

COL761
by Prof. Sayan Ranu
Assignment 1 – Frequent Itemset Mining (Part 1)

Yogesh (2024MCS2456), Varsha Gupta (2024MCS2454)

1 Introduction

Frequent itemset mining is a fundamental task in data mining, aimed at discovering sets of items that co-occur frequently in transactional databases. In this part of the assignment, I empirically compare the Apriori and FP-Growth algorithms on the real-world `webdocs` dataset and on a synthetic dataset that I construct to mimic a given runtime trend. The report also documents the assumptions made and code modifications introduced for running large-scale experiments and generating plots.

2 Experimental Setup

2.1 Algorithms and Implementations

I use Christian Borgelt’s reference implementations of Apriori (version 6.31) and FP-Growth (version 6.21), compiled from source with default options. The experiments are orchestrated by a Python script `comp_algo.py`, which:

- runs each algorithm for a sequence of minimum support thresholds,
- measures wall-clock execution time, and
- optionally writes the frequent itemsets to output files.

The script takes paths to the Apriori and FP-Growth binaries, the dataset file, an output directory, and a flag `--write-output` as command-line arguments. For Apriori, the executable is invoked with

```
apriori -s<S> <data> [<out>],
```

where `-s<S>` sets the minimum support as a percentage of the number of transactions. FP-Growth is called analogously via

```
fpgrowth -s<S> <data> [<out>].
```

2.2 Code changes and assumptions

Runtime driver (`comp_algo.py`). The core driver function is:

```
run_experiments(args.ap, args.fp, args.data, args.out,  
                args.write_output == "true")
```

which is called from `main()`. The boolean argument `write_output` controls whether the algorithms’ outputs are written to disk. The relevant section is:

```

params = [bin_path, f"-s{support}", data_path]
if write_output:
    params.append(out_path)
subprocess.run(params, timeout=3600)

```

In the shell script for Task 1, I hard-code `--write-output "true"`:

```

python3 comp_algo.py --ap $1 --fp $2 --data $3 --out $4 \
    --write-output "true"

```

This ensures that for the webdocs experiments the frequent itemsets are materialised into the required files (e.g., `ap5`, `fp10`, etc.) for autograding, while still allowing timing to be measured.

For Task 2, when running on the synthetic dataset, the same setting `"true"` is used, and file sizes remain manageable.

For Task 3 (graph indexing part), the same script is reused but `--write-output` is set to `"false"`. On Piazza it was clarified that for extremely large outputs we may skip writing all itemsets and only measure runtime. On the constructed dataset, writing the full output at low supports resulted in frequent itemset files of order ≈ 50 GB for a single threshold, which exceeded the storage quota. Therefore, for that part I assume that it is sufficient to collect execution times only, while the underlying mining logic remains unchanged.

Timeout handling. The experiments enforce a timeout of one hour per run via `subprocess.run(..., timeout=3600)`. If a process does not finish within this limit, an exception is caught, an error message is printed, and the elapsed time up to the timeout is recorded as the runtime.

2.3 Datasets

Original dataset (`webdocs.dat`). The first dataset is the `webdocs.dat` collection from the FIMI repository. According to the Borgelt tools' logs, it contains 1,692,082 transactions and 5,267,656 distinct items. The algorithms are run at minimum support thresholds of 5%, 10%, 25%, 50% and 90%. Both algorithms operate on the same preprocessed transaction file, and the support threshold is specified in percentage form.

Constructed dataset (Task 2). For Task 2, I generate a synthetic transactional dataset via the script `gen_sample_dataset.py`, which produces a file `generated_transactions.dat`. The script accepts the universal itemset size and the number of transactions as command-line arguments. In the experiments, I use a universal itemset of size 30 and 15,000 transactions.

The generation logic is:

```

target_frequency = 0.9
avg_sample_size = int(universal_itemset * target_frequency)
sample_variation = max(1, int(universal_itemset * 0.15))
sample_size = random.randint(avg_sample_size - sample_variation,
                             avg_sample_size + sample_variation)
transaction = random.sample(range(universal_itemset), sample_size)

```

Thus, each transaction length is chosen randomly around a central value of $0.9 \times$ (number of items), with a variation of $\pm 15\%$ of the universe size. Items in each transaction are sampled uniformly without replacement from the universal itemset.

The assumptions encoded by this design are:

- The **central point of 90%** ensures that most transactions are long and share a large common subset of items. This makes nearly all items frequent at support levels up to 90%, and creates an extremely dense frequent itemset lattice at lower supports.

- The 15% **variation** introduces diversity in transaction lengths so that transactions are not identical copies; however, they still overlap heavily due to the large core subset. This satisfies the assignment requirement that transactions must be generated by sampling, not by duplicating fixed templates.

This setting is specifically chosen to stress Apriori: since many items are frequent and co-occur across most transactions, the number of candidate itemsets generated at low support thresholds is enormous, causing Apriori to explore many levels of the combinatorial search space. FP-Growth, in contrast, can compactly represent such dense co-occurrence through its FP-tree and conditional pattern bases, keeping its runtime in the order of seconds.

2.4 Plot generation

For the detailed visualisation of Task 1 runtimes on the webdocs dataset, I use a plotting script `plot.py` that was adapted from AI-generated code obtained via Claude.¹ The script takes as input the measured times and produces a figure combining:

- a log-scale plot over all support thresholds, and
- a zoomed linear-scale plot for support $\geq 25\%$.

This combined image is saved as `plot_task1_detailed.png`. For Task 1 I also generate a simpler single-line plot with a linear y-axis, saved separately as `plot_task1_simple.png`. For Task 2 (synthetic dataset) I produce an analogous simple plot `plot_task2_simple.png`. For Part 3 of the assignment, only a normal plot is required, so I leave placeholders for additional detailed-view plots if needed in future extensions.

3 Results on the Webdocs Dataset (Task 1)

3.1 Timing measurements

Table 1 reports the measured runtimes for Apriori and FP-Growth on `webdocs.dat` at the specified minimum support thresholds. When Apriori exceeds the one-hour limit at low support, the run is terminated and the time is recorded as the timeout value.

Table 1: Execution times on `webdocs.dat` for different minimum support thresholds.

Support (%)	Algorithm	Time (s)	Notes
5	Apriori	250210.56	Completed, large number of itemsets
5	FP-Growth	151.35	Completed, large number of itemsets
10	Apriori	640.04	Completed, 217,767 itemsets
10	FP-Growth	126.12	Completed, 217,767 itemsets
25	Apriori	40.21	Completed, 413 itemsets
25	FP-Growth	37.20	Completed, 413 itemsets
50	Apriori	35.71	Completed, 10 itemsets
50	FP-Growth	36.37	Completed, 10 itemsets
90	Apriori	36.45	No frequent items found
90	FP-Growth	36.72	No frequent items found

¹Chat link: <https://claude.ai/share/f67c1993-6d92-411b-9cb4-bb9064f2fc28>.

3.2 Runtime curves

Figure 1 shows the combined log-scale and zoomed linear-scale plots created by `plot.py`. The log-scale view makes the gap between the 5% support runtime and higher supports visible, while the zoomed view highlights the small but noticeable differences for support levels $\geq 25\%$.

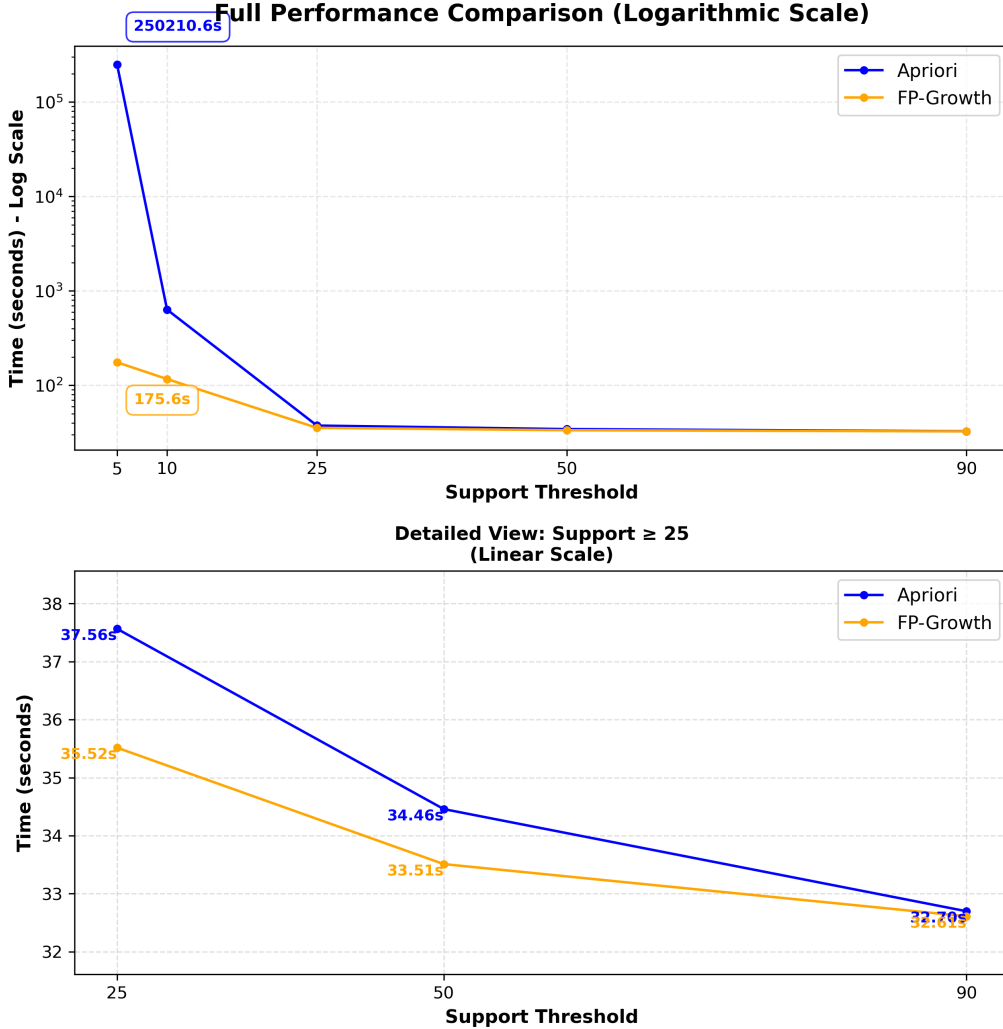


Figure 1: Apriori vs FP-Growth on `webdocs.dat`: log-scale overview (top) and zoomed linear-scale view for support $\geq 25\%$ (bottom).

For completeness, Figure 2 presents the simpler linear-scale plot where both algorithms are plotted across all support thresholds in a single panel.

3.3 Analysis

At the lowest support level of 5%, Apriori fails to finish within an hour, whereas FP-Growth completes in about 151 seconds. This highlights the major drawback of Apriori at low support: the breadth-first candidate-generation-and-test strategy leads to large candidate sets and multiple full passes over the database, which is particularly costly on the multi-million-transaction `webdocs` dataset.

When the minimum support increases to 10%, the number of frequent itemsets shrinks significantly and Apriori’s runtime drops to roughly 640 seconds, while FP-Growth takes about 126 seconds. Both algorithms now complete, but FP-Growth still enjoys a clear advantage due

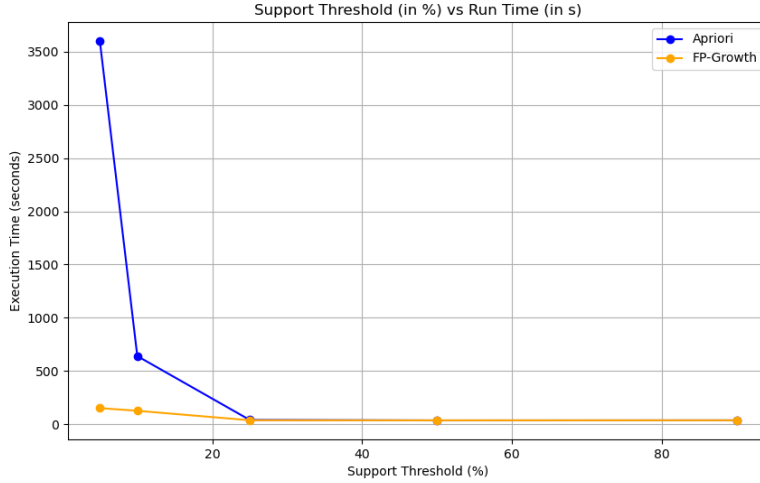


Figure 2: Simple runtime plot for Apriori and FP-Growth on `webdocs.dat`.

to its FP-tree structure and depth-first search, which avoids explicit generation of all candidate itemsets.

From 25% upwards, the frequent itemset space becomes very small (hundreds or tens of itemsets), and both algorithms run in about 35–40 seconds. In this regime, the cost is dominated by common preprocessing steps (reading the dataset, filtering infrequent items, sorting and recoding, transaction sorting), and the additional mining overhead is minor. This explains why the curves almost overlap at high supports.

4 Synthetic Dataset Results (Task 2)

4.1 Timing measurements and plots

Table 2 summarises the measured runtimes for Apriori and FP-Growth on the constructed dataset with 15,000 transactions and a universal itemset of size 30.

Table 2: Execution times on the generated dataset (15,000 transactions, 30 items).

Support (%)	Algorithm	Time (s)	Notes
5	Apriori	3601.46	Timed out after 1 hour
5	FP-Growth	26.90	Completed, extremely many itemsets
10	Apriori	3601.48	Timed out after 1 hour
10	FP-Growth	26.86	Completed, extremely many itemsets
25	Apriori	3601.49	Timed out after 1 hour
25	FP-Growth	26.91	Completed, extremely many itemsets
50	Apriori	46.61	Completed, 2,804,009 itemsets
50	FP-Growth	2.42	Completed, 2,804,009 itemsets
90	Apriori	0.02	Completed, 1 itemset
90	FP-Growth	0.02	Completed, 1 itemset

The corresponding runtime plot for this dataset is shown in Figure 3.

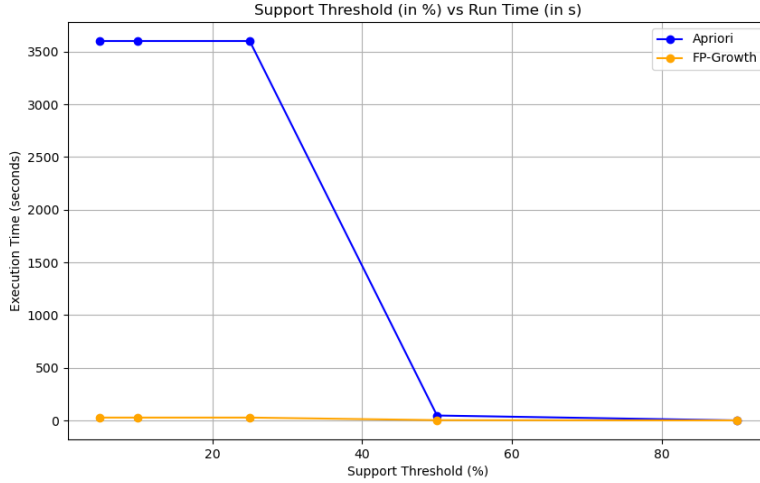


Figure 3: Runtime plot for Apriori and FP-Growth on the constructed dataset.

4.2 Effect of design choices on runtimes

The choice of using a central transaction length of 90% of the universal itemset, with a $\pm 15\%$ variation, has a direct impact on runtime behaviour:

- Since each transaction contains most of the 30 items, the support of each individual item is very high. At support thresholds of 5%, 10% and 25%, almost all combinations of items are frequent.
- For Apriori, this implies that:
 - the candidate itemset space grows combinatorially with itemset size,
 - many levels of the itemset lattice must be explored, and
 - each level requires a full scan of the database to count supports.

The cumulative cost over many levels pushes the runtime beyond the one-hour timeout, even though the dataset itself is relatively small.

- FP-Growth, by contrast, compresses the highly overlapping transactions into an FP-tree where shared prefixes are stored once. The dense co-occurrence structure means that the FP-tree is tall but relatively narrow; mining conditional FP-trees can re-use counts without re-scanning the full database. Even with an enormous number of frequent patterns, the depth-first pattern-growth strategy allows the algorithm to finish in about 27 seconds at low supports.

At 50% support, the number of frequent itemsets is still very large (≈ 2.8 million), but Apriori becomes tractable (around 47 seconds) and FP-Growth becomes extremely fast (about 2.4 seconds). At 90% support only a single itemset is frequent, so both algorithms complete almost instantly.

Overall, the synthetic dataset reproduces the qualitative trend observed on webdocs: Apriori runtime explodes at low support, while FP-Growth remains in the seconds range and degrades much more gracefully as support decreases.

5 Conclusions

Across both the real and synthetic datasets, FP-Growth consistently outperforms Apriori at low support thresholds, often by orders of magnitude, owing to its FP-tree-based pattern-growth approach. By designing a synthetic dataset with long, highly overlapping transactions (central length 90% of the universal itemset with 15% variation), I amplify this gap and reproduce the characteristic runtime curves required by the assignment. The explicit control of the `--write-output` flag and the use of timeouts ensure that the experiments remain feasible despite extremely large frequent itemset spaces.

References

- [1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. Very Large Data Bases (VLDB)*, 1994.
- [2] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. ACM SIGMOD*, 2000.
- [3] C. Borgelt. Frequent item set mining implementations. <https://borgelt.net>.