University of Dhaka

Department of Computer Science and Engineering

CSE-4255 : Introduction to Data Mining and Warehousing Lab

Lab Report: Comparative Analysis of Apriori Algorithm and Frequent Pattern Growth Algorithm

Submitted By:

Name : Zisan Mahmud

Roll No : 23

Submitted On:

APRIL 30, 2025

Submitted To:

Dr. Chowdhury Farhan Ahmed

Md. Mahmudur Rahman

Contents

1 ABSTRACT

This report presents a comparative analysis of Apriori and Frequent Pattern Growth (FP-Growth) algorithm. Total five datasets (Dense: mushroom, chess, Sparse: T10I4D100K, kosarak, retail) are used and both algorithms are implemented on the datasets at various support levels to determine the most efficient method. The time and memory usage complexity of the two algorithms is also analyzed.
keywords: Apriori Algorithm, Frequent Pattern Growth (FP-Growth) Algorithm.

2 INTRODUCTION

Finding frequent patterns plays an essential role in data mining. Association rule mining involves finding associations, patterns or correlation among data sets by applying if-then rules and drawing meaning and profitable conclusions from them. Two algorithm were discovered to solve the problem of association rule mining and they are the Apriori Algorithm and the FP-Growth Algorithm.

2.1 Apriori Algorithm

The apriori algorithm perform mining through generation of frequent item set by scanning the database multiple times. It uses a "bottom up" approach where candidate generation is done by extending frequent subset of an itemset one step at a time. It uses breadth-first search (BFS) approach and generate candidate itemset K from their previous subset k-1. Due to frequent search in database, it was proved inefficient. To improve the efficiency of the level-wise generation for frequent itemsets, Apriori property is used to reduce the search space. The property states that all the nonempty subset of frequent itemset is also frequent. Although the efficiency of Apriori algorithm is not good as others, it is simple and effective to understand.

2.2 FP-Growth Algorithm

The most notable algorithm is the FP-Growth, this algorithm uses a "divide and conquer" approach to efficiently generate a frequent pattern tree. It is implemented based on Apriori algorithm but generates candidate sets more efficiently than Apriori. FP-Growth changed the underlying data structure into in a tree which allows for faster and efficient can. The FP-Growth algorithm work by first counting the occurrence of each individual item in the database, then it will use the minimum support as a threshold to filter out non-frequent items, it will then sort the items depending on their occurrences and finally create the tree and add the transactions one after the other.

3 IMPLEMENTATION DETAILS

This section outlines the implementation specifics for both the Apriori and FP-Growth algorithms, including data handling, core processes, and performance measurements.

3.1 Apriori Algorithm

The Apriori algorithm implementation leverages key data structures and optimization techniques to efficiently mine frequent itemsets from transaction databases. Central to the implementation is the candidate generation process, which follows different strategies based on the itemset length k.

- **For** $k = 1$: The implementation initializes candidates from unique items in the transaction database:

$$\text{items} = \{[item] \mid item \in \text{unique\_items}\}$$

- **For** $k = 2$: The implementation generates candidate pairs by combining frequent 1-itemsets. Each candidate is created by merging two 1-itemsets and ensuring the resulting itemset has exactly length 2:

$$\text{candidates} = \{\text{sort}(\text{set}(itemset_i \cup itemset_j)) \mid i < j \wedge |\text{set}(itemset_i \cup itemset_j)| = 2\}$$

- **For** $k > 2$: The implementation employs the standard Apriori join-prune approach.
  - *Join step*: Combine $(k-1)$-itemsets that share the first $(k-2)$ items and have different $(k-1)$-th items:

    $$\text{new\_candidate} = itemset_i[1:k-2] \cup \{itemset_i[k-1], itemset_j[k-1]\}$$

    where $itemset_i[1:k-2] = itemset_j[1:k-2]$ and $itemset_i[k-1] < itemset_j[k-1]$.
  - *Prune step*: Eliminate candidates whose $(k-1)$-subsets are not all frequent:

    $$\text{candidate is valid if } \forall \text{subset} \subset \text{candidate}, |\text{subset}| = k-1 \Rightarrow \text{subset} \in \text{frequent\_itemsets}$$

Support calculation is optimized using set operations. Itemsets are converted to `frozenset` for immutability, and containment in transactions is checked with `issubset()`:

$$\text{support}(X) = \frac{|\{t \in T \mid X \subseteq t\}|}{|T|} \times 100\%$$

The core algorithm iteratively builds frequent itemsets level by level. At each level k:

1. Generate candidate k-itemsets

2. Calculate support for each candidate
3. Retain only candidates meeting minimum support threshold
4. Generate candidates for level $k + 1$
5. Repeat until no more frequent itemsets are found

Performance optimization techniques include:

- Consistent sorting of itemsets for reproducible results
- Efficient conversion between data structures (lists, sets, tuples) based on operation
- Early pruning using the Apriori principle before support calculation
- Tracking memory and execution time using `memory_profiler` and `time` modules

Result structure:

- Frequent itemsets are maintained as a list of dictionaries
- Each dictionary at index $k - 1$ maps frequent k-itemsets to tuples of (absolute_support, percentage_support)
- Results are displayed with PrettyTable for clarity
- Includes memory usage, execution time, and itemset distribution statistics

3.2 FP-Growth Algorithm

Similar to the Apriori approach, the FP-Growth implementation starts by reading the transaction data into a list of item lists. Memory and time measurements are recorded at the beginning using `psutil.Process().memory_info().rss` and `time.time()`, respectively.

The FP-Tree is built through the following steps:

- A first scan of the dataset counts the frequency of each item.
- Items not meeting the minimum support threshold are pruned.
- Remaining items are stored in a header table with a format of `item: [count, node_link]`.

Tree construction proceeds by inserting each transaction:

- Infrequent items are removed from each transaction, and the remaining items are sorted in descending order of global support.
- Transactions are recursively inserted into the FP-Tree using the `insert_tree` function, updating counts and maintaining header links.

Frequent pattern mining is performed by traversing the header table:

- For each item, its conditional pattern base is constructed by following node links and extracting prefix paths.
- A conditional FP-Tree is generated from these paths.
- If the conditional tree is non-empty, recursion is applied to mine further patterns, combining base items with prefixes.

Finally, all frequent patterns are collected into a comprehensive list. The final memory usage, execution time, and the number of patterns mined are calculated and printed in the `__main__` block.

4 EXPERIMENTAL RESULT

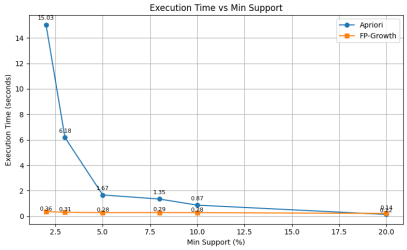
Here I present the experimental result of the implementation of Apriori Algorithm and FP-Growth Algorithm. Test were run on a Laptop with 2.30 GHz Intel(R) Core(TM) i7 - 11800H (11th Gen) processor and 8Gbyte Ram. The operating system was Windows 11. The datasets were divided into 2 parts - Dense and Sparse. The results are presented in the form of line graphs and bar graphs.

Sparse Dataset

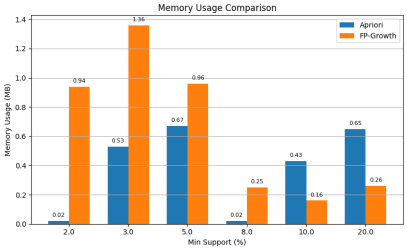
In sparse dataset the number of frequent itemset is small.

Retail

It has total 88,162 transactions and 16,470 distinct items. From the experimental result we can see that the FP-Growth algorithm is faster than Apriori, but consumes more memory than Apriori.



(a) Execution Time



(b) Memory Usage

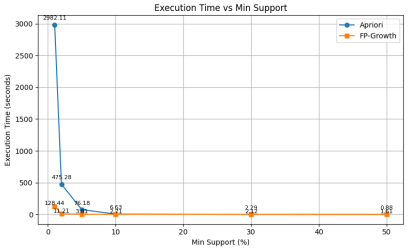
Min_sup	Metric	Apriori	FP-Growth
2.0%	Runtime (s)	15.03	0.36
	Memory (MB)	0.02	0.94
	Frequent Items	55	55
3.0%	Runtime (s)	6.18	0.31
	Memory (MB)	0.53	1.36
	Frequent Items	32	32
5.0%	Runtime (s)	1.67	0.28
	Memory (MB)	0.67	0.96
	Frequent Items	16	16
8.0%	Runtime (s)	1.35	0.29
	Memory (MB)	0.02	0.25
	Frequent Items	13	13
10.0%	Runtime (s)	0.87	0.28
	Memory (MB)	0.43	0.16
	Frequent Items	9	9
20.0%	Runtime (s)	0.14	0.22
	Memory (MB)	0.65	0.26
	Frequent Items	3	3

(c) Comparison Table

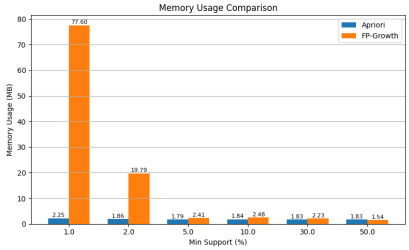
Figure 1: Performance Comparison of Apriori vs FP-Growth (Retail Dataset)

Kosarak

It has total 9,90,002 transactions and 41,270 distinct items. From the experimental result we can see that the FP-Growth algorithm is faster than Apriori, but consumes more memory than Apriori.



(a) Execution Time



(b) Memory Usage

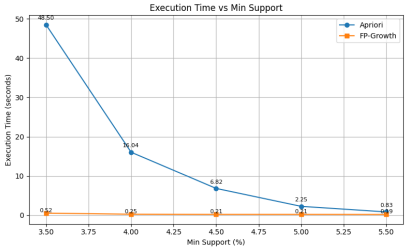
Min_sup	Metric	Apriori	FP-Growth
1.0%	Runtime (s)	2982.11	128.44
	Memory (MB)	2.25	77.60
	Frequent Items	383	383
2.0%	Runtime (s)	475.28	11.21
	Memory (MB)	1.86	19.79
	Frequent Items	121	121
5.0%	Runtime (s)	76.18	3.81
	Memory (MB)	1.79	2.41
	Frequent Items	33	33
10.0%	Runtime (s)	6.63	2.71
	Memory (MB)	1.84	2.48
	Frequent Items	9	9
30.0%	Runtime (s)	2.29	2.12
	Memory (MB)	1.83	2.23
	Frequent Items	4	4
50.0%	Runtime (s)	0.88	1.61
	Memory (MB)	1.83	1.54
	Frequent Items	1	1

(c) Comparison Table

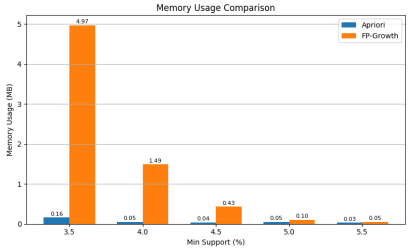
Figure 2: Performance Comparison of Apriori vs FP-Growth (Retail Dataset)

T10I4D100K

It has total 1,00,000 transactions and 870 distinct items. From the experimental result we can see that the FP-Growth algorithm is faster than Apriori, but consumes more memory than Apriori for smaller minimum support.



(a) Execution Time



(b) Memory Usage

Min_sup	Metric	Apriori	FP-Growth
3.5%	Runtime (s)	48.50	0.52
	Memory (MB)	0.16	4.97
	Frequent Items	40	40
4.0%	Runtime (s)	16.04	0.25
	Memory (MB)	0.05	1.49
	Frequent Items	26	26
4.5%	Runtime (s)	6.82	0.21
	Memory (MB)	0.04	0.43
	Frequent Items	17	17
5.0%	Runtime (s)	2.25	0.21
	Memory (MB)	0.05	0.10
	Frequent Items	10	10
5.5%	Runtime (s)	0.83	0.19
	Memory (MB)	0.03	0.05
	Frequent Items	6	6

(c) Comparison Table

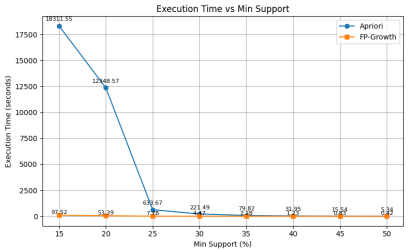
Figure 3: Performance Comparison of Apriori vs FP-Growth (Retail Dataset)

Dense Dataset

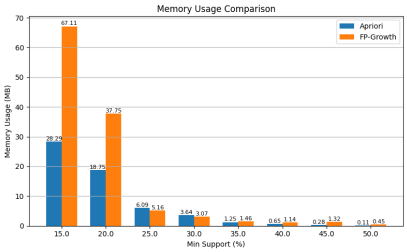
In sparse dataset the number of frequent itemset is large. We can visuzalize the performance gap between Apriori and FP-Growth more precisely for dense dataset.

Mushroom

It has total 8124 transactions and 119 distinct items. From the experimental result we can see that the FP-Growth algorithm is much faster than Apriori, but consumes more memory than Apriori.



(a) Execution Time



(b) Memory Usage

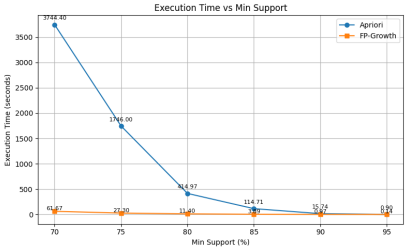
Min_sup	Metric	Apriori	FP-Growth
15.0%	Runtime (s)	18311.55	97.52
	Memory (MB)	28.29	67.11
	Frequent Items	98575	98575
20.0%	Runtime (s)	12348.57	53.39
	Memory (MB)	18.75	37.75
	Frequent Items	53583	53583
25.0%	Runtime (s)	633.67	7.78
	Memory (MB)	6.09	5.16
	Frequent Items	5545	5545
30.0%	Runtime (s)	221.49	4.47
	Memory (MB)	3.64	3.07
	Frequent Items	2735	2735
35.0%	Runtime (s)	79.82	2.28
	Memory (MB)	1.25	1.46
	Frequent Items	1189	1189
40.0%	Runtime (s)	31.95	1.23
	Memory (MB)	0.65	1.14
	Frequent Items	565	565
45.0%	Runtime (s)	15.54	0.83
	Memory (MB)	0.28	1.32
	Frequent Items	329	329
50.0%	Runtime (s)	5.34	0.42
	Memory (MB)	0.11	0.45
	Frequent Items	153	153

(c) Comparison Table

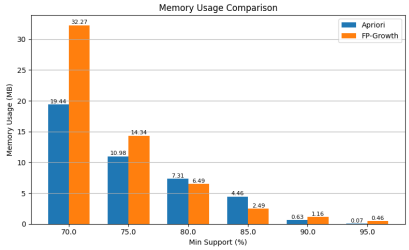
Figure 4: Performance Comparison of Apriori vs FP-Growth (Retail Dataset)

Chess

It has total 3196 transactions and 75 distinct items. From the experimental result we can see that the FP-Growth algorithm is faster than Apriori, but consumes more memory than Apriori.



(a) Execution Time



(b) Memory Usage

Min_sup	Metric	Apriori	FP-Growth
70.0%	Runtime (s)	3744.40	61.67
	Memory (MB)	19.44	32.27
	Frequent Items	48731	48731
75.0%	Runtime (s)	1746.00	27.30
	Memory (MB)	10.98	14.34
	Frequent Items	20993	20993
80.0%	Runtime (s)	414.97	11.40
	Memory (MB)	7.31	6.49
	Frequent Items	8227	8227
85.0%	Runtime (s)	114.71	3.89
	Memory (MB)	4.46	2.49
	Frequent Items	2669	2669
90.0%	Runtime (s)	15.74	0.97
	Memory (MB)	0.63	1.16
	Frequent Items	622	622
95.0%	Runtime (s)	0.90	0.14
	Memory (MB)	0.07	0.46

(c) Comparison Table

Figure 5: Performance Comparison of Apriori vs FP-Growth (Retail Dataset)

5 CONCLUSION

From the results retrieved from the experiment, it can be concluded that with a fixed dataset if the intention is to find frequent itemsets faster then FP-Growth is the ultimate choice. On the other hand, if one is to use less memory space, then Apriori algorithm is an efficient method. Also, if the situation in ambiguous then FP-Growth will be preferable because it has a linear-like execution time. Based on the results, although FP-Growth is preferred due to the flexibility it gives in terms of optimized performance at any given support threshold. If one is sure to depend on low memory resource then apriori algorithm can be used.

References

[1] Srikant, R., & Agrawal, R. (1997). *Comparative analysis of Apriori Algorithm and Frequent Pattern Growth Algorithm in Association Rule Mining.* `https://www.researchgate.net/publication/373776467`