Evaluation of the Apriori Algorithm

Name: Emmanuel Derry

Index Number: PS/MCS/22/0012

University of Cape Coast

Title: Efficiency, Optimization Techniques, and Scalability Analysis of the Apriori Algorithm

ABSTRACT

The apriori algorithm is one among the many algorithms that are used to mine frequent itemset in a transaction database. This paper delves into the core computational complexities of the Apriori algorithm, examining its worst-case time and space complexities. Candidate itemset generation and the use of the Apriori property are explored in optimizing the algorithm's performance. This study begins by providing a comprehensive overview of the Apriori algorithm, elucidating its theoretical underpinnings, and the fundamental concepts of frequent itemset generation and association rule discovery. As observed in the literature there are drawbacks to the originally proposed algorithm. Therefore, this paper also evaluates some of the ways to improve the algorithm's efficiency.

INTRODUCTION

A method for mining single-dimensional, single-layer, and Boolean association rules was developed by R. Agrawal et al. in 1993. The fundamental concept is to perform a layer-by-layer search recursively. Itemsets that include all frequent itemsets are candidates (Du et al., 2016). The Apriori property, which stipulates that all nonempty subsets of a frequent itemset must also be frequent, is the basis for the algorithm's name (Singh et al., 2017).

1) First step: Search the transaction database D, determine the level of support for each 1-candidate item set C, and compare it to the minimal level of support min_sup. Then, produce frequent 1-item sets L_1 that were not less than min_sup.
2) Second step: L_1 join itself ($L_1 \infty L_1$) and prunes it to generate the 2- candidates C_2, then choose the item which is bigger than min_sup to generate frequent 2-item sets L_2.
3) Third step: Use L_2 to generate L_3, the process is operated repeatedly and iterative until the result meets the final qualification $L_k = \emptyset$, the algorithm ends.

Apriori is the most well-known and fundamental method in mining association rules, and its principle is to find valuable association rules whose support and confidence must satisfy the minimum support and confidence confirmed by the user beforehand (Xie et al., 2008). Mining association rules is currently one of the main contents of data mining research, with the emphasis particularly on finding the relation of different items in a database.

Apriori is the most well-known and fundamental method in mining association rules, and its principle is to find valuable association rules whose support and confidence must satisfy the minimum support and confidence confirmed by the user beforehand. (Du et al., 2016). Mining association rules is currently one of the main contents of data mining research, with the emphasis particularly on finding the relation of different items in a database.

LITERATURE REVIEW

The ***Apriori algorithm*** is an efficient association discovery algorithm that filters item sets by incorporating item constraints (support) (Srihari, n.d.) Association rule mining has been widely

used to solve data mining problems in numerous applications, including financial analysis, the retail industry, and business decision-making (Sandhu et al., 2010). The algorithm has two filtering stages with two user-defined threshold values. Its completeness mining on a subset of given data, lower support threshold + a method to determine the completeness. (Srihari, n.d.). The Apriori Algorithm uses a layer-by-layer search iterative method to find the largest k-term frequent set. First, the database is traversed and searched to get the candidate 1 itemset and its support. If its support is lower than the minimum support, it is pruned to get the frequent 1 itemset. According to the properties, a corollary is obtained that the superset of infrequent itemsets must be infrequent (Zheng & Chen, 2022). Using this property and inference, all levels of frequent itemsets that meet the threshold of support and credibility.

APRIORI PROCEDURE

Step 1. The database is traversed once and irrelevant transaction item records are deleted.)e total number of transaction items is set as m and the traversal database as D. When D_x (x = 1, 2, …m) count =1, D_x is deleted and traversed repeatedly to get a new dataset D! (Zheng & Chen, 2022). Knowledge discovery in databases is the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data. For example, 10^4 frequent 1-itemsets will generate 10^7 candidate 2-itemsets. To discover a frequent pattern of size 100, e.g., {a1, a2, …, a100}, one needs to generate $2^{100} \approx 10^{30}$ candidates. The following is a formal statement of the problem: Let I= $\{i_1, i_2, i_3, \ldots i_m\}$ be a set of items, called items. Let D be a set of transactions, where each transaction T is a set of items such as $T \subseteq I$. Associated with each transaction is a unique identifier, called TID. Therefore, a transaction T contains X, a set of some items in I, if and only if $X \subseteq T$. An association rule is an implication of the form $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds in the transaction set D with confidence c if c% of transactions in D that contain X also contain Y. The rule $X \Rightarrow Y$ has supports in the transaction set D if s% of transactions in D contains $X \cup Y$. Given a set of transactions D, the problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified minimum support (called minsup) and minimum confidence (called minconf) respectively (Yang et al., 2009). Note that, this problem statement is neutral with respect to the representation of D. For instance, D could be a data file, a relational table, or the result of a relational expression.

Apriori Description

The Apriori algorithm is the most effective method that candidate k+1-itemsets may be generated from frequent k-itemsets according to the nature of the Apriori algorithm that any subset of frequent itemsets are all frequent itemsets. See the classical pseudo code of the Apriori algorithm below.

```
L1 = {large 1-itemsets};
for (k=2; Lk-1 ≠∅;k++) do begin
    Ck=apriori-gen(Lk-1);
```

//new candidate itemsets generated
For all transactions, t∈ D do begin
Ct=subset(Ck,t);
//transaction t contains in the candidate itemsets
for all candidates c∈ Ct do
c.count++;
end
L_k= {c ∈ C_k |c.count ≥ minsup}
end
Answer=$_k L_k$

See also the flowchart of the Apriori algorithm below.

Analysis of Apriori Algorithm

- ✓ In the connection operation, use L_{k-1} joining itself ($L_{k-1} \infty L_{k-1}$) to generate C_k, in this case, there is a judgment to decide whether L_{k-1} satisfies the join condition, but this judgment will take many comparisons, for example, supposing there are n items in L_{k-1} the time complexity of join judgment is O((k-1) * n^2). Besides, the operation also generates a huge candidate, for example, if the item number of L_1 is 100, it will generate 4950 C_2, and with the increase of the item number in L_1, the item sets of C_2 will be more.
- ✓ In the pruning operation, mark (∀ C_k ∈ C_k), it spends a lot of time on repeat scanning the database to determine whether all C_k`s combination (k-1) items belong to L_{k-1}
- ✓ During the generation of L_k, it will scan the database n times to calculate the support of c_k and make a comparison with the minimum support.

It is not difficult to conclude from the analysis of the aforementioned three points that the Apriori algorithm has three issues that could hinder its performance (Du et al., 2016). These issues are: likely producing a large number of candidate item sets; needing to frequently scan the database; and requiring a significant amount of support computation. The second phase, based on Inference 1, assesses if the requirements for creating the next candidates are satisfied by the frequent item sets currently in use. If so, move on to the next question; otherwise, the frequent item sets provide the solution.

(Xie et al., 2008) also noted that the Apriori algorithm's bottleneck is caused by the necessity of repeatedly scanning a huge database and producing a high number of meaningless candidates itemsets.

Problem Description

Similarly, following the previous analysis of the apriori algorithm, (Xie et al., 2008) further analyzed the algorithm and reported the following three shortcomings.

i. First, because there are lots of transactions in the database, computing the frequencies of candidate itemsets must scan the database frequently and spend much more time.
ii. Second, Ck is generated by joining two frequent itemsets that belong to L_{k-1}. If we can reduce the operating frequencies, we can improve the efficiency of the Apriori algorithm.
iii. Third, the expending in time and space for frequent itemsets is too much. For example, if we have 10^4 frequent 1-itemsets, we can get 10^7 frequent 2- itemsets.

Therefore, lowering the magnitude of Ck can significantly increase the Apriori algorithm's efficiency. Therefore, in order to lower the size of the database and the number of itemsets produced from Ck by the join method, it is required to eliminate unnecessary transactions from the database.
When C_k+1 is generated from $L_k \infty L_k$, each item of the first L_k is joined with every item in the second L_k in the classical Apriori algorithm.

The Apriori technique has undergone a number of changes in recent years in an attempt to address some of its drawbacks, including the need to constantly search the database and generate large candidate sets. The core concept is to use the logical "And" operation to retrieve the transaction matrix and sing the matrix to generate frequent item sets, which could improve computational efficiency and reduce calculation time (Du et al., 2016). Some researchers transformed the transaction database into a matrix to simplify its operation.

METHODOLOGY

Apriori employs an iterative approach known as a level-wise and breadth-first search, in which k-itemsets are used to generate (k+l)-itemsets (Chang & Liu, 2011). However, the algorithms based on generated and tested candidate itemsets have two major drawbacks:

i. The database must be scanned multiple times to generate candidate sets. Multiple scans will increase the 110 load and is time-consuming.
ii. The generation of huge candidate sets and the calculation of their support will consume a lot of CPU time.

Improving the Apriror Algorithm

Reducing the creation of candidate itemsets and the number of transaction records to be compared while acquiring itemset support are two general strategies to improve the process of mining frequent itemsets (Zheng & Chen, 2022). It is important to note that the main difficulty researchers have faced with frequent itemset mining has been to shorten the execution time. Effective implementation and the usage of efficient data structures are crucial for enhancing the Apriori algorithm's performance (Ansari et al., 2018). The performance of the Apriori algorithm may be faster if the items are sorted during the first iteration, according to some recent research that has taken item ordering into consideration.
According to (Ji et al., 2006), one of the ways of improving the algorithms is to reduce scans of the database or redundant generation of itemsets. Everything is available for generating the trie with the frequency order because the precise frequency order is only known after the first scan of the entire database, and this is done without adding any overhead to the original process.

Some ways to improve the Apriori Algorithm

Trie Structures (Prefix Trees): The discussion of Trie structures focused on how they significantly enhanced the Apriori method. Tries provide speedy superset/subset checks, efficient candidate generation, and fewer redundant transaction database scans. Due to these benefits, candidate generation and pruning have time complexity of O (1), making them nearly constant-time operations. Trie structures also offer space complexity (O(M)) that is often manageable for the majority of real-world datasets. This complexity is related to the number of different objects in the dataset.

MapReduce Framework: The usage of the MapReduce framework to construct the Apriori algorithm was discussed. MapReduce allows for the distributed processing of enormous datasets, which makes it ideal for routine itemset mining. The MapReduce-based Apriori algorithm's large O notations were given, emphasizing the linear time complexity in terms of the quantity of transactions and size of the itemset. The intermediate data generated during the map and reduce phases determines the space complexity.

Trie-Based Apriori

A tree-like data structure called a Trie, which stands for "reTRIEval," is used to hold a dynamic set of strings. Each string serves as an itemset in the context of the Apriori algorithm, and the Trie structure facilitates efficient organization and search of these itemsets.

The root is defined to be at depth 0 and a node at depth d can point to nodes at depth d + 1. A pointer is also referred to as an edge or a link. Each node represents an item sequence which is the concatenation of labels of the edges that are on the path from the root to the node so that a path from the root to each node represents an itemset (Ansari et al., 2018). In this implementation, the value of each node is the support count for the itemset it represents. In Fig. 1 a trie is displayed alongside the path to the node representing itemset {A, B} whose support count is 7.

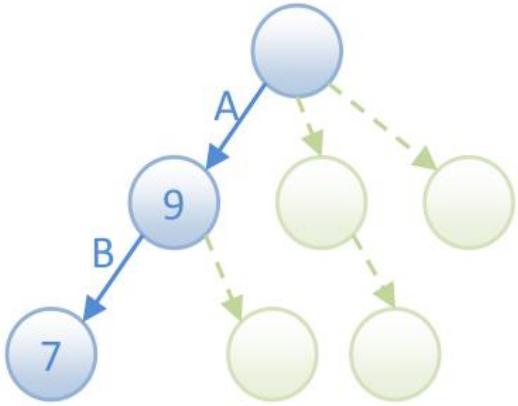

Fig. 1. A sample *trie*.

How Tries Optimize the Apriori Algorithm:

i. ***Efficient Candidate generation***
Tries are especially helpful for quickly producing candidate itemsets. By combining frequent itemsets of length (k-1) to produce itemsets of length k, you must produce new candidates in Apriori. By arranging the frequent itemsets in a systematic way, a trie can make this easier.

ii. ***Quick superset/Subset Checks***
Makes it simple to determine whether or not a candidate itemset is a subset of a bigger itemset. The Apriori algorithm's pruning procedures depend on this characteristic. A candidate itemset can be quickly pruned without additional review if it turns out to be a subset of a known frequent itemset.

iii. ***Reduce Redundant Scans***
One of the major computational bottlenecks in the Apriori algorithm is scanning the transaction database repeatedly to count the support of candidate itemsets. Tries can help reduce redundant scans by efficiently tracking the support of prefixes of itemsets. This means that you don't have to scan the entire database every time; you only need to scan parts of it that are relevant to the current candidate.

Time and space complexities using the Tries data structure

The size of the transaction database, the number of different items, and the desired minimum support threshold all affect how time- and space-complex the Apriori algorithm is, whether it uses Trie structures or not. However, the analysis below shows generic Time and Space complexities using the Tries data structure.

1. **Candidate Generation with Tries:**
    - Time Complexity: O (1)

        - The Trie structure enables constant-time candidate generation for the next level because you can efficiently traverse and combine frequent itemsets using the Trie.
2. **Support Counting with Tries:**
    - Time Complexity: O(n), where n is the number of transactions in the database.
        - Counting the support for a candidate itemset using Tries typically requires scanning the transaction database once. This is done by traversing the Trie for the candidate itemset.
3. **Pruning with Tries:**
    - Time Complexity: O (1)
        - Pruning using Tries involves quick subset/superset checks, which can be done in constant time due to the Trie structure.
4. **Space Complexity with Tries:**
    - Space Complexity: O(M), where M is the number of distinct items in the dataset.
        - The Trie structure's space complexity is proportional to the number of distinct items in the dataset. It may require additional memory to store counts associated with nodes in the Trie.
5. **Overall Complexity with Tries:**
    - The overall time complexity of the Apriori algorithm with Trie structures is dominated by support counting, which is linear in the number of transactions (O(n)). The constant-time complexities for candidate generation and pruning significantly reduce the computational load compared to the non-Trie version of the algorithm.

Summary

Comparatively, the time and space complexities of the apriori algorithm if the Tires data structure is employed perform better. Thus, execution and memory usage are optimized in this approach. As initially stated, the Apriori algorithm is sensitive to the minimum support threshold and dataset characteristics.

MapReduce

Hadoop's MapReduce parallel programming approach was created for the parallel processing of massive amounts of data. In order to map reduce and convert input datasets into (key, value) pairs, we must construct two dependent methods using the Apriori algorithm on the MapReduce architecture (Pengfei et al., n.d.) Candidate itemset generation and frequent itemset generation are the two key iterative steps that make up the Apriori algorithm (Singh et al., 2018). Each task scans the database, produces local candidates using Mapper, and then sums the local count and outcomes from frequent itemsets using Reducer. The disadvantages of serial algorithms include high memory usage and I/O costs associated with database repetitive scanning. Numerous sequential and parallel strategies have been proposed to boost the Apriori algorithm's performance. The Hash-based strategy, Transaction minimization, Partitioning, Sampling, and Dynamic itemset counting are a few common sequential approaches.

The transactional database is divided into blocks by HDFS, which then distributes each block to each machine's running mapper. The list of items, or transactions, is turned into (key, value)

pairs for each transaction, with the TID serving as the key (Singh et al., 2017). One transaction is read at a time by the mapper, which produces (key, value) pairs with the key being each item in the transaction and the value being 1. The combiner makes the local sum of the values for each key and combines the pairs with the same key. The output pairs of all combiners are shuffled & and exchanged to make the list of values associated with the same key', as (key', list (value'')) pairs. Reducers take these pairs and sum up the values of respective keys. Reducer output (key', value''') pairs where key' is the item and value''' is the support count ≥ minimum support, of that item.

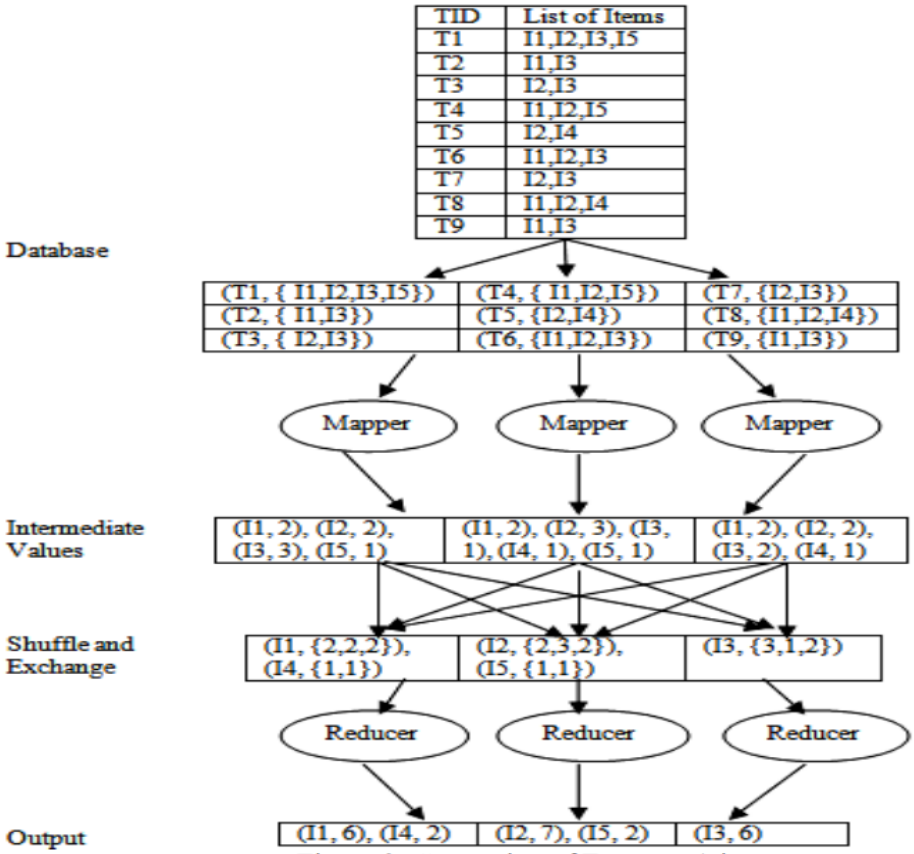

Figure 3: Generation of Frequent 1-itemsets.

Apriori Algorithm Using Map Reduce

Using the results of the first stage, we determine a support threshold for each item. Using the result of the first Map Reduce task, the second Map Reduce job once more scans the database to produce k-itemsets by trimming rarely used items in each session.

ID	ITEMSET
1	M O N K E Y
2	D O N K E Y
3	M U C K Y
4	M A K E
5	C O O K E

First, map-reduce conducts the mapper and reducer functions for this transaction list and determines the frequency count for each item. The initial phase is now complete. For use in further calculations, this output is saved in temporary files. The second map shrinks and scans all transactions once more while frequently calculating k-itemsets with count.

ITEMSET	COUNT
M	3
O	4
N	2
K	3
E	4
Y	3
D	1
A	1
U	1
C	2

As we have considered number of transactions is 5. Let's assume a minimum support value which is given input is 60% Support = minimum support * number of transactions = 60/100 * 5 = 3. Support =3. Using this support value prune all itemsets which has a lesser frequency count than the support value. The pruned output will be shown in the below table.

ITEMSET	COUNT
M	3
O	4
K	3
E	4
Y	3

Similar way we will find k-itemsets by pruning infrequent itemsets from the database. We will get the following itemsets for 2 sizes of itemsets.

ITEMSET	COUNT
M, K	3
O, K	3
O, E	3
K, E	4
K, Y	3

For size 3 of itemset:

ITEMSET	COUNT
O, K, E	3

Time and space complexities using Parallel processing (Hadoop MapReduce)

1. **MapReduce for Candidate Generation:**
    - Time Complexity: O(n * k * m), where n is the number of transactions, k is the maximum itemset size, and m is the number of unique items.
        - In this step, each map task generates candidate itemsets by combining frequent itemsets of size (k-1). The map tasks run independently, but the overall time complexity is still linear in the number of transactions and items.
2. **MapReduce for Support Counting:**
    - Time Complexity: O(n * k), where n is the number of transactions, and k is the maximum itemset size.
        - Each map task independently counts the support for the candidate item sets it has been given. The counts for each item set are combined during the reduction step. The time complexity is a linear function of the itemset size and the number of transactions.
3. **MapReduce for Pruning:**
    - Time Complexity: O(k), where k is the number of candidates itemsets to be pruned.
        - Pruning involves comparing candidate itemsets with known frequent itemsets. This step can be performed with low computational complexity.
4. **Space Complexity:**
    - The intermediate data generated during the map and reduce phases is often what determines the space complexity in MapReduce. It depends on how many maps there are, how many activities have been decreased, and how much data each task processes.
5. **Overall Complexity:**
    - The MapReduce-based Apriori algorithm's overall time complexity is influenced by the step that takes the longest to complete, which is typically the support counting phase. Since n is the number of transactions and k is the largest possible itemset size, the time complexity is frequently O (n * k).

Summary

Parallel processing can be a powerful technique to accelerate the Apriori algorithm, especially when dealing with large datasets. However, it requires careful design, data partitioning, and consideration of potential challenges such as communication overhead (data serialization and deserialization) to achieve optimal results. MapReduce is an efficient, flexible, and simple model for large computational problems in a distributed computer framework.

DISCUSSIONS

The Apriori algorithm is a widely used technique for frequent itemset mining in association rule learning. To enhance its performance, several strategies and data structures have been discussed, including Trie structures and parallel processing.

Scalability:

Apriori is known for its combinatorial nature, which can result in a large number of candidates itemsets, especially when dealing with large datasets. Hadoop MapReduce is designed to handle scalability effectively by distributing the computation across multiple nodes in a cluster. With MapReduce, you can parallelize the frequent itemset generation process across multiple mappers, allowing you to process large datasets more efficiently.

Distributed processing:

Hadoop MapReduce enables distributed processing, which means that each mapper can independently generate local frequent itemsets from a portion of the dataset. This parallelism significantly reduces the overall execution time. The reducer phase can then merge and aggregate these local frequent itemsets to produce the final global frequent itemsets.

Data Shuffling:

Apriori involves multiple iterations to generate increasingly longer frequent itemsets. During these iterations, intermediate results need to be shuffled and sorted to identify common items. Hadoop's built-in data shuffling capabilities can efficiently handle this process, reducing the need for custom implementations.

Fault Tolerance:

Hadoop MapReduce offers built-in fault tolerance mechanisms. If a node fails during processing, Hadoop can automatically reroute tasks to other available nodes, ensuring that the computation continues without data loss or disruption.

Resource Utilization:

Hadoop's resource management system, like YARN, optimizes resource allocation for MapReduce jobs. This ensures that cluster resources are used efficiently, which is particularly important when running Apriori on large datasets.

Optimization Techniques:

Various optimization techniques, such as pruning strategies to reduce the number of candidates itemsets, can be integrated into the MapReduce implementation of Apriori to further improve efficiency. Combiner functions in MapReduce can be used to aggregate intermediate results

before they are sent to reducers, reducing the volume of data shuffled and improving performance.

Performance Trade-offs:

While Hadoop MapReduce can significantly improve the efficiency of Apriori for large datasets, there may be some performance trade-offs, such as increased overhead due to data serialization and deserialization. However, the benefits of distributed processing often outweigh these trade-offs.

CONCLUSION

Applying the MapReduce approach to mine the frequent itemset in a large database the algorithm will be improved and optimized. In conclusion, the Apriori algorithm's efficiency and scalability can be greatly improved by incorporating Trie structures and parallel processing techniques. Trie structures offer advantages in candidate generation and pruning, reducing the computational load and enhancing overall performance. Parallel processing, especially within the MapReduce framework, allows for the distributed processing of large datasets, further speeding up the algorithm. However, it's important to carefully address challenges related to data partitioning, communication overhead, and memory management when implementing parallelism. By combining these strategies, practitioners can efficiently mine frequent itemsets and association rules in large transaction databases, making the Apriori algorithm a valuable tool for various data mining and recommendation applications.

REFERENCES

Ansari, E., Sadreddini, M. H., Mirsadeghi, S. M. H., Keshtkaran, M., & Wallace, R. (2018). TFI-Apriori: Using new encoding to optimize the apriori algorithm. *Intelligent Data Analysis*, *22*(4), 807–827.

Chang, R., & Liu, Z. (2011). An improved apriori algorithm. *Proceedings of 2011 International Conference on Electronics and Optoelectronics*, *1*, V1-476.

Du, J., Zhang, X., Zhang, H., & Chen, L. (2016). Research and improvement of Apriori algorithm. *2016 Sixth International Conference on Information Science and Technology (ICIST)*, 117–121.

Ji, L., Zhang, B., & Li, J. (2006). A new improvement on apriori algorithm. *2006 International Conference on Computational Intelligence and Security*, *1*, 840–844.

Pengfei, G., Jianguo, W., & Pengcheng, L. (n.d.). Research and Improvement of Apriori Algorithm Based on Hadoop. *International Journal of Advanced Network, Monitoring and Controls*, *3*(3), 100–105.

Sandhu, P. S., Dhaliwal, D. S., Panda, S. N., & Bisht, A. (2010). An Improvement in Apriori algorithm using profit and quantity. *2010 Second International Conference on Computer and Network Technology*, 3–7.

Singh, S., Garg, R., & Mishra, P. K. (2017). Review of apriori based algorithms on mapreduce framework. *ArXiv Preprint ArXiv:1702.06284*.

Singh, S., Garg, R., & Mishra, P. K. (2018). Performance optimization of MapReduce-based Apriori algorithm on Hadoop cluster. *Computers & Electrical Engineering*, *67*, 348–364.

Srihari, S. N. (n.d.). *Apriori Algorithm for Sub-category Classification Analysis of Handwriting*.

Xie, Y., Li, Y., Wang, C., & Lu, M. (2008). The optimization and improvement of the Apriori algorithm. *2008 International Workshop on Education Technology and Training & 2008 International Workshop on Geoscience and Remote Sensing*, *2*, 663–665.

Yang, G., Zhao, H., Wang, L., & Liu, Y. (2009). An implementation of improved apriori algorithm. *2009 International Conference on Machine Learning and Cybernetics*, *3*, 1565–1569.

Zheng, Y., & Chen, Y. (2022). The Identification of Chinese Herbal Medicine Combination Association Rule Analysis Based on an Improved Apriori Algorithm in Treating Patients with COVID-19 Disease. *Journal of Healthcare Engineering*, *2022*.