Optimization of the Apriori Algorithm

XinYu Chen, Yu Gao

***Abstract*—With the development of data mining technology, the Apriori algorithm, as a classical association rule mining algorithm, has been widely used in data analysis and business applications. However, the Apriori algorithm faces issues of high computational complexity, multiple database scans, and significant memory consumption when dealing with large-scale datasets. This paper proposes an improvement to the Apriori algorithm to address these issues. The improved algorithm first designs a new data structure to reduce database scan times and speed up calculations. It then proposes two counting methods and dynamically determines which to use during runtime. Finally, it optimizes the generation of candidate itemsets through sorting. Experimental results show that the improved algorithm demonstrates significant performance enhancement in handling large-scale datasets, particularly in calculation time.**

I. Introduction

ASSOCIATION rule mining is a key task in data mining, primarily used to discover relationships between itemsets in datasets. The Apriori algorithm, a classic algorithm for association rule mining, has been widely applied and studied since its proposal. The fundamental idea of the Apriori algorithm is based on a "bottom-up" generation process of frequent itemsets. First, the algorithm scans the database to find all frequent 1-itemsets. Then, it generates frequent 2-itemsets based on these, and so on, until no new frequent itemsets can be generated. The Apriori algorithm leverages the important property that "subsets of frequent itemsets are also frequent itemsets" to prune candidate itemsets during the generation process, reducing unnecessary computations. Despite its theoretical efficiency, the Apriori algorithm has several shortcomings in practical applications. First, the algorithm requires multiple scans of the database, which is inefficient when handling large-scale data. Second, the generation and verification process of candidate itemsets is complex, especially in dense or high-dimensional datasets. Additionally, the Apriori algorithm consumes significant memory resources, making it difficult to handle large-scale datasets.

II. Method

A. Steps of the Apriori Algorithm

1) Generate Frequent 1-itemsets:
   - Scan the entire transaction database and count the occurrences of each item.
   - Itemsets with support greater than or equal to the minimum support threshold form the frequent 1-itemsets (L1).
2) Generate Candidate Itemsets:
   - Generate candidate k-itemsets (Ck) from frequent (k-1)-itemsets. The generation follows the "join step" and "prune step":
     - Join Step: Combine pairs of frequent (k-1)-itemsets having the first (k-2) items in common to form candidate k-itemsets.
     - Prune Step: Remove candidate itemsets that have any (k-1)-subset not present in the frequent (k-1)-itemsets.
3) Filter Frequent Itemsets:
   - Scan the transaction database to calculate the support of each candidate itemset.
   - Candidate itemsets with support greater than or equal to the minimum support threshold form the frequent k-itemsets (Lk).
4) Repeat Steps 2 and 3 until no new frequent itemsets can be generated.

B. Binary Number Storage Method

In the above Apriori algorithm steps, data storage involves the transaction database, frequent itemsets, and candidate itemsets; data operations include counting candidate item occurrences (support calculation), item merging, and generating subsets. The following sections convert these data storage and calculation parts into binary storage and computation.

The basic idea of the binary number storage method is that each transaction and itemset can be represented by a binary number, where each bit represents a specific item. If a bit is 1, the item is present in the transaction or itemset; if it is 0, it is absent. For example, with items A, B, C, and D, they can be assigned binary bits as follows: A: 0001, B: 0010, C: 0100, D: 1000. A transaction 'A, C' can be represented as 0101, and an itemset 'A, C' can also be represented as 0101.

Steps to Use

1) Item Mapping: Create a dictionary that maps each item to a unique binary bit.
2) Compressed Representation of Transactions: Traverse each transaction and convert it to a binary number.
3) Generate Candidate Itemsets:
   - Generate candidate itemsets from frequent items using the first k-2 common items (e.g., 100110 and 010110, where all 1s except for the highest bit are in the same positions).
   - Generate candidate itemsets using bitwise OR (e.g., 100110 and 010110 generate 110110).
   - Prune: Check if any k-1 subset of the candidate itemset is present in the frequent k-1 itemsets. For example, for the itemset 110110, check if 010110, 100110, 110010, and 110100 are in the frequent 3-itemsets.
4) Support Counting: Traverse the transactions and perform a bitwise AND operation between each transaction and

each candidate itemset. If the result is not zero, the transaction contains the candidate itemset.

Finally, filter the frequent itemsets based on the counting results and continue generating candidate itemsets until the frequent itemsets are empty.

C. Support Counting Optimization: Traverse Candidate Itemsets or Transaction Subsets

When calculating support in the Apriori algorithm, if the number of candidate itemsets is very large, traversing each candidate itemset to compare its presence in the transactions is very time-consuming. In this case, generating subsets of transactions can be more efficient. Traversing the subsets can be less frequent than traversing candidate itemsets, and checking if each subset exists in the candidate itemsets can also be counted. However, the number of subsets generated from each transaction and the length and number of candidate itemsets change with each iteration. For example, the number of subsets generated from transactions in the initial iterations is relatively small, while the number of candidate itemsets increases and then decreases. Therefore, when traversing each transaction, the traversal scheme can be chosen by comparing the number of transaction subsets (calculating the combination number C_n^k) and the size of the candidate itemset.

D. Candidate Generation Optimization: Sort and Generate Candidate Itemsets from Adjacent Frequent Itemsets

When generating candidate k-itemsets from frequent (k-1) itemsets, it is necessary to compare frequent (k-1) itemsets pairwise, determine if their intersection forms a k-itemset, and check if all other (k-1) subsets of the intersection are also in the frequent (k-1) itemsets. These two steps are very time-consuming.

We note the following rule: a necessary condition for a k-itemset $t = \{x1, x2, x3, ..., xk\}$ to be a candidate k-itemset is that its (k-1) subsets $t1 = \{x1, x3, ..., xk\}$ and $t2 = \{x2, x3, ..., xk\}$ are in the frequent (k-1) itemsets. Mapped to binary storage, $t1$ and $t2$ have the same suffix, which is the binary representation of {x3, x4, ..., xk}. We only need to specially sort the frequent (k-1) itemsets such that items with the same suffix are adjacent. Then, when combining, only adjacent items need to be combined to generate the candidate itemsets, rather than pairwise combinations.

Since the above is a necessary condition, the candidate itemset is a subset of the obtained set. This set still contains many non-candidate items. Although this increases the counting step's overhead, we choose to ignore these non-candidate items because the overhead of checking each item is greater than not checking. By shifting some calculations to the counting step, the overall running speed is increased.

III. Experiments

According to the points mentioned in the Method section, this experiment compares the effectiveness of four methods:

1) Basic Apriori algorithm
2) Apriori algorithm with binary number storage
3) Apriori algorithm with binary number storage and support count optimization
4) Apriori algorithm with binary number storage, support count optimization and candidate generation optimization

The experiment uses the Groceries dataset, setting support thresholds of 3, 4, and 5 for comparison. The time consumption (in seconds) is recorded.

The result is showed in I

Support	Method 1	Method 2	Method 3	Method 4
3	4105	2567	2258	16
4	1542	793	590	10
5	759	388	239	8

TABLE I
Execution Time Comparison for Different Support Thresholds

Visualizations of the time consumption are shown below for three thresholds.

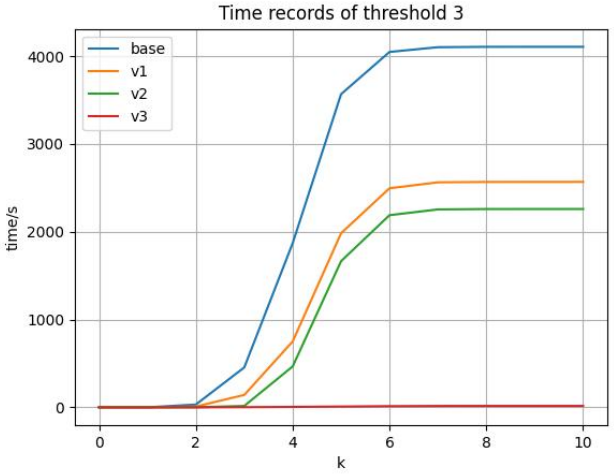

Fig. 1. Time Consumption for Support Threshold 3

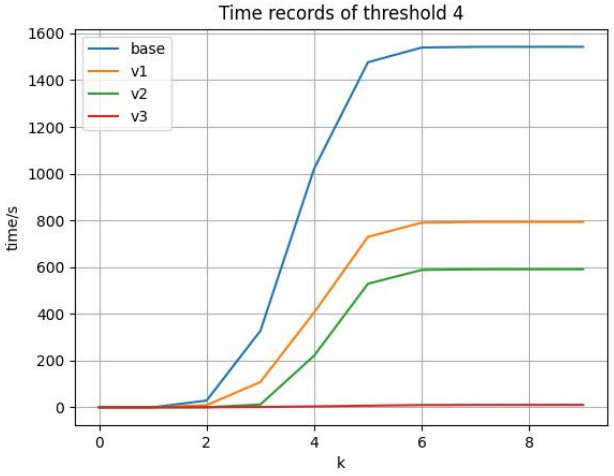

Fig. 2. Time Consumption for Support Threshold 4

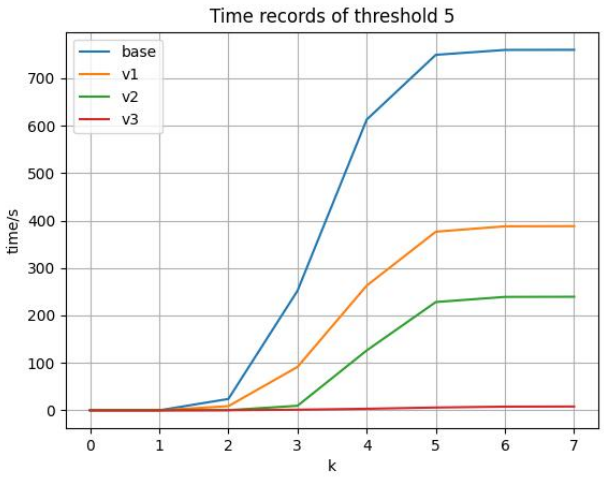

Fig. 3. Time Consumption for Support Threshold 5

The experimental results show that the improved algorithm demonstrates significant performance enhancement in handling large-scale datasets, particularly in calculation time.

1) The graphs indicate that methods 2 and 3 consume significantly less time at each stage compared to method 1, validating the computational efficiency gained by the binary number storage method. The main reason is that bitwise operations are much faster than one-by-one comparisons.
2) A closer comparison of methods 2 and 3 reveals that the primary time consumption difference lies between k = 2 and 3. Method 3 saves unnecessary candidate itemset traversal, quickly completing support counting by choosing an optimal traversal scheme.
3) Method 4 significantly enhances algorithm efficiency due to two main reasons. Firstly, by placing frequent items that can form candidate itemsets adjacent to each other, comparisons only need to be made between adjacent frequent items, reducing complexity to n instead of n^2 for n elements in frequent itemsets. Secondly, ignoring non-candidate items in the initial step, although mixed with non-candidate items, saves time compared to checking each item, thus speeding up the process.
4) Despite the current efficiency of the algorithm, further testing is needed to verify if it remains effective for larger datasets with many bits.

Through these three optimizations, the efficiency of the original program has been increased by hundreds of times, ultimately needing only 16 seconds to obtain all sets with a confidence of 3.

IV. Conclusion

After optimizing the Apriori algorithm, I have the following summaries and reflections:

1) Algorithm Understanding and Implementation: The Apriori algorithm is a classic association rule mining algorithm, mainly used to discover frequent itemsets and association rules in datasets. After understanding the basic principles of the algorithm, I implemented its core steps using Python, including generating candidate itemsets, pruning, and determining frequent itemsets.
2) Performance Bottlenecks: The initial implementation of the Apriori algorithm faced significant increases in computation time and memory consumption when handling larger datasets. The primary bottlenecks were in generating candidate itemsets and filtering frequent itemsets. As the size of the itemsets increased, the number of candidate itemsets grew exponentially, leading to a surge in computation.
3) Optimization Strategies: To enhance algorithm performance, I optimized the Apriori algorithm from three angles: data structure, candidate itemset generation, and frequent itemset filtering. This involved deeply thinking about how to reduce complexity. The improved Apriori algorithm showed significant performance enhancements in handling large-scale datasets, with reduced computation time and memory consumption, making it more efficient and practical in real-world applications.
4) Reflections:
   - Understanding the Problem: It is crucial to deeply understand the principles and performance bottlenecks of the algorithm before optimizing it. Only by clearly identifying the problem can effective optimization strategies be formulated.
   - Importance of Data Structures: Choosing the appropriate data structure significantly impacts algorithm performance. In big data processing, the selection of data structures often determines the efficiency of the algorithm.
   - Optimization from Overall to Specific: When optimizing, it is essential to consider the algorithm as a whole first, then divide it into several parts, and think in fine-grained detail about which steps can reduce complexity. This approach leads to better results.
   - Continuous Improvement and Optimization: Optimizing an algorithm is a continuous improvement process that requires constant experimentation and adjustment to find the optimal solution.

In summary, through the practice of optimizing the Apriori algorithm, I not only enhanced my understanding and application of the algorithm but also deeply appreciated the importance and challenges of algorithm optimization. In the future, I hope to continue exploring more optimization strategies and methods for various algorithms to improve their performance and application value.