# University of Dhaka

Department of Computer Science and Engineering

CSE:4255 Introduction to Data Mining and Warehousing Lab

Lab Report 1: A Comparative Analysis of Apriori and FP-Growth Algorithms

**Submitted By:**

Name : Ahaj Mahhin Faiak

Roll No : 01

**Submitted On :**

April 29, 2025

**Submitted To :**

Dr.Chowdhury Farhan Ahmed, Professor

Md. Mahmudur Rahman, Assistant Professor

## 1 Introduction

This report compares two popular algorithms for frequent itemset mining: **Apriori** and **FP-Growth**. Frequent itemset mining is a fundamental task in data mining, aimed at discovering patterns in transactional datasets. This comparison highlights the differences between the two algorithms in terms of computation time and memory usage across varying support thresholds, using both sparse (Retail, T10I4D100K, Kosarak) and dense (Mushroom, Chess, Connect-4) datasets.

### 1.1 Objectives

The main objectives of this report are:

- To gain a comprehensive understanding of the operational principles of the Apriori and FP-Growth algorithms.
- To accurately implement both the Apriori and FP-Growth algorithms.
- To evaluate and compare the performance of these algorithms in terms of execution time and memory consumption.
- To assess their performance on both sparse and dense datasets.

## 2 Theory

### 2.1 Apriori Algorithm

The Apriori algorithm is a well-known data mining technique used for discovering frequent itemsets and generating association rules in transactional datasets. It works by first identifying individual items that appear frequently, then progressively generating larger itemsets by combining these frequent items, while pruning those that do not meet the minimum support threshold. This process is repeated iteratively, with the algorithm continuing to find frequent itemsets of increasing size until no further frequent itemsets can be found. Once frequent itemsets are discovered, the algorithm can generate association rules, indicating relationships between items, such as if a customer buys bread, they are likely to buy butter. While the Apriori algorithm is simple and effective for small to moderate-sized datasets, it can become computationally expensive and memory-intensive for large datasets due to the need to evaluate many candidate itemsets.

### 2.2 FP-Growth Algorithm

The FP-Growth (Frequent Pattern Growth) algorithm is an efficient method for mining frequent itemsets in large datasets, designed to overcome the limitations of the Apriori algorithm. Unlike Apriori, which generates candidate itemsets at each level, FP-Growth uses a compact data structure called the FP-tree to represent the dataset. The algorithm works by first constructing the FP-tree, which compresses the dataset by grouping similar transactions together, and then recursively mining the tree to discover frequent itemsets. This eliminates the need for candidate generation, significantly improving efficiency and reducing memory usage. FP-Growth is particularly suited for large datasets as it avoids the costly process of scanning the entire dataset multiple times, making it faster and more scalable than Apriori in practice.

## 3 Implementation Details

### 3.1 Apriori Algorithm Implementation Details

The Apriori algorithm is a classical approach used for frequent itemset mining. The following steps describe the implementation:

- **Reading the Dataset:** The dataset is read from a file, where each transaction is represented by space-separated items. The file is parsed into a list of transactions, where each transaction is a list of integers representing the items in that transaction.

- **Finding Frequent 1-itemsets:** The first step in the algorithm is to find frequent 1-itemsets. This is done by counting the occurrence of each item in all transactions. Items whose count exceeds the minimum support threshold are considered frequent 1-itemsets.

- **Generating Candidate Itemsets:** After finding frequent itemsets of size 1, the algorithm proceeds to generate candidate itemsets of size $k$. This is done by joining frequent itemsets of size $k-1$ and ensuring that no infrequent subsets are present in the candidate itemsets.

- **Pruning Infrequent Itemsets:** The algorithm checks if any candidate itemset contains an infrequent subset. If it does, it is discarded, as it is unlikely to be frequent. This pruning step helps in reducing the number of candidate itemsets that need to be evaluated.

- **Counting the Support:** The support of a candidate itemset is calculated by checking how many transactions contain the candidate itemset. If the support is greater than or equal to the minimum support threshold, the itemset is considered frequent.
- **Iterative Process:** The algorithm starts with finding frequent 1-itemsets and iteratively generates candidate itemsets for higher values of $k$. The process continues until no more frequent itemsets are found. For each iteration, the candidates are generated, their support is counted, and infrequent itemsets are pruned.
- **Output:** The final output of the Apriori algorithm is the set of all frequent itemsets that meet the minimum support threshold. The total number of frequent itemsets found during the execution is also recorded.

The key functions involved in this implementation include:

- `find_frequent_1_itemsets`: Identifies the frequent 1-itemsets by counting item occurrences and comparing them with the minimum support threshold.
- `apriori_gen`: Generates candidate itemsets of size $k$ by merging frequent itemsets of size $k - 1$.
- `has_infrequent_subset`: Prunes candidate itemsets by checking for the presence of infrequent subsets.
- `count_support`: Computes the support of a candidate itemset by counting how many transactions contain it.
- `APRIORI_ALGO`: Orchestrates the entire process, iterating over the itemsets, generating candidates, counting support, and pruning infrequent itemsets.

This implementation ensures efficient mining of frequent itemsets by applying candidate pruning and optimizing the number of itemsets generated and evaluated.

## 3.2 FP-Growth Algorithm Implementation Details

The FP-Growth algorithm is a frequent pattern mining algorithm that efficiently finds frequent itemsets without generating candidate itemsets. It works by building a compressed tree structure called the FP-tree and mining frequent patterns from it.

- **Dataset Reading:** The dataset is read from a file where each line represents a transaction. Each transaction consists of space-separated items. The dataset is parsed into a list of transactions, where each transaction is a list of integers representing the items.
- **Building the FP-Tree:** The FP-tree is built using the `BUILD_TREE` function. This function:
  - Scans the transactions to count the frequency of each item.
  - Filters out items whose frequency is below the minimum support threshold.
  - Sorts the remaining items based on their frequency in descending order.
  - Constructs the FP-tree by inserting each transaction, where each node in the tree represents an item, and each path represents a transaction.

  The tree's root node is initialized with a count of 1. Nodes in the tree are linked together to form a header table that points to each item in the FP-tree, facilitating efficient traversal.
- **Inserting Items into the Tree:** The function `insert_tree` recursively inserts items from each transaction into the FP-tree. If an item already exists as a child of the current node, its count is incremented. If it does not exist, a new node is created, and it is linked to the header table if necessary.
- **Mining Frequent Itemsets:** The `MINE_TREE` function mines the FP-tree to extract frequent itemsets. It starts with an empty prefix and iteratively:
  - Sorts the items in the header table by their frequency.
  - For each item, creates a new prefix by adding the item to the current prefix.
  - Appends the frequent itemset along with its frequency to the list.
  - Constructs a conditional pattern base (a set of paths that lead to the item) and uses it to build a conditional FP-tree.

  - Recursively mines frequent itemsets from the conditional FP-tree.

- **Final Output:** The final output is a list of frequent itemsets, each represented as a set of items with its corresponding frequency. These itemsets meet the minimum support threshold and are displayed at the end of the algorithm.

**Key Functions:**

- `read_dataset`: Reads the dataset from the file and parses it into a list of transactions.
- `BUILD_TREE`: Constructs the FP-tree by filtering items based on support and inserting them into the tree.
- `insert_tree`: Recursively inserts items into the FP-tree.
- `MINE_TREE`: Mines the FP-tree to generate frequent itemsets.
- `FP_TREE_ALGO`: Orchestrates the entire FP-Growth process, including building the tree and mining frequent itemsets.

This approach is more efficient than the Apriori algorithm because it avoids candidate generation and pruning by using a compact tree structure and direct pattern mining.

## 4 Experimental Result

### 4.1 Sparse Dataset

#### 4.1.1 Retail

From the graphs,we see that for small minimum support values ,the difference in memory-usage and time-consumption is evident. But for larger minimum support, the impact is negligible as it is a sparse graph.

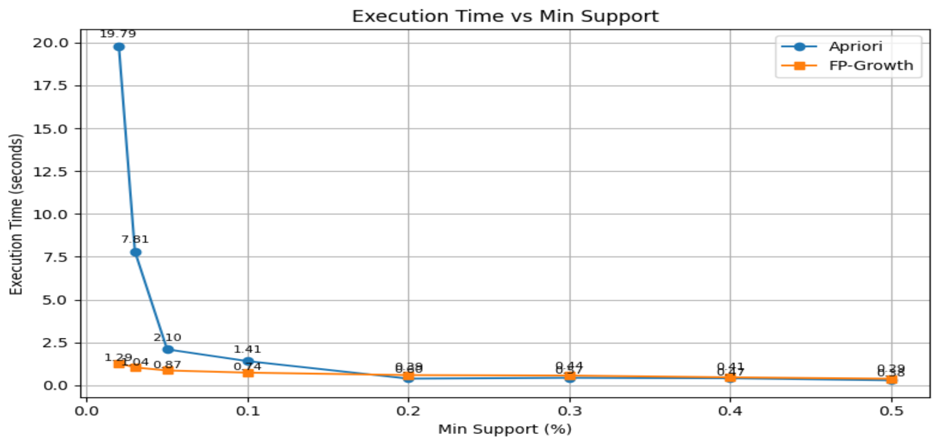


Figure 1: Retail Execution Time

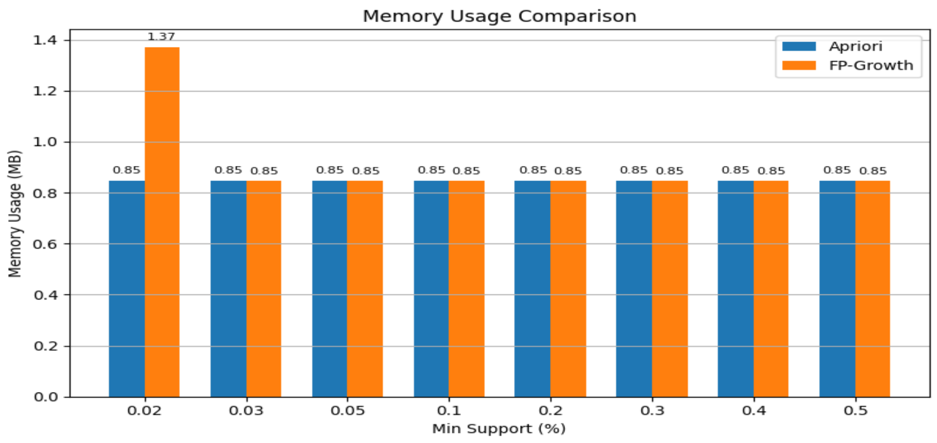
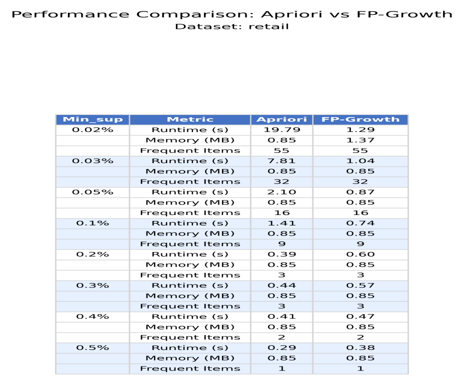


Figure 2: Retail Memory Usage


Performance Comparison: Apriori vs FP-Growth
Dataset: retail

| Min_sup | Metric | Apriori | FP-Growth |
|---|---|---|---|
| 0.02% | Runtime (s) | 19.79 | 1.29 |
| | Memory (MB) | 0.85 | 1.37 |
| | Frequent Items | 55 | 55 |
| 0.03% | Runtime (s) | 7.81 | 1.04 |
| | Memory (MB) | 0.85 | 0.85 |
| | Frequent Items | 32 | 32 |
| 0.05% | Runtime (s) | 2.10 | 0.87 |
| | Memory (MB) | 0.85 | 0.85 |
| | Frequent Items | 16 | 16 |
| 0.1% | Runtime (s) | 1.41 | 0.74 |
| | Memory (MB) | 0.85 | 0.85 |
| | Frequent Items | 9 | 9 |
| 0.2% | Runtime (s) | 0.39 | 0.60 |
| | Memory (MB) | 0.85 | 0.85 |
| | Frequent Items | 3 | 3 |
| 0.3% | Runtime (s) | 0.44 | 0.57 |
| | Memory (MB) | 0.85 | 0.85 |
| | Frequent Items | 3 | 3 |
| 0.4% | Runtime (s) | 0.41 | 0.47 |
| | Memory (MB) | 0.85 | 0.85 |
| | Frequent Items | 2 | 2 |
| 0.5% | Runtime (s) | 0.29 | 0.38 |
| | Memory (MB) | 0.85 | 0.85 |
| | Frequent Items | 1 | 1 |

Figure 3: Retail Output

### 4.1.2 Kosarak

Here FP-Growth algorithm needed more memory than apriori for smaller min-sup, because of recursive tree consumption. But for larger values of min-sup, the difference is mitigated.FP-Growth is still faster though.

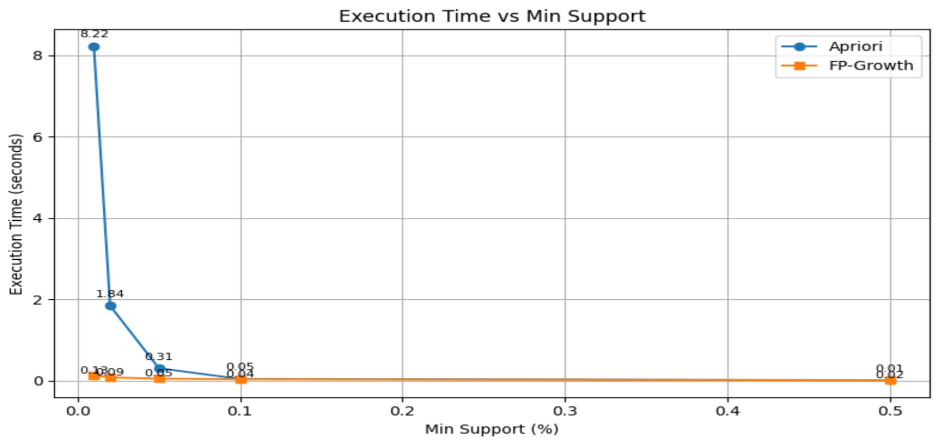


Figure 4: Kosarak Execution Time

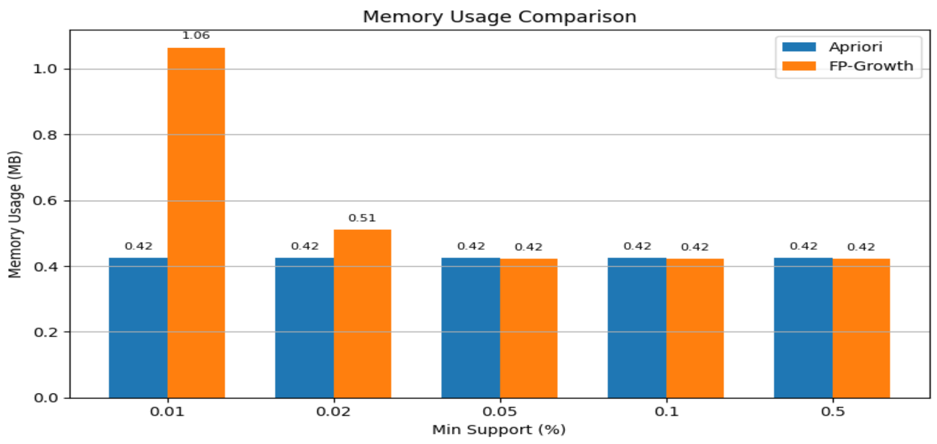
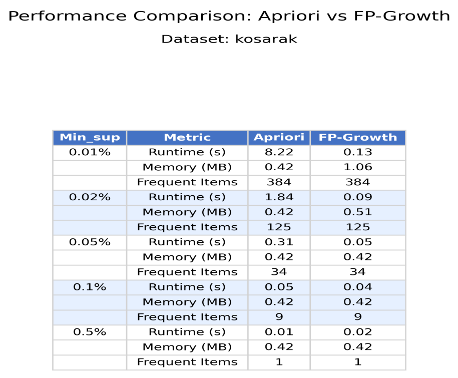


Figure 5: Kosarak Memory Usage


Performance Comparison: Apriori vs FP-Growth

Dataset: kosarak

| Min_sup | Metric | Apriori | FP-Growth |
|---|---|---|---|
| 0.01% | Runtime (s) | 8.22 | 0.13 |
| | Memory (MB) | 0.42 | 1.06 |
| | Frequent Items | 384 | 384 |
| 0.02% | Runtime (s) | 1.84 | 0.09 |
| | Memory (MB) | 0.42 | 0.51 |
| | Frequent Items | 125 | 125 |
| 0.05% | Runtime (s) | 0.31 | 0.05 |
| | Memory (MB) | 0.42 | 0.42 |
| | Frequent Items | 34 | 34 |
| 0.1% | Runtime (s) | 0.05 | 0.04 |
| | Memory (MB) | 0.42 | 0.42 |
| | Frequent Items | 9 | 9 |
| 0.5% | Runtime (s) | 0.01 | 0.02 |
| | Memory (MB) | 0.42 | 0.42 |
| | Frequent Items | 1 | 1 |

Figure 6: Kosarak Output

### 4.1.3 T10

T10 dataset showed clear differnce in memory usage and time-consumption. Clearly, FP-Growth requires more memory and Apriori takes more time.

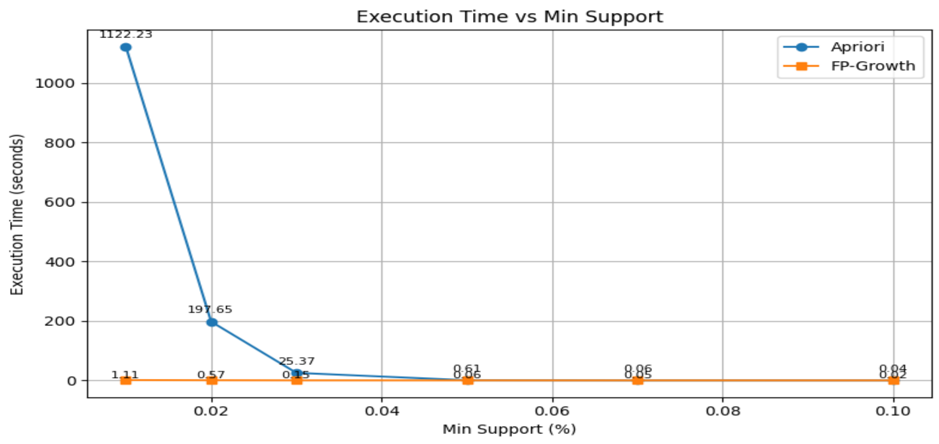


Figure 7: T10 Execution Time

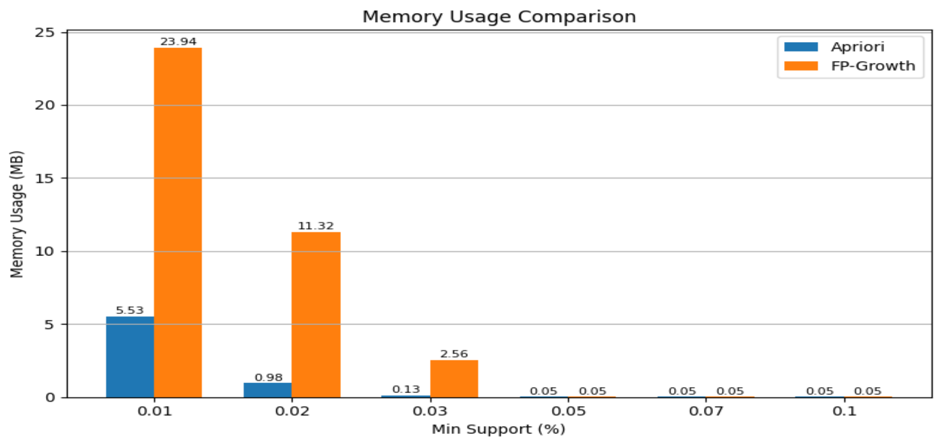
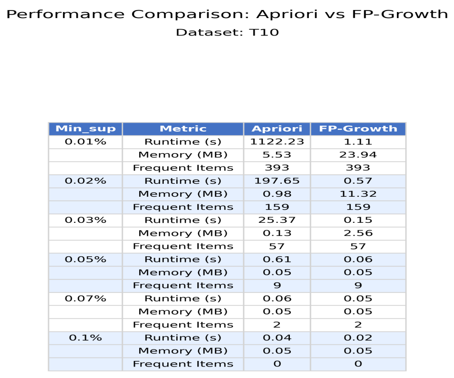


Figure 8: T10 Memory Usage



Performance Comparison: Apriori vs FP-Growth

Dataset: T10

| Min_sup | Metric | Apriori | FP-Growth |
|---|---|---|---|
| 0.01% | Runtime (s) | 1122.23 | 1.11 |
| | Memory (MB) | 5.53 | 23.94 |
| | Frequent Items | 393 | 393 |
| 0.02% | Runtime (s) | 197.65 | 0.57 |
| | Memory (MB) | 0.98 | 11.32 |
| | Frequent Items | 159 | 159 |
| 0.03% | Runtime (s) | 25.37 | 0.15 |
| | Memory (MB) | 0.13 | 2.56 |
| | Frequent Items | 57 | 57 |
| 0.05% | Runtime (s) | 0.61 | 0.06 |
| | Memory (MB) | 0.05 | 0.05 |
| | Frequent Items | 9 | 9 |
| 0.07% | Runtime (s) | 0.06 | 0.05 |
| | Memory (MB) | 0.05 | 0.05 |
| | Frequent Items | 2 | 2 |
| 0.1% | Runtime (s) | 0.04 | 0.02 |
| | Memory (MB) | 0.05 | 0.05 |
| | Frequent Items | 0 | 0 |

Figure 9: T10 Output

## 4.2 Dense Dataset

### 4.2.1 Mushroom

Dense datasets requires more time for searching patterns, but FP-Growth still shows impressive performance for execution time. But it is less impressive for its memory consumption. Also the minimum support values taken are larger compared to sparse datasets.

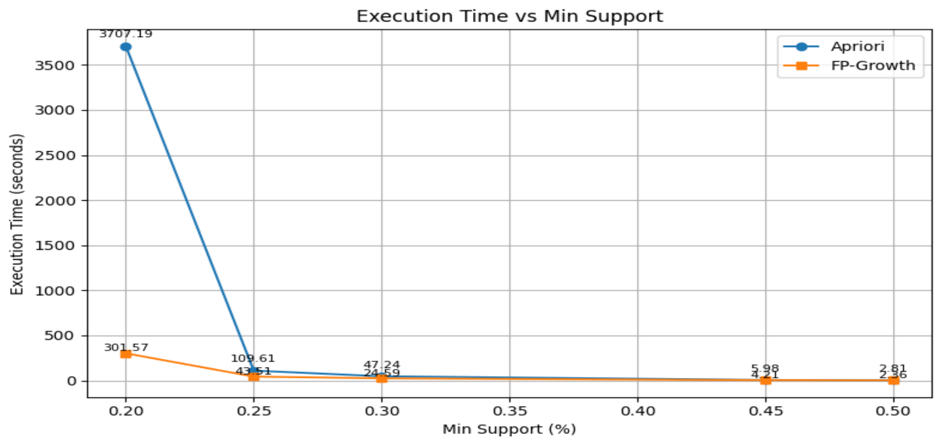


Figure 10: Mushroom Execution Time

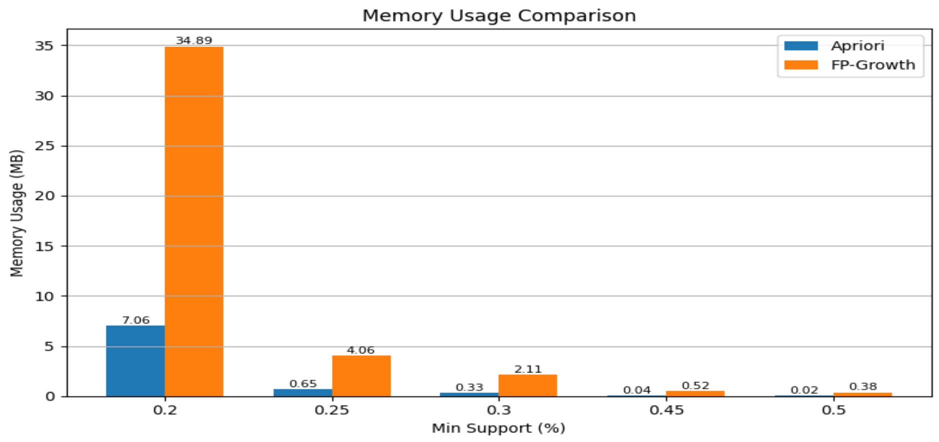
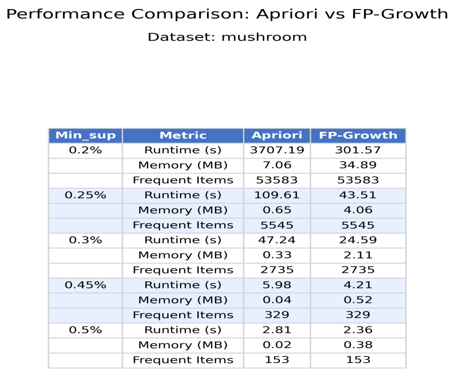


Figure 11: Mushroom Memory Usage



Performance Comparison: Apriori vs FP-Growth

Dataset: mushroom

| Min_sup | Metric | Apriori | FP-Growth |
|---|---|---|---|
| 0.2% | Runtime (s) | 3707.19 | 301.57 |
|  | Memory (MB) | 7.06 | 34.89 |
|  | Frequent Items | 53583 | 53583 |
| 0.25% | Runtime (s) | 109.61 | 43.51 |
|  | Memory (MB) | 0.65 | 4.06 |
|  | Frequent Items | 5545 | 5545 |
| 0.3% | Runtime (s) | 47.24 | 24.59 |
|  | Memory (MB) | 0.33 | 2.11 |
|  | Frequent Items | 2735 | 2735 |
| 0.45% | Runtime (s) | 5.98 | 4.21 |
|  | Memory (MB) | 0.04 | 0.52 |
|  | Frequent Items | 329 | 329 |
| 0.5% | Runtime (s) | 2.81 | 2.36 |
|  | Memory (MB) | 0.02 | 0.38 |
|  | Frequent Items | 153 | 153 |

Figure 12: Mushroom Output

### 4.2.2 Chess

Chess took the most time out of all others.Here FP-Growth showed an almost linear decrease in execution time with increasing min-sup, while Apriori algorithm showed an exponential decrease.

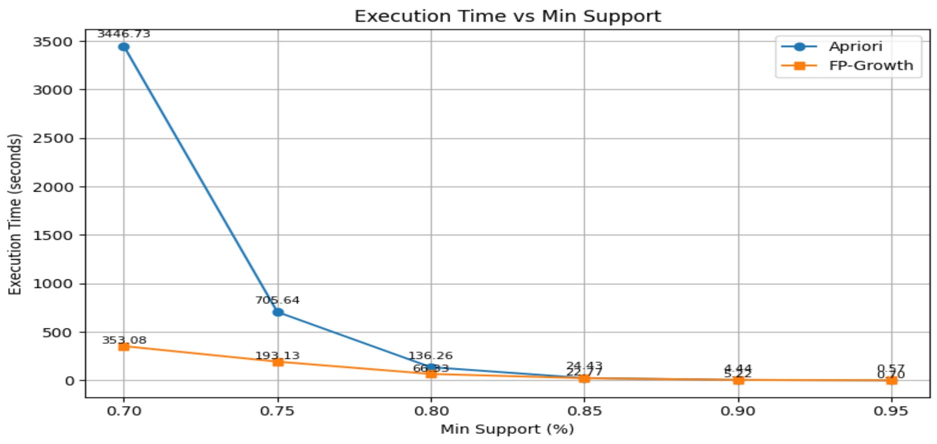


Figure 13: Chess Execution Time

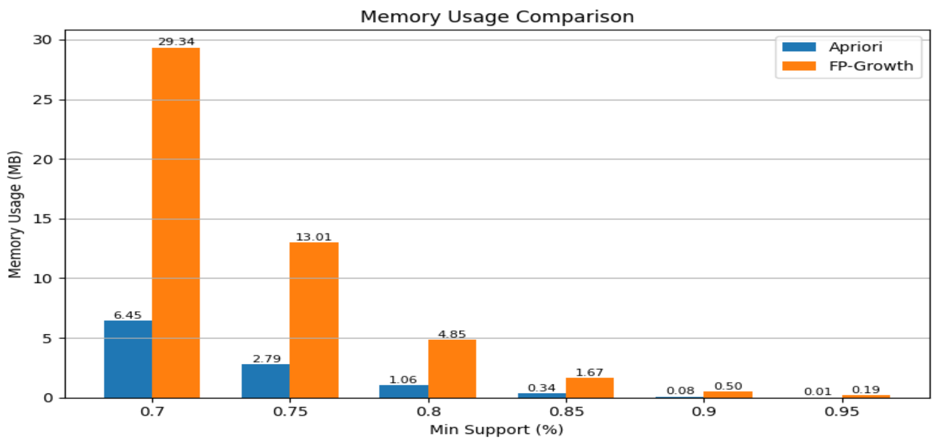
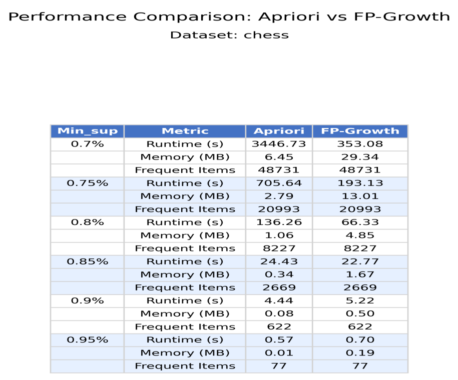


Figure 14: Chess Memory Usage



Performance Comparison: Apriori vs FP-Growth
Dataset: chess

| Min_sup | Metric | Apriori | FP-Growth |
|---|---|---|---|
| 0.7% | Runtime (s) | 3446.73 | 353.08 |
| | Memory (MB) | 6.45 | 29.34 |
| | Frequent Items | 48731 | 48731 |
| 0.75% | Runtime (s) | 705.64 | 193.13 |
| | Memory (MB) | 2.79 | 13.01 |
| | Frequent Items | 20993 | 20993 |
| 0.8% | Runtime (s) | 136.26 | 66.33 |
| | Memory (MB) | 1.06 | 4.85 |
| | Frequent Items | 8227 | 8227 |
| 0.85% | Runtime (s) | 24.43 | 22.77 |
| | Memory (MB) | 0.34 | 1.67 |
| | Frequent Items | 2669 | 2669 |
| 0.9% | Runtime (s) | 4.44 | 5.22 |
| | Memory (MB) | 0.08 | 0.50 |
| | Frequent Items | 622 | 622 |
| 0.95% | Runtime (s) | 0.57 | 0.70 |
| | Memory (MB) | 0.01 | 0.19 |
| | Frequent Items | 77 | 77 |

Figure 15: Chess Output

## 5 Conclusion

The experiment successfully demonstrated the expected outcomes. It was observed that the FP-Growth algorithm generally performed faster than Apriori, although it required more memory. Additionally, the execution time showed a consistent decrease as the minimum support threshold increased.