COL 761 HW1 Report Question 1 Analysis

February, 2025

Team members	Entry Number
Hasit Nanda	2024VST9015
Ghosal Subhojit	2022MT61976
Arnab Goyal	2022MT61963

This analysis presents an empirical comparison of **Apriori** and **FP-tree (Frequent Pattern Growth)** algorithms performance across different support thresholds and explains the observed trends based on the algorithm execution.

1 Algorithms Execution Summary

To explain the performance differences, we provide below a brief summary of how **Apriori** and **FP-tree** execute:

1.1 Apriori Algorithm Execution

Apriori follows a **Breadth First Search** approach, starting from **single-item sets** and iteratively generating larger itemsets. This is done by generating candidate itemsets of size k, scanning the dataset to count occurences, pruning candidates which do not meet the minimum support, and then proceeding to candidates of size $k+1$ and repeating the process until no frequent itemsets are found.

Performance Bottleneck:

- **Multiple dataset scans** cause significant performance degradation since the entire dataset must be scanned each time a candidate itemset needs to be verified as frequent. In practical scenarios, databases are usually large and cannot fit entirely into the main memory. As a result, frequent disk scans are required, leading to high I/O overhead.
- **Candidate explosion**: As support decreases, the number of candidates grows exponentially as most itemsets become frequent.

1.2 FP-tree Algorithm Execution

Instead of generating candidate sets explicitly, FP-tree **compresses the dataset into a prefix tree**. This is done by scanning the dataset once to get frequent items, building

an FP-tree, extracting frequent itemsets using recursive tree traversal, and then mining conditional FP-trees for subsets of frequent items.

Performance Bottleneck:

- **Memory usage**: The FP-tree structure requires storing transaction data with linked nodes.
- **Tree depth increases at low support**, since most itemsets become frequent, making traversal expensive

2 Observed Performance Results

The following table summarizes the execution times for different support thresholds:

Support Threshold (%)	Apriori Time (s)	FP-tree Time (s)
90%	49.83	50.16
50%	49.09	48.37
25%	54.48	52.67
10%	858.96	145.19
5%	TLE	MLE

Table 1: Execution times of Apriori and FP-tree at different support thresholds.

3 Performance at High and Moderate Support Thresholds (90%, 50%, 25%)

Note: Apriori and FP-tree have similar runtimes.

- At high support thresholds, fewer frequent itemsets exist, making Apriori's candidate generation manageable.
- At high support thresholds, FP-tree remains shallow, allowing fast traversal.

4 Performance at Lower Support Thresholds (10%, 5%)

Note: FP-tree is **much faster** at 10% support but fails at 5% due to memory overflow.
At 10% support:

- Apriori's runtime increases to 858.96s due to candidate explosion.
- FP-tree is 5.9× faster (145s) because it avoids candidate generation which grows exponentially when most itemsets become frequent, and scans the dataset only twice.

At 5% support:

- **Apriori fails** due to excessive candidate generation, since at low support there is exponential growth in candidate itemsets. This leads to a Time Limit Exceeded (TLE) error.
- **FP-tree fails** because at low support, many more items are frequent and therefore more conditional FP-trees need to be created recursively. Each conditional tree can be nearly as large as the original FP-tree. This leads to excessive memory usage and recursive depth, causing memory exhaustion (MLE: Memory Limit Exceeded).

5 Conclusion

Factor	Apriori	FP-tree
Dataset Scans	Multiple (slow)	Only twice (fast)
Low Support Handling	Fails due to excessive candidates	Fails due to memory overflow
Best for	Small datasets, high support	Large datasets, moderate support

Table 2: Summary comparison of Apriori and FP-tree.

Key Takeaways:

- Apriori is efficient when frequent itemsets are small but breaks down at low support thresholds due to candidate explosion.
- FP-tree is significantly faster for lower support thresholds but consumes high memory.