CSE304 Introduction to Data Mining
Group #6
Contact point zxzx1825@unist.ac.kr

Assignment #1

1 Apriori Algorithm

1.1 Introduction

Finding a meaningful association rule requires to check all possible item combinations to determine whether they occur together more frequently than a predefined threshold. The brute force approach involves identifying all possible combinations of items. With total n of items, the support of 2^n candidate itemsets must be calculated. Since the entire database must be scanned every time the support of each candidate itemset is calculated, the time complexity is $O((number\ of\ itemsets) \times\ 2^n)$. This approach incurs prohibitive computational costs as the number of unique items increase.

Therefore, an efficient algorithm, Apriori algorithm [1], is introduced to reduce computation required in the brute-force method. The main idea of this algorithm is: *in a database, if a pattern with length k is not frequent, then any super-pattern with length k+1 derived from it is not frequent* Thus, the set of candidate patterns of length (k + 1) can be derived from the set of candidate patterns of length k. In this way, the Apriori algorithm avoids unnecessary calculations.

The Apriori algorithm make a significant contribution to the analysis of market-basket transaction database by generating all significant association rules between products. This method enables market operator to uncover hidden patterns, associations, trends within massive transaction data, thereby increasing its sales and revenues.

1.2 Problem Statement

In this report, we utilize the contrapositive of the Downward Closure Property. By using the fact that itemsets containing infrequent subsets are infrequent, we prune some itemsets having a subset which support is lower than the threshold. This approach contributed to reduce computational time effectively.

Definition 1. *Association rule: relationship between items in a given dataset, in form of X -> Y which X and Y are itemsets.*

Definition 2. *Support: Frequency of occurrence of an itemset in a given dataset*

Definition 3. *Confidence: Ratio of a given rule turns out to be true in practice.*

Theorem 1. *Downward closure property: If an itemset is frequent, all of its subsets must also be frequent.*

Proof. Assume we have a dataset of transactions, and the itemset is called frequent if its support value is greater or equal to minimum support threshold.

Let F be a frequent itemset, which meets the minimum support threshold. Let S be the subset of F. All transactions including F must include S, because S is the subset of F. In addition, there can exist transactions that include S but not all items in F. so, $Support(F) \leq Support(S)$. Therefore, if F meets the minimum support threshold, then S must meet the minimum support threshold, indicating that S is frequent as well. □

1.3 Algorithm Description and Analysis

Algorithm 1 Apriori algorithm

```
1: procedure APRIORI($minsup$)
2:     $L_1 \leftarrow \{large\ 1 - itemset\}$                          ▷ A set of 1-item
3:     while $L_{k-1}$ is not empty do
4:         $C_k \leftarrow$ SELFJOIN($L_{k-1}$)                        ▷ A set of candidate k-items
5:         $C_k \leftarrow$ PRUNE($L_{k-1}, C_k$)                      ▷ Delete infrequent itemset
6:         for each transactions $t \in D$ do                         ▷ D is Database
7:             for each candidates $c \in subset(C_k, t)$ do          ▷ Candidates contained in t
8:                 $c.count \leftarrow c.count + 1$
9:         $L_k = \{c \in C_k \mid c.count \geq minsup\}$             ▷ A set of frequent k-items
10:        $k \leftarrow k + 1$
11:    return $\bigcup_k L_k$                                          ▷ Return all frequent itemsets
12: procedure SELFJOIN($L_{k-1}$)
13:    for each pair of itemsets $P_1, P_2\ in\ L_{k-1}$ do
14:        if $length(P_1 \cup P_2) = k$ and $(P_1 \cup P_2) \notin C_k$ then
15:            $C_k \leftarrow C_k \cup \{P_1 \cup P_2\}$
16:    return $C_k$
17: procedure PRUNE($L_{k-1}$,$C_k$)
18:    for $each\ itemset\ c \in C_k$ do
19:        for All $(k-1) - subsets\ s\ of\ c$ do
20:            delete $c$ from $C_k$ if $s \notin L_{k-1}$
21:    return $C_k$
```

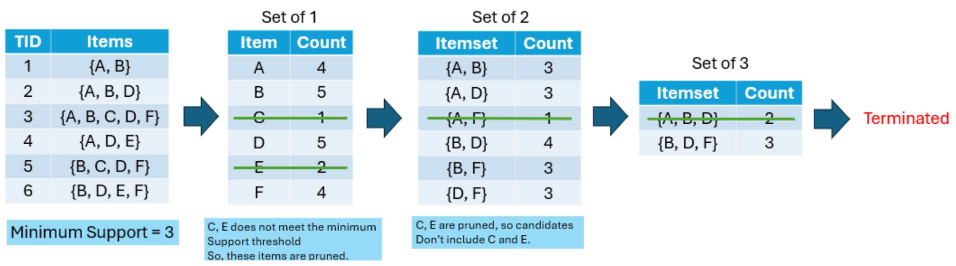
In this section, we introduce an algorithm developed to find frequent itemsets without computing the support of all possible itemsets. The procedure of this algorithm are detailed in Algorithm 1. The core idea of this algorithm is to remove itemsets that have any infrequent subsets from the support calculation.

The algorithm starts by making a initial frequent itemset set L_1 that has a set of single item (line 3). A loop(line 4-12) iterates while L_{k-1} is not empty. In every iteration, an algorithm makes C_k, which is a set of candidate k-item sets by using SELFJOIN (line 5) and PRUNE procedure(line 6). Next, for each transactions t in database(line 7-9), we count the number of each candidates in the subset of the C_k that contains t (line 8-9). And make a set of frequent k-item sets which count is equal or bigger than minimum support(line 10) and k is increased by 1. After iteration, Apriori algorithm return union set of the frequent itemset (line 12).

Procedure SELFJOIN is about generating candidate itemsets with k items based on itemsets in L_{k-1} which has k-1 items each. for each pair of itemsets P_1, P_2 (line 14), check whether the union of two itemsets has size of k or not, also excluding the case that union is already in candidate set(line 20-21). if the size is k, put the union of two itemsets into set of candidate k-items C_k(line 15). Finally, it returns C_k (line 16).

Procedure PRUNE is about pruning itemsets based on contrapositive of Downward Closure Property. Which is, "If at least one of the subset of itemset is not frequent, then that itemset is not frequent". For each itemsets in C_k(line 18), for all $(k-1)-subsets$ of c (line 19), delete c from C_k if s is not in L_{k-1} (line 20). Than, return C_k (line 21).

1.4 Example



TID	Items
1	{A, B}
2	{A, B, D}
3	{A, B, C, D, F}
4	{A, D, E}
5	{B, C, D, F}
6	{B, D, E, F}

Minimum Support = 3

Set of 1

Item	Count
A	4
B	5
~~C~~	~~1~~
D	5
~~E~~	~~2~~
F	4

C, E does not meet the minimum Support threshold
So, these items are pruned.

Set of 2

Itemset	Count
{A, B}	3
{A, D}	3
~~{A, F}~~	~~1~~
{B, D}	4
{B, F}	3
{D, F}	3

C, E are pruned, so candidates Don't include C and E.

Set of 3

Itemset	Count
~~{A, B, D}~~	~~2~~
{B, D, F}	3

Figure 1: Algorithm procedure

Example 1. *Figure 1 shows how Apriori algorithm works. Initially, count each item's frequency and prune items which do not meet the minimum support threshold, specifically items C and E in the given example. Next, with remaining items Generate all two-item sets, and prune the itemsets which do not meet the minimum support threshold, specifically itemset {A, F} in the given example. Then, process is repeated by increasing size of the itemsets by 1 per iteration. It is terminated when there is no more itemsets to generate.*

1.5 Advantages and Disadvantages

1.5.1 Advantages

- **Simplicity and ease of implementation**: The Apriori algorithm offers a straightforward and intuitive method, making it relatively easier to understand and implement compared to other methods.
- **Fast execution time**: By pruning infrequent itemsets, Apriori algorithm can discover frequent itemsets effectively.
- **Application in various fields**: This concept successfully applied in various fields, including product recommendation through shopping cart analysis, discovery of associations between diseases and symptoms, and analysis of web usage patterns.

1.5.2 Disadvantages

- **Manual setting of Threshold**: Setting the threshold too high may prune the important frequent itemsets , while setting it too low may allow unnecessary itemsets to pass through, increasing the running time.

- **Computational complexity**: As the number of item types increases, number of candidate itemsets significantly grows, leading to much more longer computational time and larger memory usage.
- **Discovery of complex pattern is limited**: The Apriori algorithm focuses on analyzing binary relationships between items. So, if there is a more complex relationship between items, the Apriori algorithm cannot find an association rule.

1.6 Experiment

In this section, we present our experimental approaches and findings about the Apriori algorithm. Our evaluation metric is running-time based measurement, and all experiments were conducted on Intel(R) Core(TM) i5-8250U CPU@1.60GHz, 8GB DDR3 PC3-14900@1867MHz, Oracle OpenJDK 11, and Windows 11. Our implementation code and dataset can be accessed at `https://github.com/thinkin9/DataMining_G6`

1.6.1 Dataset

Our data sets are generated using techniques outlined in [1], reflecting transactional data characteristics. Our data generation assumptions are summarized as follows: (1) We assumed the average transaction size follows a Poisson distribution with mean T. Given each item is chosen with probability p among N items, the expected number of items in a transaction follows a binomial distribution with parameters N and p. Therefore, the transaction size can be approximated by a Poisson distribution with mean Np. (2) The association between items are designed by selecting some items from previous transactions for the current transactions, allowing for shared items across transactions and reflecting arbitrary association patterns. To determine proper, varying reflection ratio of the previous transaction, we assumed an exponentially distributed random variable with a mean equal to the correlation level(0.5).

Detailed synthetic data parameters and configurations of our data sets are presented in Table 1.

T	Average size of the transactions
N	Number of items
D	Number of transactions

(a) Parameters

Name	T	N	D
T10.N100.D10K (base)	10	100	10K
T2.N100.D10K	2	100	10K
T5.N100.D10K	5	100	10K
T15.N100.D10K	15	100	10K
T20.N100.D10K	20	100	10K
T10.N100.D1K	10	100	1K
T10.N100.D20K	10	100	20K
T10.N100.D50K	10	100	50K
T10.N100.D100K	10	100	100K

(b) Data set configurations

Table 1: Detailed synthetic data parameters and configurations

1.6.2 Experiment Data

To evaluate the performance of the Apriori algorithm, we conducted experiments based on execution time using the data in Section 1.6.1. We varied parameters including the average size of transactions, the number of transactions, and minimum support value. Figure 2 shows the results of the execution time.

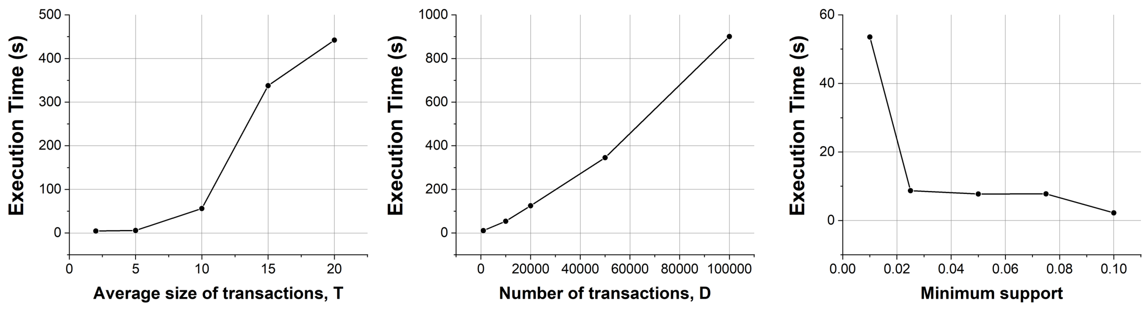


Figure 2: Execution time with respect to average size of transactions(left), number of transactions(center), and minimum support(right)

Average size of transaction: We varied the average size of transactions from 2 to 20 while fixing the number of items to 100, the minimum support threshold to 0.01, and the number of transactions to 10000. With the increase of average size of transactions, The result shows significant increase in execution time, attributed to the expansion of candidate set.

Number of transactions: We varied the number of transactions from 1000 to 100000 while fixing the number of items to 100, the minimum support threshold to 0.01, and the average size of transactions to 10. The result shows linear increase in execution time. Amount of transactions does not directly influence the average size of the frequent itemsets, but increases the time of scanning, resulting in a linear relation with execution time.

Minimum support: We varied the minimum support threshold from 0.01 to 0.1 using the T10.N100.D10K dataset. With the increase of minimum support threshold, The result shows a significant decrease in execution time, attributed to the increase of the pruned candidate itemsets.

1.7 Conclusion and Future Directions

In conclusion, the Apriori algorithm significantly increase efficiency by pruning infrequent itemsets, thereby boosting the discovery of meaningful association rules. While it is simple and fast, it has limitations, including increasing computational complexity as data grows, need of finding appreciate minimum support threshold, and difficulty of identifying complex patterns. Future directions should focus on designing a new algorithm to solve the limitations of the Apriori algorithm, especially the increase of computational complexity with large datasets.

2 FP-growth Algorithm

2.1 Introduction

Many studies had been conducted to optimize the Apriori algorithm, such as Direct Hashing and Pruning[3] for efficient candidate set generation and Partition algorithm[4] for reducing both CPU and I/O overhead. However, the Apriori approach has a significant drawback in candidate set generation when dealing with large-scale data involving a huge number of candidate sets.

The FP-growth [2] algorithm was introduced to reduce the computational cost associated with candidate set generation and test in large databases containing numerous frequent patterns and long patterns. This approach utilizes three core techniques: (1) *FP-tree*, a compact tree structure, (2) FP-tree based pattern growth strategy, and (3) Partition-based, divide-and-conquer approach. With these techniques, the FP-growth method achieved a significant improvement in performance for both mining long and short frequent patterns in large database.

2.2 Problem Statement

In Section 1.2, we addressed the problem statement of association rule mining. Now, we detailed the specific limitations of the Apriori algorithm in large database:

- Candidate set generation is still expensive. Canditate patterns of length $(k+1)$ are derived from those of length k, resulting in accumulation of candidate patterns as the size of it increase. While Pruning reduces unnecessary candidate patterns, discovering frequent patterns from large candidates and lengthy transactions requires to generate almost exponential number of candidates.
- Repeated scanning for database is expensive. Total k times of scanning for database are required to find frequent k-items sets from candidate patterns of length k.

The FP-growth approach utilize the following techniques to address problems of the Apriori approach. These techniques contribute to the elimination of candidate set generations and tests and a substantial reduction of computational costs for scanning the entire database.

- FP-tree, a compact tree structure for frequent patterns. This structure captures the occurrence pattern of frequent items whose frequency exceeds the minimum support.
- FP-tree based pattern growth strategy. This strategy involves incrementally extending a set of frequent patterns by only examining its conditional pattern base which contains a set of frequent items co-occurring with the suffix pattern.
- Partitioning-based, divide-and-conquer approach. Within FP-growth, the frequent k-itemset mining problem can be converted into a sequence of k frequent 1-itemset mining problems with their corresponding conditional pattern bases.

2.3 Algorithm Description and Analysis

The fundamental concept behind FP-growth approach is to compress database into a compact and efficient data structure for frequent pattern tree, FP-tree. To acheieve this goal, the FP-growth algorithm involves three key steps: FP-tree construction, Conditional FP-trees construction, and Mining frequent patterns.

Since only the frequent itmes play a role in the frequent pattern mining process, the first step is to make the frequent item list. In this F-List, each items are sorted in decreasing order of frequency with obtaining their count value.

Algorithm 2 FP-Tree construction

```
1: procedure FP-Tree construction($freqTransactions$)
2:     $node \leftarrow \{\}$
3:     for each transaction $in\ freqTransactions$ do
4:         for each item $in\ transaction$ do
5:             if exist $node.child.name$ = item then
6:                 $node.child.count + +$
7:             else
8:                 $child \leftarrow$ new FP node with item name
9:                 set $child.parent = node$ & $node.child = child$
10:                if item name exist in $Header\ Table$ then
11:                    find the $LastNode$ of item name
12:                    $LastNode.next = child$
13:                else
14:                    put $child$ in $Header\ Table$
15:                $node\ =\ child$
```

The algorithm (Algorithm 2) takes frequent transactions as input, which are transactions filtered and sorted by a frequent list. For each transaction, if the item is in the children of the current node, we simply increment count value of the child. If not, we first check if a node with the item's name exists in the header table. If not, we add the new node to the header table; otherwise, we traverse to the last node pointing to that item and add the new node after it. This ensures that all items in the tree are represented in the header table, providing access to all items through their name.

Building conditional FP trees follows the steps of FP-Tree construction but requires conditional pattern bases for each item instead of frequent transactions. Then, we proceed the removal process for items whose count values fall below the threshold.

The pattern growth algorithm (Algorithm 3) adopts a method of gradually expanding a pattern from the prefix. When initially called, it uses the FP tree generated from transactions and a null prefix.

If all nodes in the FP tree have only one child, the tree is said to have a single path. If the tree has a single path, we create subpaths for all combinations of nodes along that path and set the support value of the new subpath to the minimum support value of the node in the subpath. These subpaths are combined with the prefix to form the final patterns.

If the FP tree does not have a single path, pattern bases are created for all item sets in the header table, and conditional FP trees are constructed accordingly. Each item is added to the prefix during this process, and the support value of the prefix is updated to the support value of the new item. By storing the prefix, all complete information for the divide-and-conquer process steps can be maintained. This process continues iteratively, creating new pattern bases and trees for each new prefix.

Algorithm 3 FP-Growth

```
 1: $R \leftarrow ResultList[]$
 2: procedure FP-GROWTH($Tree$, $prefix$, $HeaderTable$)
 3:     if $Tree$ is single path then
 4:         for each subpath of the nodes in the single path do
 5:             $newpattern = subpath \cup prefix$
 6:             $newpattern.support$ = minimum support of nodes in subpath
 7:             $R.add(newpattern)$
 8:     else
 9:         for each item in $HeaderTable$ do
10:             $prefix = item \cup prefix$
11:             $prefix.support$ = item.$support$
12:             construct conditional FP-Tree $Tree_{item}$
13:             FP-Growth ($Tree_{item}$, $Prefix$, $HeaderTable_{item}$)
14:             $R.add(prefix)$
```

2.4 Example for FP-tree Construction

TID	Name	Items
1	Junghoon Kim	Chicken breast, Coke, Beef, Rice
2	Hyewon Kim	Beef, Soju, Lettuce, Potato, Rice,
3	Hyeonjin Jo	Apple, Banana, Lemon, Tomato, Rice
4	Sungwon Bae	Salmon, Rice, Beef, Coke, Soju, Pork
5	Hyunju Kim	Beef, Coke, Apple, Olive oil
6	Junseo Kim	Rice, Pork, Soju, Cheese, Beef

(a) Market-Basket Transactions.

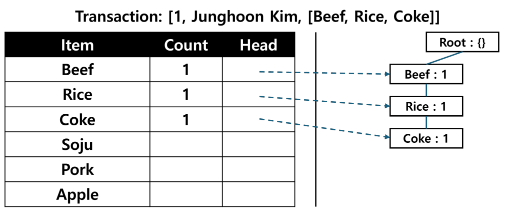
Min_support: 2 | F-list: Beef - Rice - Coke - Soju - Pork - Apple

TID	Name	Ordered Items
1	Junghoon Kim	Beef, Rice, Coke
2	Hyewon Kim	Beef, Rice, Soju
3	Hyeonjin Jo	Rice, Apple
4	Sungwon Bae	Beef, Rice, Coke, Soju, Pork
5	Hyunju Kim	Beef, Coke, Apple
6	Junseo Kim	Beef, Rice, Soju, Pork

(b) Ordered Market-Basket Transactions.

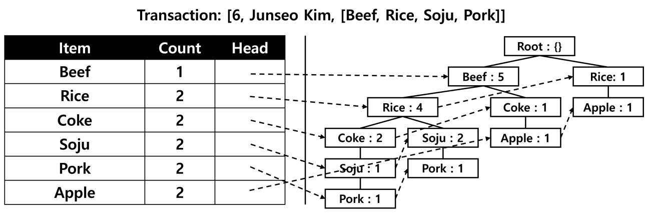
Figure 3: Real-world Market-basket transactions examples.

Example 2. *Figure 3a shows an example of the raw market-basket transaction database. Figure 3b shows an example of the ordered market-basket transaction database with minimum_support value as 2 and Frequent item lists as [Beef, Rice, Coke, Soju, Pork, Apple].*


Transaction: [1, Junghoon Kim, [Beef, Rice, Coke]]

Item	Count	Head
Beef	1	
Rice	1	
Coke	1	
Soju		
Pork		
Apple		

(a) FP-tree with the first transaction.

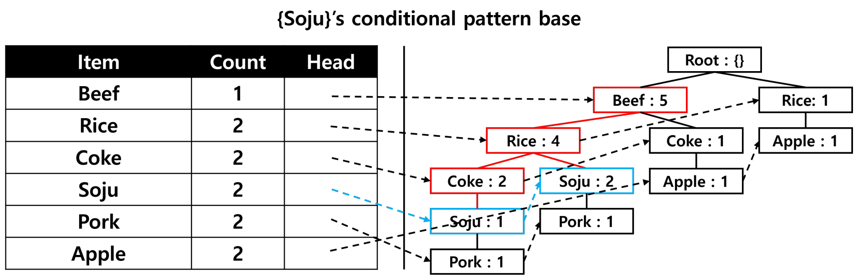

Transaction: [6, Junseo Kim, [Beef, Rice, Soju, Pork]]

Item	Count	Head
Beef	1	
Rice	2	
Coke	2	
Soju	2	
Pork	2	
Apple	2	

(b) FP-tree with the last transaction.

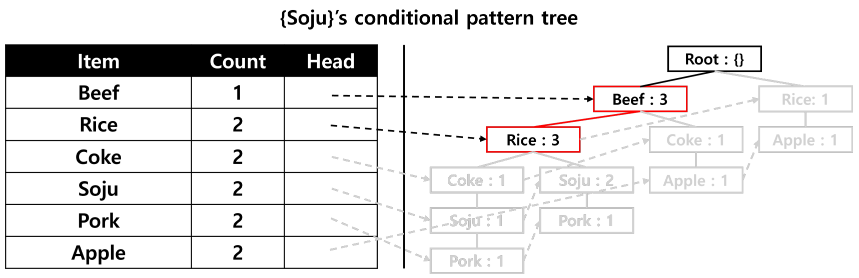
Figure 4: Building FP-tree examples.

Example 3. *Figure 4a shows an example of processing the first transaction data, [1, Junghoon Kim, [Beef, Rice, Coke]]. Figure 4b shows an example of the last transaction data, [6, Junseo Kim, [Beef, Rice, Soju, Pork]].*



Item	Count	Head
Beef	1	
Rice	2	
Coke	2	
Soju	2	
Pork	2	
Apple	2	

(a) Soju's conditional pattern base.



Item	Count	Head
Beef	1	
Rice	2	
Coke	2	
Soju	2	
Pork	2	
Apple	2	

(b) Soju's conditional pattern tree.

Figure 5: Building conditional pattern base and conditional pattern tree examples.

Example 4. *Figure 5a shows the set of prefix paths in the FP-tree for "Soju". <Beef, Rice, Coke>: 1 and <Beef, Rice>: 2. Figure 5b shows the FP-tree from the given conditional pattern bases with minimum_support value as 2. <Beef-Rice>: 3.*

2.5 Advantages and Disadvantages

2.5.1 Advantages

- **Divide-and-conquer approach**: The frequent k-itemset mining task is converted into a series of k frequent 1-itemset mining tasks with their pattern bases and prefixes. This approach enables the algorithm to handle larger datasets more effectively.
- **Efficient Memory Usage**: FP-Tree, a compact tree structure, minimizes memory usage to store frequent patterns and expedites operations within FP-Growth.
- **Reduced database scans**: Only two times of database scan are required: one for finding frequent 1-itemsets and another for inserting each transaction into FP-Tree.

2.5.2 Disadvantages

- **Challenging implementation**: Implementing FP-Growth requires a more code and entails a more complex implementation compared to the Apriori algorithm, despite the apparent simplicity of its algorithmic approach.
- **Handling large number of unique items**: When the dataset contains a large number of unique items, the FP-Tree can become overly complex, which leads to decreased efficiency of the FP-Tree as it is failed to utilize the advantage of it.
- **Handling imbalanced item distribution**: Similar to the previous disadvantage, the FP-Tree can become more skewed or imbalanced data structure with an disproportionate distribution of items, resulting in decreased efficiency of the FP-Tree.

2.6 Experiment

Since the experimental results of the FP-growth algorithm exhibit a similar pattern to that of the Apriori (Figure 6), in this section, we mainly focus on the comparison between two models - FP-growth and Apriori - based on the same criteria as in Section 1.6. Experimental environments and data set configurations are the same as specified in Section 1.6.1 and 1.6.2.

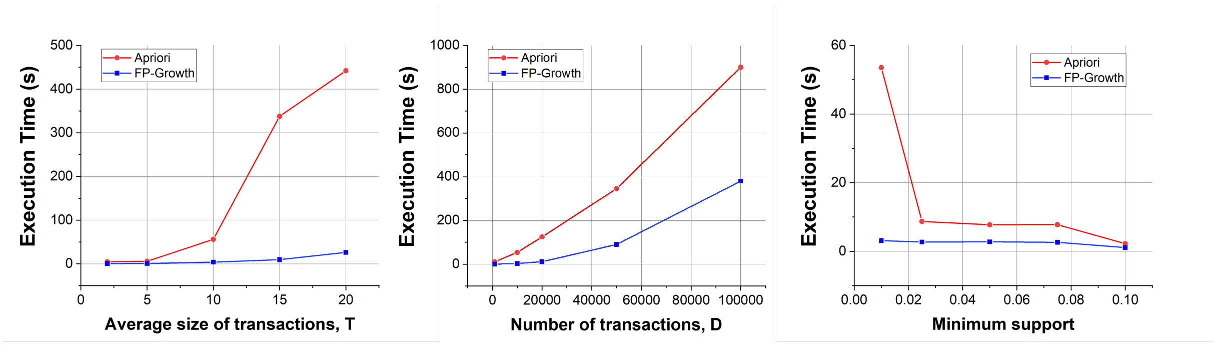


Figure 6: Execution time comparision between the Apriori and the FP-growth with respect to average size of transactions(left), number of transactions(center), and minimum support(right)

Average size of transaction: Even the average size of transactions increases, the FP-Growth algorithm maintains the number of database scans and only increases the height of the tree. However, the Apriori algorithm requires an increased number of candidate set generations and database scans proportional to the increase in the average size of transactions, resulting in a significant difference in execution time.

Number of transactions: In the case of FP-Growth, unlike the average size of transactions, the increase in scan time of the database itself led to a steeper linear increase. As the number of transactions increases, the gap in execution time between FP-Growth and Apriori becomes increasingly larger, indicating that FP-Growth scales much better.

Minimum support: The FP-Growth algorithm shows better scalability than the Apriori algorithm, exhibiting almost no differences in execution time. Since Apriori relies on candidate generation to find frequent patterns, lowering the threshold requires handling a vast amount of candidates, thus demanding more time.

The experimental results confirm that the FP-Growth algorithm is more efficient than the traditional Apriori algorithm in finding frequent patterns.

2.7 Conclusion and Future Directions

We provided an in-depth analysis of the FP-growth algorithm and implemented this algorithm using Java. The compact tree structure for frequent patterns, FP-tree, along with the efficient pattern-growing method, FP-growth, resulting in a significant improvement in execution time compared to the Apriori algorithm. In summary, FP-growth has demonstrated efficiency in handling large-scale datasets and finding frequent patterns.

Future directions can be addressed by considering several aspects of the algorithm. First, optimization techniques can further enhance it's performance, such as, parallelization and bitwise storage of nodes. Moreover, novel algorithms for constructing FP-tree and growing frequent patterns for large datasets having large number of unique items or imbalanced item distribution need to be developed to address the FP-growth's limitation.

References

[1] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499. Santiago, 1994.

[2] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *ACM sigmod record*, 29(2):1–12, 2000.

[3] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash-based algorithm for mining association rules. *Acm sigmod record*, 24(2):175–186, 1995.

[4] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Databases (VLDB)*, pages 432–444, 1995.

Name	**Sign**	**Individual Contribution**	**Percentage**
SUNGWON BAE	[illegible]	Apriori draft and function code	25%
JUNSEO KIM	[illegible]	Apriori implementation and Text writing	25%
HYEONJIN JO	[illegible]	FP-Tree Generation and Dataset Generation	25%
HYUNJU KIM	[illegible]	Conditional FP-Tree and Pattern Growth	25%

Table 2: Percentage of the contribution must be 100% in total. For writing individual contribution, refer to `https://www.elsevier.com/researcher/author/policies-and-guidelines/credit-author-statement`