COL761: Assignment 1 - Part 1

Abhay Sharma
2012CS50272

Akash Dabaniya
2021CS10111

Mridul Singh
2018CS50412

February 4, 2026

1 Task 1: Comparison of Apriori and FP-Tree Algorithms

Introduction

An empirical comparison between two prominent Frequent Itemset Mining (FIM) algorithms: **Apriori** and **FP-Growth** (Frequent Pattern Growth). The objective is to analyze their runtime performance across varying support thresholds (5%, 10%, 25%, 50%, and 90%).

Dataset

The experiments utilize the `webdocs.dat` dataset.

- **Characteristics:** The dataset is *dense* and contains very long transactions, unlike typical sparse market-basket data.

Algorithm Overview

Apriori Algorithm

The Apriori algorithm uses a *generate-and-test* approach, originally proposed by Agrawal and Srikant [1]. It relies on the **Apriori Property** (or Downward Closure Property), which states that:

All non-empty subsets of a frequent itemset must also be frequent.

How it works:

1. It works iteratively (level-wise). In the k-th pass, it generates candidate itemsets of length k (C_k) using frequent itemsets from the previous pass (L_{k-1}).
2. It scans the entire database to count the support for these candidates.
3. An itemset X is considered frequent if its support meets the threshold:

$$Support(X) = \frac{\text{count}(X)}{|D|} \geq \text{min\_sup}$$

Pros & Cons:

+ **Pros:** Simple to implement and easy to parallelize.
- **Cons:** It requires multiple scans of the database. Crucially, on dense datasets like `webdocs`, it suffers from **candidate explosion**, generating huge numbers of candidates that turn out to be infrequent, wasting significant processing time.

1.0.1 FP-Growth Algorithm

The FP-Growth algorithm adopts a *divide-and-conquer* strategy using a tree structure, avoiding candidate generation entirely, as introduced by Han et al. [2].

How it works:

1. **Compression:** It performs two scans of the database. The first scan counts item frequencies. The second scan compresses the database into a compact structure called the **FP-Tree** (Frequent Pattern Tree) in memory.

2. **Mining:** It mines the tree by recursively building "conditional FP-trees." For a specific item, it looks only at the paths in the tree ending with that item to find frequent prefixes.

Pros & Cons:

+ **Pros:** Extremely efficient because it avoids the cost of generating candidates. It only scans the database twice, regardless of pattern length.

- **Cons:** The FP-Tree is stored in main memory (RAM). For extremely large databases, the tree might become too large to fit.

Experimental Results

The algorithms were executed at support thresholds of 5%, 10%, 25%, 50%, and 90%. The runtimes were recorded and plotted (see Figure 1).

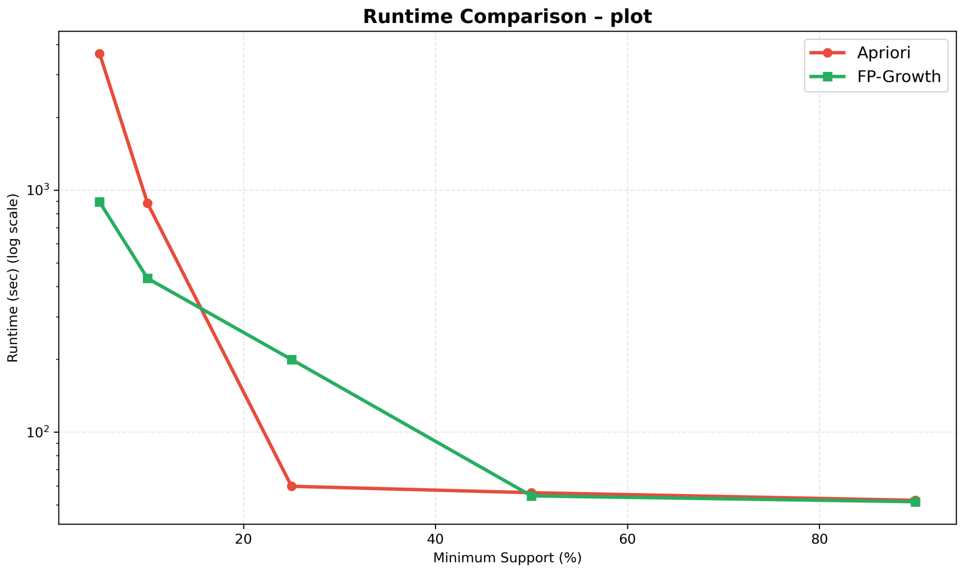

Figure 1: Runtime Comparison of Apriori vs. FP-Growth (Log Scale).

1.0.2 Analysis of Observations

- **At Low Support (5% - 10%):** We observe a massive gap in performance. **Apriori** takes significantly longer (or times out), whereas **FP-Growth** remains efficient. *Reason:* At low support, many items are considered frequent. Because `webdocs` has long transactions, the search space for Apriori explodes (checking millions of candidate combinations). FP-Growth avoids this by simply following the paths that effectively exist in the tree.

- **At High Support (50% - 90%):** The performance of both algorithms converges. *Reason:* At 90% support, very few items appear frequently enough to be counted. The computational cost becomes dominated by simple file I/O, which is similar for both.

2 Task 1.2: Dataset Creation for Runtime Replication

Objective

The goal of this task was to construct a synthetic transactional dataset that replicates the specific performance characteristics observed in Task 1.1:

1. **Low Support (5%–10%):** Apriori should be significantly slower than FP-Growth.
2. **Medium/High Support (25%–50%):** Apriori should outperform or match FP-Growth.

Dataset Construction Strategy

To achieve these runtime trends, we developed a Python script (`generation.py`) that generates approximately 15,000 transactions and 1000 items. The generation logic relied on two critical statistical properties:

- **High Transaction Density (Long Patterns):** The primary bottleneck of Apriori is the combinatorial explosion of candidate generation. If a transaction has length L, it contains $2^L - 1$ potential subsets. To force Apriori to struggle at low support, we sampled transaction lengths from a heavy-tailed distribution ranging from **60 to 450 items**. These "dense" transactions create massive candidate sets (C_2, C_3) when the support threshold is low (5%), causing Apriori's runtime to spike exponentially.
- **Zipfian Item Distribution** ($\alpha = 1.35$)**:** We distributed item frequencies using a Zipfian distribution with exponent $\alpha = 1.35$.

$$P(rank) \propto \frac{1}{rank^{1.35}}$$

This ensures that while a few items are very frequent, the "tail" of the distribution is thin.

Experimental Results

We executed both algorithms on the constructed dataset. The runtime trends are visualized below.

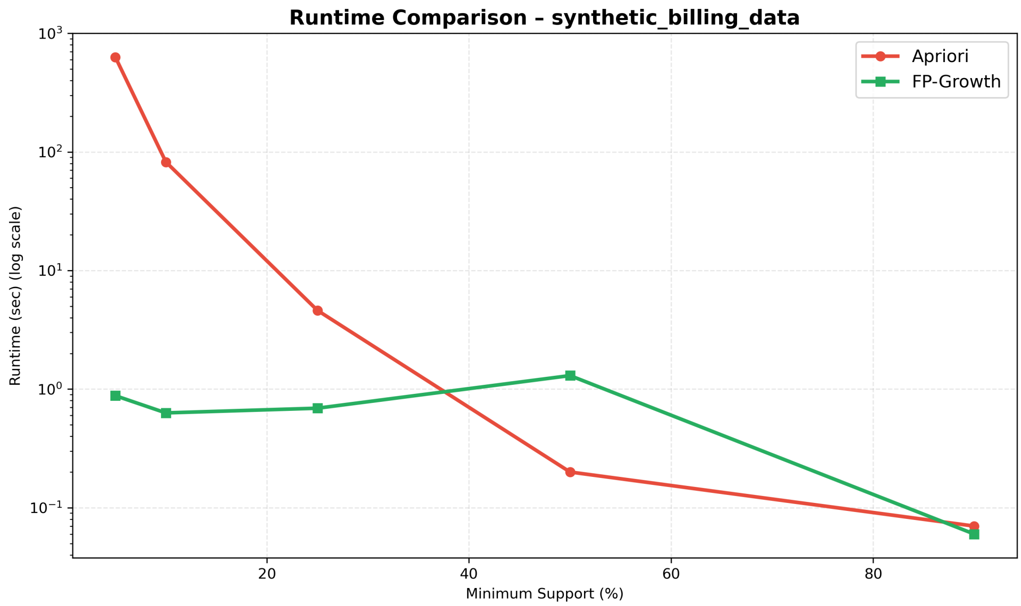

Figure 2: Runtime comparison on the Synthetic Dataset (Task 1.2).

Analysis

The resulting plot confirms that our dataset construction strategy was successful:

1. **At 5%–10% Support:** Apriori has a massive performance degradation, indicated by the steep curve/spike. This confirms that the long transaction lengths effectively overwhelmed the candidate generation step. In contrast, FP-Growth handled the dense data efficiently due to its compact tree structure.

2. **At 25%–50% Support:** A clear crossover is observed. As the support threshold eliminates the ”long tail” of infrequent items (due to the Zipfian skew), Apriori's runtime drops below that of FP-Growth.

Disclaimer

Large Language Models were used during this assignment for clarifying the logic of Frequent Itemset Mining algorithms and for assistance with LaTeX syntax. The implementation logic, data generation strategy, and experimental conclusions are our own work.

References

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," *VLDB*, 1994.

[2] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation," *SIGMOD*, 2000.