform the strategy of determine the support value of itemset.

For the breadth-firset search strategy, owing to all of the candidate itemsets for each level have to be discovered, the longer the maximum length of frequent itemset is, the more the candidate itemsets are generated. When discovering the long patterns, the depth-first search strategy is employed to reduce the candidate generation. Eclat is such an algorithm that utilizes the depth-first search strategy.

Moreover, Eclat takes the advantages of depth-first search strategy and utilizes the transaction-ID intersecting strategy to achieve the fewest times of database scan. Eclat maintains transaction-ID lists for every frequent item. The support value of an itemset is calculated from the length of trasaction-ID list of it. Although the partition algorithm [14] also takes the advantages of transaction-ID list intersecting, the breadth-first search strategy causes the expensive computing cost to determine the frequent 2-itemset by intersecting all of the 1-itemset transaction-ID lists [4].

FP-Growth reduces the number of times of database scan in to twice. The first time is to determine the frequent 1-itemset, and the second time is to construct the FP-Tree. Like the other algorithms with depth-first search strategy, FP-Growth finds the maximum frequent itemset firstly. However, FP-Growth does not search the maximum frequent itemsets from 1-itemst, 2-itemset, and so on, but generates them by search the prefix path suffixed by frequent 1-itemset in the FP-Tree. Therefore, FP-Growth economizes the time of generating candidate itemsets, and becomes the most efficient algorithm for frequent itemset mining at present.