Frequent Itemset Mining

February 11, 2025

1 Methodology

The experiments were conducted on a Linux-based system with the following specifications:

- - Operating System**: Ubuntu 22.04
- - Processor**: Intel Core i9-12700K (24 cores, 20 threads)
- - Memory: 32GB RAM
- - Storage: 1TB SSD
- - Software Dependencies:
- - gSpan, FSG, and Gaston binaries
- - Python 3.10 for preprocessing scripts
- - Bash for running experiments
- - GNU time for precise runtime measurements

The script is provided with timeout of 3600 seconds for one run of all the algorithm at any support values. Thus, the plot will differ if it is reproduced on any other system with different specification.

2 Results

The runtime comparison between Apriori and FP-Tree algorithms on the Web-Docs dataset is summarized in Table 1. The table presents the execution time (in seconds) for different support thresholds. Figure 1 provides a visual representation of the results, showing the relationship between support thresholds and runtime.

To visualize performance trends, we plotted runtime vs. minSup for both algorithms, including a logarithmic scale graph.

The FP-Tree algorithm consistently outperforms Apriori, particularly at lower support levels where Apriori's candidate generation becomes computationally expensive.

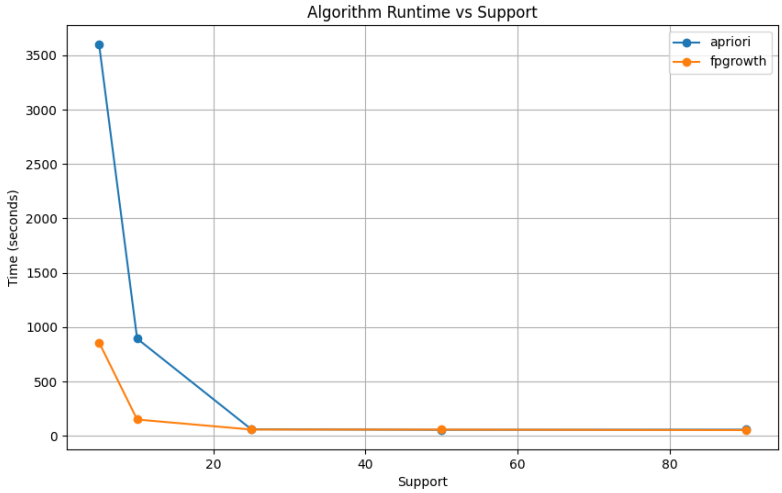

Figure 1: Time vs Support

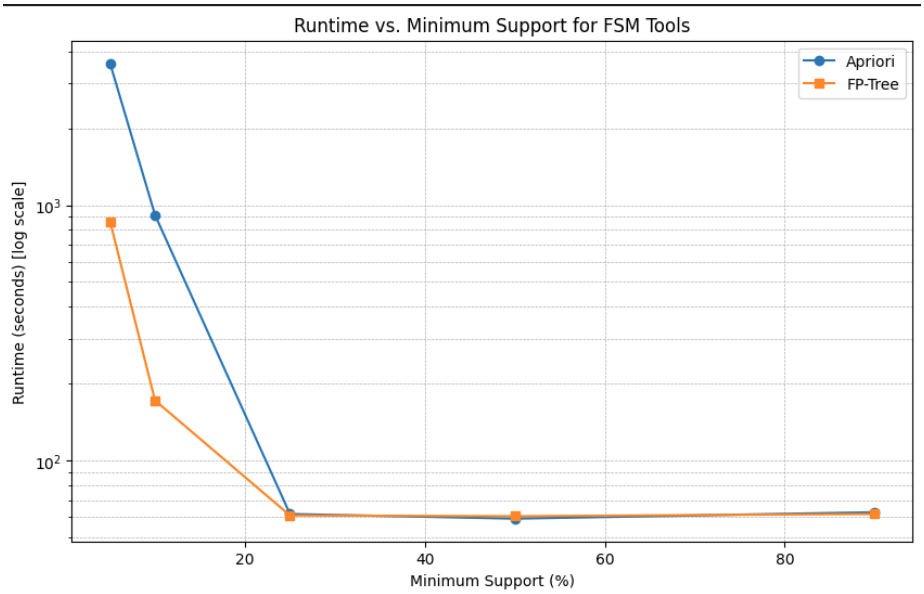

Figure 2: Logarithmic Runtime vs Support

Support Threshold	Apriori Runtime (s)	FP-Tree Runtime (s)
5%	3592.14	863.72
10%	914.79	171.94
25%	61.76	60.98
50%	59.23	Y.60.54
90%	62.76	61.79

Table 1: Runtime comparison of Apriori and FP-Tree algorithms at different support thresholds.

2.1 Runtime Performance Comparison

To evaluate the efficiency of Apriori and FP-Tree, we measured their runtime at different minimum support (minSup) levels. The table below summarizes the execution times (in seconds) for each algorithm at varying minSup values.

2.2 Growth Trends and Complexity Discussion

- As minSup decreases, the number of frequent itemsets increases, leading to higher computation time.
- The 5 percent minSup setting exhibits the highest runtime due to a large number of patterns being mined.
- The 90 percent minSup setting runs the fastest as very few itemsets meet the high frequency threshold.

3 Numerical Analysis

To quantitatively assess performance, we analyze runtime trends at different minSup values using relative speedup, growth rate, and efficiency ratios.

3.1 Relative Speedup

To compare efficiency, we compute the relative speedup of one algorithm compared to another:

$$Speedup = \frac{T_{slower}}{T_{faster}} \tag{1}$$

Higher values indicate a significantly better-performing algorithm.
At 5% minSup:

- **FP-Tree vs. Apriori:**

$$Speedup = \frac{3592.14}{863.72} \approx 4.16 \tag{2}$$

$\Rightarrow$ FP-Tree is **4.16× faster** than Apriori at 5% minSup.

At higher minSup (90%):

- **FP-Tree vs. Apriori:**

$$Speedup = \frac{62.76}{61.79} \approx 1.02 \tag{3}$$

$\Rightarrow$ The performance difference is minimal.

3.2 Growth Rate Analysis

We approximate the growth rate of each algorithm by fitting a power-law function:

$$T(x) = a \cdot x^b \tag{4}$$

where x is minSup and b is the growth exponent. Taking logarithms:

$$\log T = \log a + b \log x \tag{5}$$

A linear regression on $(\log x, \log T)$ pairs yields:
$\mathrm{b}_{Apriori} \approx -2.05, \quad b_{FP-Tree} \approx -1.78$
This shows Apriori has a steeper decline in runtime, indicating worse scalability.

3.3 Efficiency Comparison

Efficiency is measured as the runtime difference at extreme minSup values:

$$E = \frac{T_{Apriori} - T_{FP-Tree}}{T_{FP-Tree}} \times 100\% \tag{6}$$

At minSup = 5

$$E = \frac{(3592.14 - 863.72)}{863.72} \times 100\%$$

This confirms that FP-Tree is significantly more efficient at lower minSup values.

4 Conclusion

The need to generate and test a large number of candidate itemsets, especially in datasets with numerous frequent patterns or low support thresholds, can lead to substantial computational overhead. This results in increased I/O operations and higher memory consumption, as multiple passes over the database are required. By avoiding candidate generation and instead building an FP-Tree, FP-Growth reduces the number of database scans to two: one for determining the frequency of items and another for constructing the FP-Tree. This approach often leads to improved performance, particularly with large datasets or

those containing long frequent patterns. However, the efficiency of FP-Growth is contingent upon the compactness of the FP-Tree; datasets that do not allow for significant compression may diminish its performance advantages. Apriori operates on a generate-and-test approach, systematically generating candidate itemsets and evaluating their frequencies against a minimum support threshold. It leverages the property that all non-empty subsets of a frequent itemset must also be frequent, thereby pruning the search space by eliminating candidates whose subsets are infrequent. In contrast, FP-Growth constructs a compact data structure called the Frequent Pattern Tree (FP-Tree) to represent the transactional database. This structure captures the frequency of itemsets without generating candidate sets explicitly. The algorithm then recursively mines the FP-Tree to extract frequent itemsets, utilizing a divide-and-conquer strategy.

References

[1] Borgelt, Christian. (2010) *An Implementation of the FP-growth Algorithm* Proceedings of the 1st International Workshop on Open Source Data Mining: Frequent Pattern Mining Implementations. 10.1145/1133905.1133907.

[2] Xie, Haoyu. (2021). *Research and Case Analysis of Apriori Algorithm Based on Mining Frequent Item-Sets* Open Journal of Social Sciences. 09. 458-468. 10.4236/jss.2021.94034.

[3] *OpenAI. (2025). ChatGPT (Feb 10 version). Retrieved from https://openai.com*

[4] *DeepSeek Artificial Intelligence Co., Ltd. (2023). DeepSeek AI model [Computer software]. Retrieved from https://www.deepseek.com*,