SSRG International Journal of Computer Science and Engineering
ISSN: 2348–8387 / https://doi.org/10.14445/23488387/IJCSE-V11I9P104

Volume 11 Issue 9, 26-38, September 2024
© 2024 Seventh Sense Research Group®

Original Article

Augmenting Association Rule Mining in Apriori Algorithm using Cuckoo Search with Opposition Parameters-Based Learning

N. Bhanu Prakash[1], E. Kesavulu Reddy[2]

[1,2]Department of Computer Science, S.V.University College of CM&CS, Tirupati, Andhra Pradesh, India.

Corresponding Author : ekreddysvu2008@gmail.com

Received: 22 July 2024 Revised: 30 August 2024 Accepted: 16 September 2024 Published: 30 September 2024

Abstract *- Data mining extracts hidden patterns from large datasets, making the information extracted useful for improving decisions and, hence, business outcomes. Among these methods, frequent itemset mining is a very popular and core technique within association rule mining The Apriori algorithm is one of the most popular algorithms in this area of frequent itemset and association rule discovery. Applications include market basket analysis, educational course selection, stock management, and medical data analysis. However, large datasets are exponentially increasing the computational burden of the Apriori algorithm, and hence, execution on parallel-distributed environments can improve performance. The improved approach presented in this paper integrates the Apriori algorithm with the Cuckoo Search algorithm using opposition parameters-based learning (CS-OPBL). The Cuckoo Search mechanism with opposition-based learning efficiently prunes the transactions and items in each transaction. It is an approach whose processing time is greatly reduced if executed on a Spark in-memory distributed environment. The experimental results showed that the proposed CS-OPBL-based method outperforms the competing algorithms; for example, at a minimum support threshold of 0.75%, the processing time of this approach is only about 5.8% of that by using the state-of-the-art method on the retail dataset.*

Keywords *- Data Mining, Frequent Itemset Mining (FIM), Association Rule Mining, Apriori Algorithm, Cuckoo Search and Spark.*

1. Introduction

The data age is where we find ourselves today, wherein many sources, including social media platforms, sensors, search engines, medical records, and more, are constantly producing data [1]. Supporting people in getting information is needed right away(valuable stuff) from this data. In databases, this procedure is known as discovery of knowledge (KDD) [2]. "Data mining," or the act of identifying patterns from huge datasets, is a crucial component of KDD. Applications for data mining may be found in marketing, finance, education, telecommunications, fraud detection, and medicine. The Apriori method is incremental and operates step by step. During each cycle, it systematically examines the database to provide a vast quantity of potential candidates derived from common itemsets. The typical execution takes place on a solitary system, which cannot handle such a substantial volume of data. In order to handle concerns such as data duplication and synchronization, it is necessary to use numerous computers and a parallel method. The Apriori algorithm's significant limitations in terms of computing complexity render it inefficient for usage with bigger data sizes.

A detailed account of an empirical investigation into the Apriori algorithm can be found in reference [4]. An experiment was carried out using a sample of 2,000 transactions from a total of 2,064 hospital transactions. The Apriori algorithm's runtime has a positive correlation with a list of all the deals, indicating that as the data amount rises, the method takes longer to execute. The following provides essential context for our work.

1.1. Association Rule Mining

Association Rule Mining (ARM) is an approach to mining data that is used to discover common definitions of patterns and significant relationships between factors in extensive datasets. It facilitates making choices by identifying the correlation between various aspects of a database. There are measurements of Rank and sort things by how interesting they are in choosing highly intriguing regulations, such as trust and help. In dataset D with N transactions D={T_1, T_2,....., T_N}, A part of each transaction is made up of items from I, which includes M items I={i_1, i_2,…., i_M}. If Separate Z and L subsets of I such that $Z \cap L = \emptyset$and a rule Z =>L exists, where Z is the precursor (or left side) and L is the result (or right side).

$$support\ Z = \frac{count\ (Z)}{N} \qquad (1)$$

The proportion of transactions that include every item in the itemset Z (number of transactions including Z/total the number of transactions) is supporting for itemset Z. If an item set's support hits or goes above the minimum support level, it is said to be common and is given as,

$$confidence\ (Z \longrightarrow L) = \frac{support\ (ZL)}{support\ (Z)} \qquad (2)$$

The confidence of a rule (Z□L) is determined by dividing the number of transactions that include both Z and L by the number of transactions containing Z. This shows how likely it is that L will happen in a deal that has Z in it.

1.2. Apriori Algorithm

The goal of Apriori is to iteratively identify frequently used item sets and Create rules for how to connect different things. For any integer k greater than or equal to 1, the kth iteration of Apriori produces sets of frequently occurring items. The method begins by constructing frequent itemsets, referred to as L1, which consist of single items (1-itemsets). The algorithm can recursively generate a frequency of 2-itemsets (L2) frequent of 3-itemsets (L3) until there are no more sets of things that meet the predefined minimum support threshold. Each iteration builds upon the results of the previous one to generate a new set of candidate itemsets. It follows that {A} and {B} must likewise be common itemsets if {AB} is found to be one. This is due to the principle that frequent itemsets must be derived from other frequent itemsets.

- ➢ Finding the sets of objects that appear most often in the database with little help
- ➢ Association rules are generated by using collections of objects that appear frequently.

There are two main parts to the process of creating Apriori frequent itemsets:

- ➢ In each iteration k, the join phase involves joining L_{k-1} with itself to create a candidate set C_k.
- ➢ The pruning phase is utilized to eliminate k-itemsets whose support count is below the target threshold. This process results in generating collections of frequently occurring itemsets, L_k, from C_k.

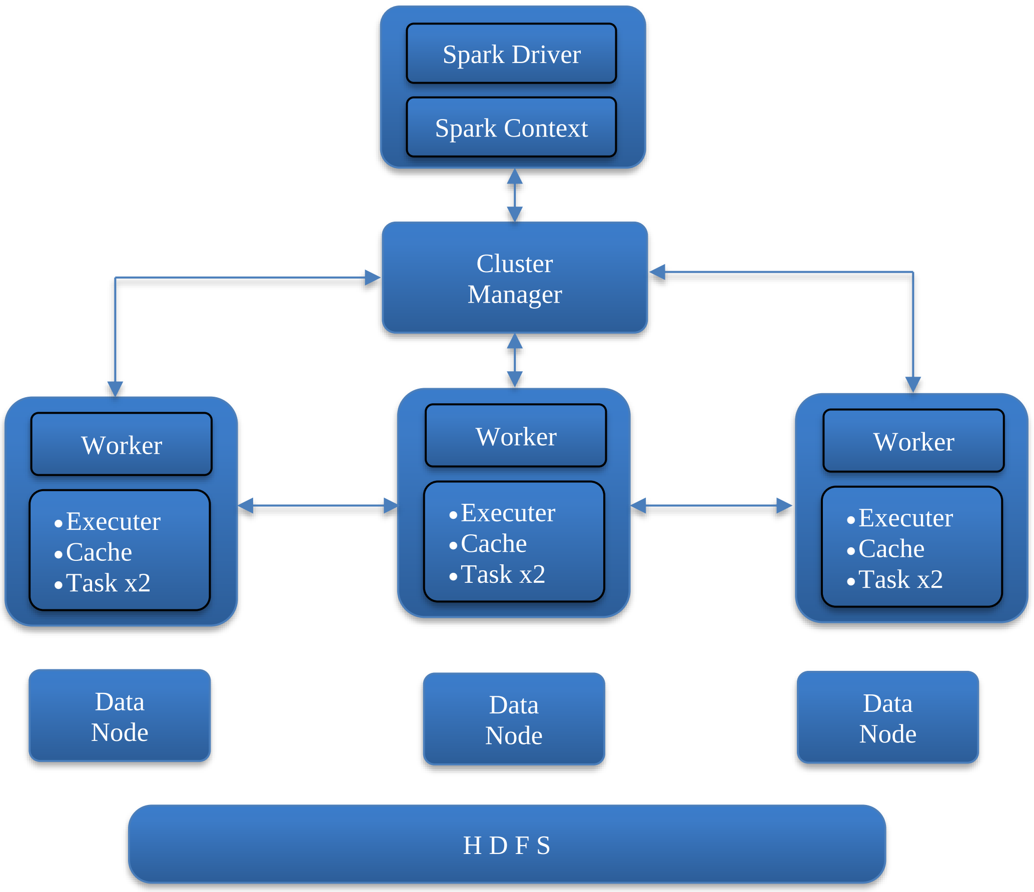

Fig. 1 Working flow of apache spark

1.3. Apache Spark Utilization

According to [5], Apache Spark is a powerful and general-purpose open-source framework for cluster computing that processes large volumes of data more efficiently than Hadoop MapReduce. Spark is reported to be ten times better in memory usage and one hundred times better in storage. This advantage arises because Hadoop MapReduce, which reads and writes data for the Hadoop distributed file system (HDFS), faces challenges with algorithmic iterations due to the high I/O strain, leading to increased processing times. It offers compatibility with Hadoop, Mesos, cloud environments, and standalone setups, unlike MapReduce, which is limited to Hadoop. It supports various databases such as HBase, Cassandra, and HDFS and is compatible with multiple programming languages, including Scala, Python, and Java. At its core, Spark utilizes the RDDs as the fundamental data structure for its programming interface.

Spark manages the automatic partitioning and distribution of RDD information spanning the cluster and performs actions on them at the same time [17]. While Hadoop achieves fault tolerance by replicating each data block three times, Spark ensures fault tolerance by tracking the lineage of each RDD. To ensure fault tolerance, Spark monitors the lineage of each RDD, maintaining references to parent RDDs when new ones are created. This lineage information is represented as a lineage graph, which records the dependencies of each RDD. This graph is useful for recomputing and recovering lost data when needed.

Additionally, Spark can cache RDDs in memory if they are accessed frequently, which accelerates subsequent operations. These distinctive features of Spark enhance where iterative programming is executed. In Spark, a central coordinator called the driver communicates with several distributed workers, or slave nodes, to execute application code. This setup is shown in Figure 1. Spark functions utilizing a master/slave architecture. Executors are responsible for data storage and computation, and the driver asks the manager of the cluster to start them with resources. Assigning tasks to executors, the driver decomposes Spark jobs. Executors report back to the driver with the outcomes of their work after it is over.

1.4. Cuckoo Filter Assembly

Probabilistic data structures like cuckoo filters and Bloom filters offer efficient time and space performance. They enable rapid and accurate membership checks for large datasets. These filters allow you to quickly determine if an item is part of a set and also to add items to the set. The following points show why a cuckoo filter is superior over a Bloom:

- ➢ The Bloom filter does not support deleting existing items without reconstructing the entire filter, a process that requires O(1) time. However, it does support efficient deletion, also with an O(1) time complexity.
- ➢ Enhancing lookup efficiency in a cuckoo filter requires only O(1) time and involves checking just two locations.
- ➢ When the target false-positive rate is below 3%, the space requirements will be reduced.

Figure 2 illustrates the construction of the cuckoo filter. In this filter, each of the m buckets. There are two distinct hash functions used by the cuckoo hash table, which allows it to hold b objects.

- ➢ When inserting or searching for an item, the appropriate location is ascertained via
- ➢ The hash functions h1(x) and h2(x)A cuckoo filtered maintains only the value of fingerprint data, generated by the hashing function = fingerprint(x).

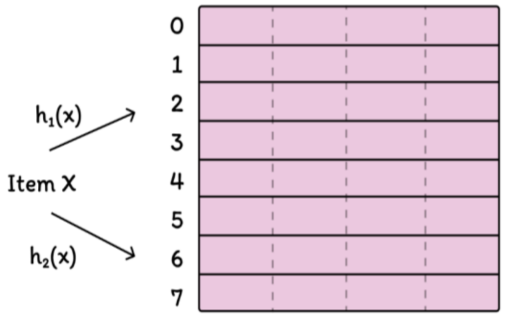

Fig. 2 CS-OPBL with two hashes with four entries each

The cuckoo filter Saves only the f-bit products' fingerprints rather than the products themselves. Both the size of each bucket b and the length of the fingerprints f in bits impact the filter's false-positive rate. The required length of the fingerprint, f, is approximately estimated in [3].

$$F \geq \lceil log_2\left(\frac{2a}{\varepsilon}\right)\rceil = \lceil log_2\left(\frac{1}{\varepsilon}\right) + log_2(2a)\rceil \text{ bits} \qquad (3)$$

An empirical study of the cuckoo filter with a = 2, 4, and 8 bucket sizes is described in reference [20]. The findings indicate that the cuckoo filter when using a = 4, achieves a low false-positive rate and excellent space efficiency.

1.5. Research Gaps

While various methods have been proposed to enhance an Apriori method for massive data, certain issues remain unaddressed. CS-OPBL seeks to address these gaps and improve the efficiency of the Apriori method as data volumes increase. Table 1 outlines these deficiencies and how CS-OPBL addresses them. The CS-OPBL algorithm addresses the limitations of the original Apriori method. It is designed to work with distributed data management platforms that support parallel storage, access, and processing of large datasets, such as ecosystems, such as Hadoop and Spark.

The CS-OPBL method ditches candidate generation of the priori algorithm's stage for a more efficient and less computationally intensive alternative. Instead, it uses the cuckoo filter to keep only the most frequently occurring items and employs these to prune transactions, thereby enhancing performance. The paper is organized as follows: Part 2 reviews key research on association rule mining. Part 3 provides an overview of the suggested techniques. Part 4 Provides the experimental outcome and discusses their alignment with previous findings. Finally, Part 5 concludes with a summary and forecast directions.

2. Related Works

Extensive research in the field of association rule mining in recent decades has resulted in numerous proposals for improving the Apriori algorithm. Agrawal and Shafer [5] proposed parallelizing the Apriori algorithm, but it struggles with performance due to timing and communication challenges as data volume increases. Using the MapReduce structure to perform Apriori was suggested by Li et al. [6]. There are two steps to this process: planning and lowering. During the mapping stage, the data is turned into pairs of keys and values, and possible candidate sets are found. In the lowering stage, the outcomes of different mappers are added together to make a final result. This result includes sets of things whose support counts meet or go above the minimum level. This step is repeated until there are no more frequent item sets to be found.

Singh and Miri [7] developed a parallel Apriori algorithm that leverages a Bloom filter to minimize the time required for subsequent runs. This algorithm is divided into three stages. Initially, the mapper and reducer identify singleton frequent items. In the next stage, the Bloom filter stores these individual items, and each transaction is pruned to include only items Contained within the Bloom filter. The Spark RDD-based parallel Apriori algorithm known as YAFIM (Yet Another Frequents Item set of Mining) [8] operates in two different phases. The initial phase identifies individual Recurring items, while another phase iteratively produces (k+1)-frequent data from k-frequent data. To expedite the process of finding (k+1)-frequent data, YAFIM uses a hash-based tree structure for managing candidate (k+1)-data, which must first meet the minimum support threshold. However, when dealing with a large number of potential item groups, YAFIM may not be as efficient as the approach proposed by Li et al. [6].

Using the Spark RDD, Rathee et al. [9] developed R-Apriori, a high-speed parallel Apriori consisting of three different stages. In the first stage, singleton frequent items are generated. The second phase involves using these singleton items stored in a Bloom filter to prune transactions, retaining only items present in the Bloom filter. In the third phase, the algorithm uses the singleton items from the previous phase to create a list of all possible item pairs and identifies the 2-frequent itemsets. The third phase then iteratively generates k-frequent item sets and candidate storage from (k+1)-frequent item sets of (k+1)-items in a hash tree, which speeds up searching.

Regarding YAFIM, this approach offers improved performance. "EAFIM [10] is a sophisticated frequent item set mining technique tailored for Spark, relying on the Apriori method. It works in two main stages: first, it creates candidate itemsets and determines their favourable ratings; second, it iteratively refines the collection of data by removing items and transactions that are not important. When compared to R-Apriori and YAFIM, EAFIM performs better. Furthermore, HFIM, a hybrid typical itemset mining technique designed specifically for Spark, was suggested by the researchers in [11]. HFIM follows a two-step process: first, it transforms the dataset into a vertical format (consisting of items and IDs) to pinpoint unique frequent items. Then, it distributes this vertical dataset across all nodes. The adaptive-miner approach, proposed by Rathee and Kashyap [12], consists of two different phases. This dynamic programming approach adapts to the dataset's structure, enhancing the identification of frequent item groups. Furthermore, Gao et al. [13] suggested enhanced Apriori techniques for Spark to tackle scalability challenges present in the original Apriori method.

The suggested method minimizes both the event count and data processing time. Castro et al. [14] assessed different Apriori algorithms on Hadoop MapReduce and Spark using various datasets with minimal intervention. A Spark-based Apriori approach was introduced by Raj et al. [15]; it improves efficiency and scalability by minimizing the costs coming from RDD shuffle tasks during each iteration that follows. Kumar and Mohbey [16,17] introduced the CEUPM (communications for the cost-effective utility-based patterns mining) algorithm, designed, which utilizes a search space division strategy to distribute tasks evenly across cluster nodes. This approach reduced communication costs during the shuffle process. The algorithm proved to be more efficient overall, as it ran faster, used less memory, and offered better scalability. They explored various methods for pattern mining in large-scale data using Hadoop, and Spark's ability to process data in parallel and distribute tasks efficiently was evaluated through four key types of itemset mining: highly-utility itemset mining, parallel frequent itemset, sequential pattern, and frequent itemset datasets (like sensor data or experimental data).

Gawwad et al. [18] introduced techniques for frequent databases that can be executed in parallel and designed to handle large datasets by utilizing the multiple cores of the hardware. Their method involved using prime numbers to determine the largest common factor among transactions, with each item in the transactions assigned a unique prime number. Meanwhile, Kumar and Mohbey [19,20] introduced the UBDM (Uncertain Big Data Mining) method, which performed effectively within the Spark framework. Furthermore, the Stellar Mass Black Hole Optimizing approach was introduced by Kannimuthu and Premalatha [21]. This method eliminates the need for an initial minimum benefit barrier by extracting the top k high-utility collections from transaction networks. This method enhances processing efficiency and reduces memory consumption by incorporating a trimming technique that eliminates unnecessary search regions.

Furthermore, the Stellar Mass Black Hole Optimizing approach was introduced by Kannimuthu and Premalatha [21]. This method eliminates the need for an initial minimum benefit barrier by extracting the top k high-utility collections from transaction networks.

This addresses a major key difficulty in mining high-utility itemsets is determining the appropriate minimum utility threshold, which defines the level of utility required for an itemset to be considered significant that varies depending on the database.

By using a combined evolutionary method for identifying high-utility itemsets in internet service design, Kannimuthu and Chakravarthy [22] were able to enhance processing speed and memory efficiency significantly. In addition, itemsets with negative utility ratings are a common challenge for the present algorithms. In order to get over these restrictions, Kannimuthu and Premalatha [23] developed a useful pattern-growth technique that worked well in contrast to previous methods for managing sets of items with zero values. Chiclana et al. [24] suggested a new association rule mine technique inspired by animal behaviour optimized strategies. This approach aims to decrease the quantity of generated rules, as well as to lower processing time and memory usage. It eliminates unnecessary or low-support rules and retains only the frequent rules for optimization within the animal motion framework. Meanwhile, Rajagopal et al. [25] developed a crop selection strategy that outperformed other methods by enabling the selection of the most profitable crop.

2.1. Performance Comparisons of the Apriori Algorithm and its Evaluations [4]

Table 1. Analysis

S.No	Properties	Apriori	Apriori TID	Apriori hybrid	Tertius
1	Candidate generation	Apriori produces candidate item sets from ofprevious pass by not takingthe transaction in the database.	Once the first pass is completed, the database is not considered for counting supportof candidate itemsets	It generates Candidate itemsets by using Apriori but later jumps to Apriori TID.	Candidates are generatedby considering attribute pairs for the rule generation.
2	Methodology	Join and prune phases/steps	Considers Join and Prune incombination with TIDS	Combination ofapriori andaprioritid	First-order logic presentation is preferred
3	Database scan	Needs many scans of databases	Needs only one scan	Addition of Apriori and aprioritid	Scan depends on the countof literals in rules
4	Memory usage	It occupies high memoryspace for the process of candidate generation	In the first pass, this algorithm needs memory for Lk-1 and Ci-1 candidate generation. It indulges extra cost in case it does not fit in memory.	It infers extra memory when sliding from Apriorito Apriori TID	Consumes considerable time and printsout rules when the program runs short of memory and messages
5	Execution time	Mainly spends more on Candidate Generation	Executes fast in contrast to Apriori for small problems but incurs more time for large ones.	Preferably better than Apriori and Apriori TID	Consumes a considerably long time for larger sets, i.e.even hours
6	Data support	Limited	Nearly large sets	Very Large datasets	Limited
7	Accuracy	Less	Better than Apriori	Increased Accuracy compared to Apriori TID	Considerable, not high, ieAverage
8	Applications	It can be mainly preferredfor closed Item sets.	Preferred for small problems.	Well suited for closed sets.	Most generally preferred.
9	Privacy-preserving approach preferred	Heuristic approach	Exact approach	Heuristic approach	Cryptographic approach

Table. 2 Analysis

Sno	Properties	ECLAT	FP Growth	AIS	PSO
1	Candidate generation	Uses bit matrix representationof transactions and prefix tree in DFS order.	Does not generate a candidateset but takes a few passes overthe database	Scans database eachtime for generating candidates.	Candidates are generatedwhen the db scan is in progress.
2	Methodology	Bit matrix rep.& depth-first search of prefix tree construction from bit matrix	Two phases of divide and rulemethod	2 stages, first frequent item setgeneration	Uses the concept of 'neighbourhood.'
3	Database scan	Only once till matrix construction	Scans fewer no of times untilthe construction of the fp-tree.	Multiple scans	Same as AIS
4	Memory usage	Considerably less as prefixtree rep. Is considered.	Comparatively average	Occupies muchspace	Less comparatively
5	Execution time	Faster initially and averagelater	Average	Long time	Very fast
6	Data support	Large	Very large	Less	Very large
7	Accuracy	Considerably better	High when compared withApriori TID	Too small or less	Excellent
8	Applications	Mainly preferred for freeitemsets.	Preferred for large applications	Well suited forsmall problems	Large scale, including closed sets and free itemsets, etc.
9	Privacy-preserving approach preferred	Reconstruction based approach	Reconstruction based approach	Exact approach	Cryptographic approach

3. The Proposed Techniques

Apriori is a repetitive method that finds the most common sets of things and builds association rules from them in a certain order. It has two parts: first, making single-item sets of frequent items; second, making sets of frequently used items in an iterative way.

Because potential sets with all possible combos of common item sets are made in the second part of each repetition, the method is ineffective. It becomes higher priced to run as the amount of the data grows. These sets of frequent items come from earlier versions and are now being used to choose common sets of items moving forward.

They are evaluated individually against each transaction to determine the quantity of each new set of items. The main goal of the CS-OPBL algorithm was to fix the main problems with the preliminary Apriori algorithm. The answer looks into using multiple computers and a parallel method to get around the fact that Apriori's speed drops as data size grows.

If the original data is very big, the cuckoo filter can help you quickly check if an item is a member. It makes it faster to check for each transaction whether an item is often bought or not so that you can decide whether to keep it or get rid of it if it has not often been bought. There are two parts to the CS-OPBL formula.

3.1. Stage 1

Program 1 describes a program that is in charge of making all the individual common things in this phase. As a line graph of RDDs, Figure 3 displays the processing flow. The results of transactions are divided up and given to several worker computers when they are uploaded into the RDD of Spark from HDFS, allowing each worker to view them independently. These transactions are read by the map() function, which turns each one into a list of things. For every transaction, which is a list of things, the fatMap() method is used to split each item on its own. The map() method turns each item on the list into a key-value pair of the form (item, 1). After that, the ReduceByKey() method guesses what every item's function is. Then, the filter() method removes items with frequencies less than the minimum support count (min_sup), leaving only the 1-frequent itemsets with their respective items and support values. Subsequently, the keys() function is utilized to extract just the items, discarding the counts, from these 1-frequent itemsets. These items are then inserted into the cuckoo filter for further processing. The things that are used a lot are stored in memory to quicken up the next step. Once phase one is over, the cuckoo filter structure is used to store the daily things. It is shared among all of the nodes in the broadcast function with the use of a cluster of the Spark framework. The ReduceByKey() method cut down on the time needed to make the singleton common items.

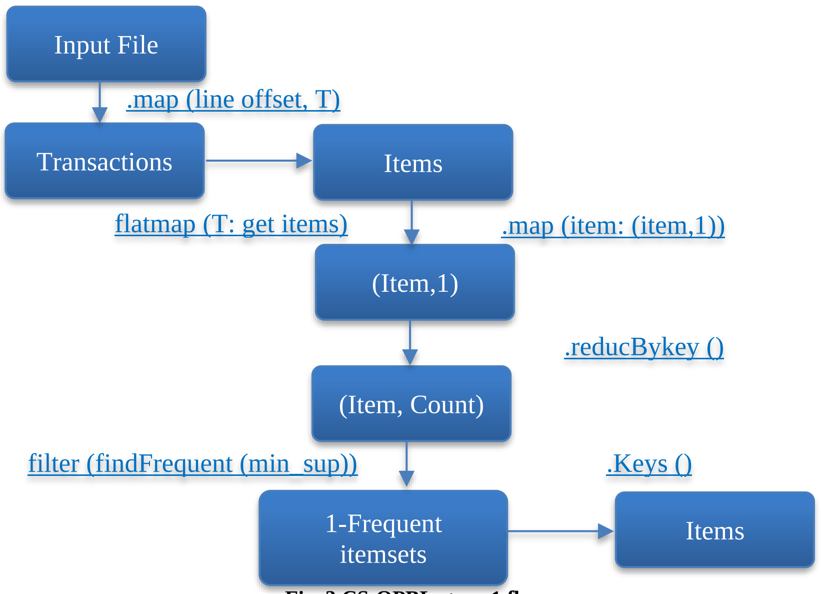

Fig. 3 CS-OPBL stage 1 flow

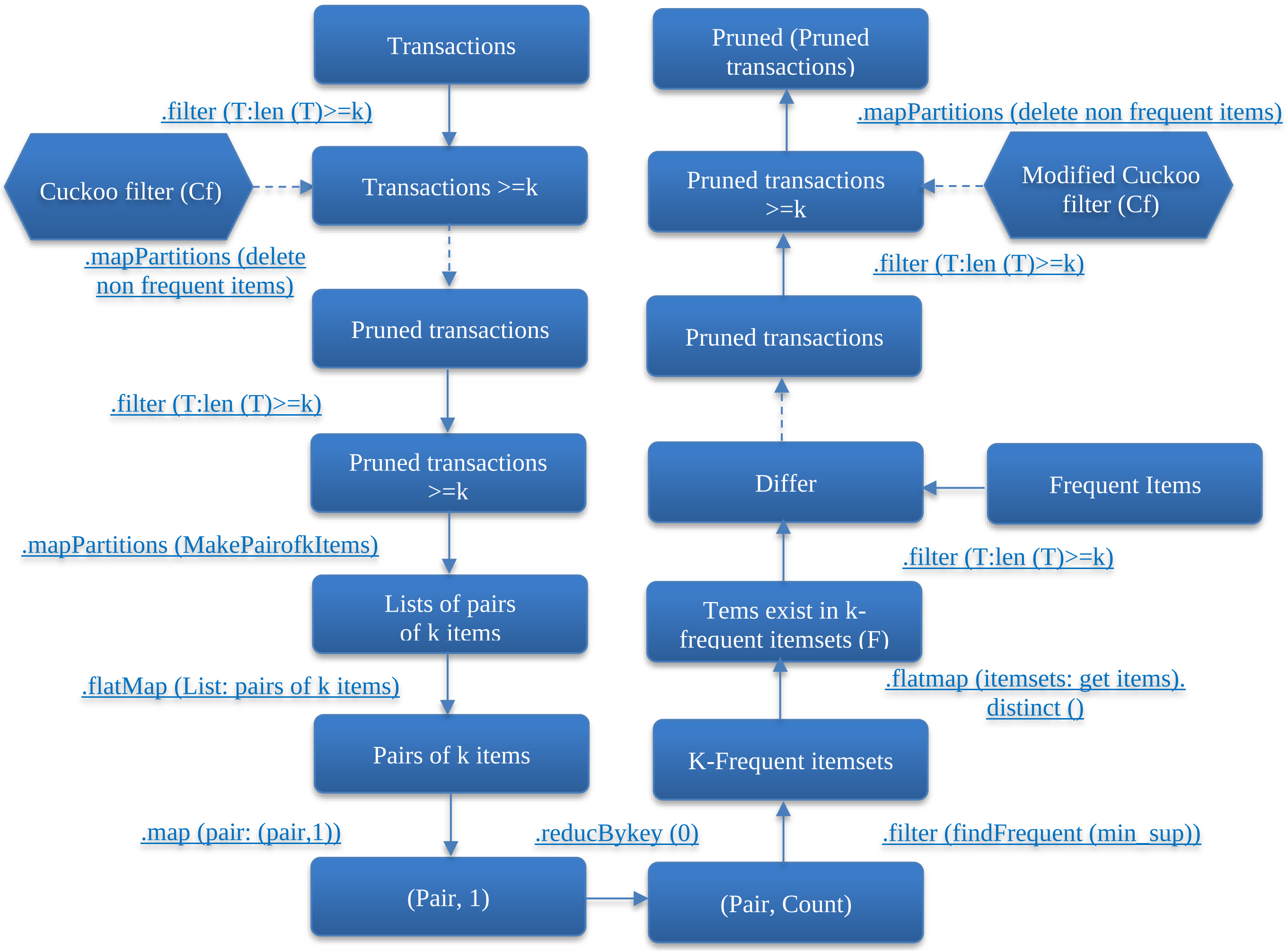
Transactions
.filter (T:len (T)>=k)
Cuckoo filter (Cf)
Transactions >=k
.mapPartitions (delete non frequent items)
Pruned transactions
.filter (T:len (T)>=k)
Pruned transactions >=k
.mapPartitions (MakePairofkItems)
Lists of pairs of k items
.flatMap (List: pairs of k items)
Pairs of k items
.map (pair: (pair,1))
(Pair, 1)
.reducBykey (0)
(Pair, Count)
.filter (findFrequent (min_sup))
K-Frequent itemsets
.flatmap (itemsets: get items). distinct ()
Tems exist in k-frequent itemsets (F)
.filter (T:len (T)>=k)
Differ
Frequent Items
Pruned transactions
.filter (T:len (T)>=k)
Pruned transactions >=k
Modified Cuckoo filter (Cf)
.mapPartitions (delete non frequent items)
Pruned (Pruned transactions)

Fig. 4 CS-OPBL stage 2 flow

3.2. Stage 2

In this part, the method shown in method 2 works by going through steps over and over again. This function creates the k-frequent item sets (k > 1) from scratch. To clean up deals and reduce the number of items in each one, CS-OPBL uses the cuckoo filter structure. Instead of generating every possible candidate itemset, the algorithm constructs candidate sets by pruning transactions to include only those with at least kk items, where kk denotes the length of the frequent itemsets to be found. Transactions shorter than kk are excluded, as they cannot produce kk-item candidates. The execution flow, represented by the graph of RDDs Figure4, demonstrates this method. Initially, transactions are filtered to remove all non-frequent items, retaining only those that appear in the cuckoo filter. Transactions with kk or more items, starting with k=2k=2, are then kept. Because of this, the collection has fewer deals and things. Second, the steps below are done each time the loop goes around, as long as k>=2 and k is the length of a common itemset:

- The `mapPartitions()` method works on individual partitions (or blocks) of the RDD. Following this, `flatMap()` is used to split these combinations into individual items, with each combination being handled separately.
- The `map()` function processes a list of k items and creates key-value pairs for every item combination. The `ReduceByKey()` method then applies a custom hash function, murmur-hash3 (mmh3), to determine the frequency of each composite key combination.
- If multiple k-frequent data are identified, the algorithm advances for (k + 1)-data. If no k-frequent data are found, the algorithm resets kk to 1 and concludes.

This process is carried out through the following steps:

1. The k-frequent itemsets are given the fatMap() method to split the unique items that makeup F.
2. We then compare these items with those frequently used and stored in the cuckoo filter.
3. If a discrepancy is identified, the subsequent actions are taken.
4. The cuckoo filter structure is changed by removing the items that were different in the previous step.
5. The RDD's mapPartitions () function executes on every division (block). Deals with lengths more than or equal to k are first trimmed down, and then those deals are trimmed back one more until only things in the cuckoo lens remain. In order to employ them more quickly in the next round, it then saves them in storage.
6. The execution begins at step 1 in order to begin the subsequent iteration. If there is no change, it is assumed that the k-frequent item is set from the memory, and k is increased. Step 1 is finally achieved to start the subsequent cycle.

During this stage, the Apriori method's need for creating candidates was effectively removed by the cuckoo filter architecture. Consequently, there is no longer a need for expensive evaluations because of the significant decrease in their computational demands.

3.3. CS-OPBL Illustrations

For instance, consider a scenario with three data partitions and a minimum support threshold of 3. The following phase first, with A, B, D, and F have been identified as frequent, with support values that satisfy or exceed the minimum support criteria. The cuckoo filter stores these transactions. The cuckoo filter prunes transactions with a length of at least 2 in each partition during the initial iteration of the second phase. Following this, the pruned transactions with a length of at least 2 generate combinations of 2 items. Every combination is converted into the format. We analyze every partition concurrently, using division 2 as an example (Figure 5(a)). Next, we employ the filter() function to eliminate combinations with a frequency less than 3, as illustrated in Figure 5(b). The output consists of two frequent item sets. Figure 5(b) shows that the number of 2-common item sets is greater than 1, which means that we need to keep going by raising k. We have changed the cuckoo filter to only store items from the two most common sets.

This is shown in Figure 5(c). We use the cuckoo filter in the second round (k = 3) to eliminate events in each section that have a length of three or more. As seen in Figure 5(d), the outcome for partition 2 is a single transaction with a duration of 3 seconds. No combinations of three items may be formed from a given transaction, and it will be eliminated if it is included in a certain partition and its Length adjustments before or after pruning in the cuckoo filteris less than 3. Figure 5(e) illustrates the calculation of the frequency of each combination following the application of the reduceByKey() function to all partitions. The filter() function subsequently eliminates any sets of three items that occur less than three times.

Algorithm 3.1: CS-OPBL For Enhancing Apriori Algorithm

```
Inputs:
Database: Transaction dataset
min_support: Minimum support threshold
Output:
1-frequent items: RDD of single-item frequent sets
For each entry in the processed Dataset
a. Perform_Mapping(offset_index, entry)
b. expand_Mapping(entry, extract_elements)
c. For each item in entry
i. Perform_Map(item, 1)
End For each
End expand_Mapping
End Perform_Mapping
End For each
End KeyAggregation
1-item_set =
filter(find_frequent_items(min_support_level))
items = 1-item_set.item_keys()
C_filter = CuckooHashTable(items)
shared_dataset = distribute(C_filter)
```

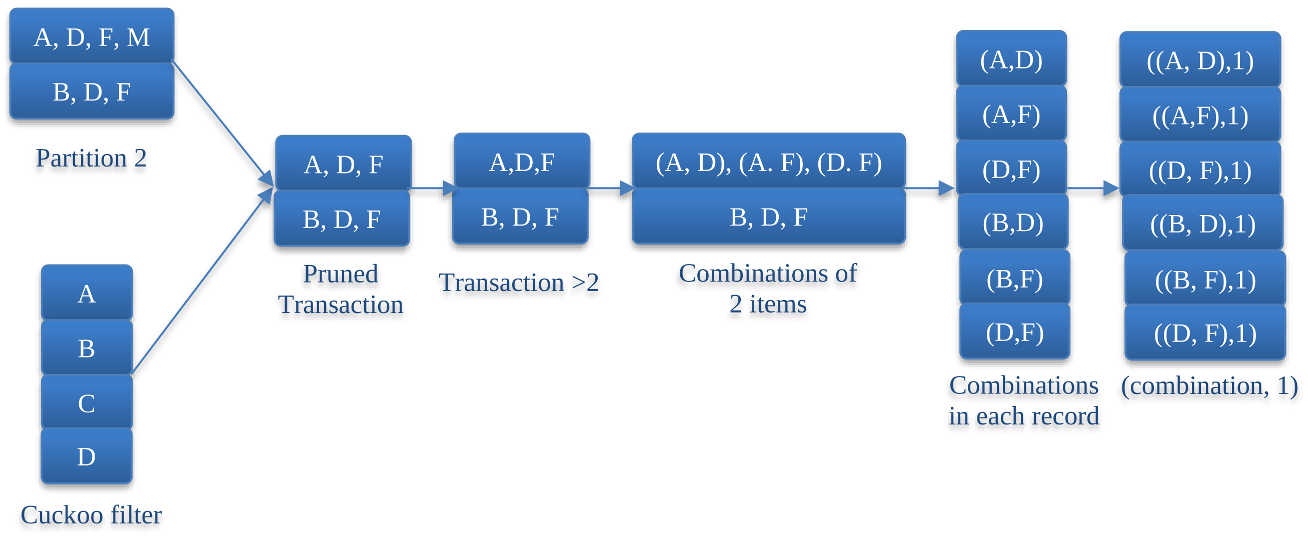

Fig. 5(a) Partition using division of 2

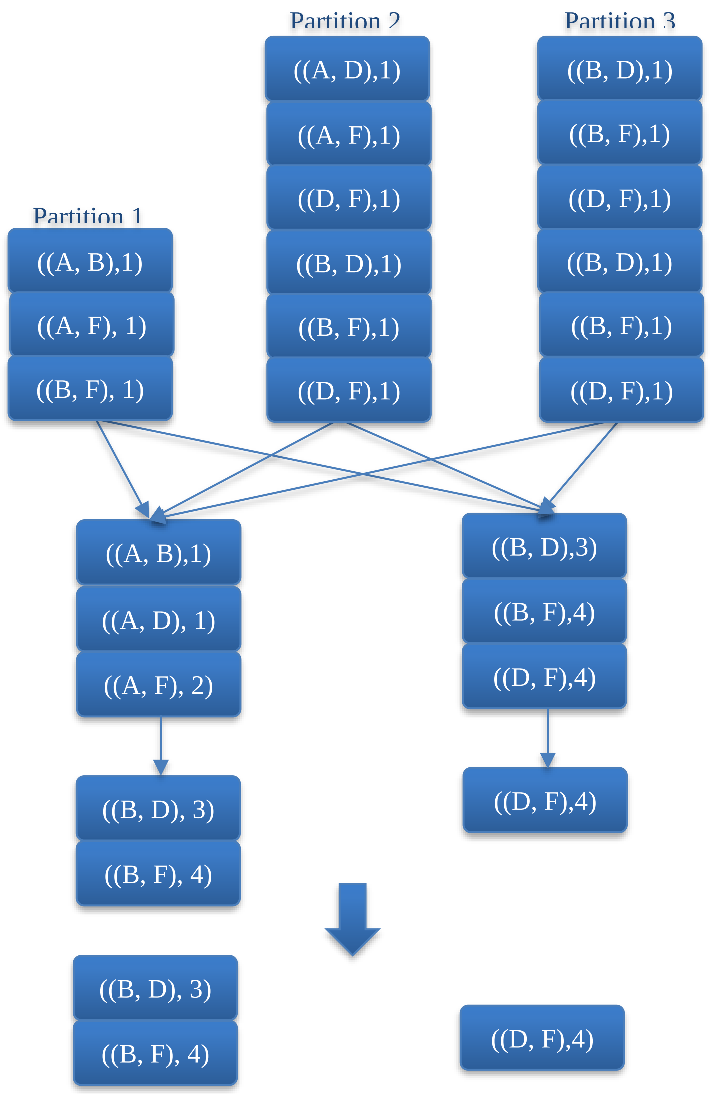

Fig. 5(b) Number of 2-common item sets is greater than 1

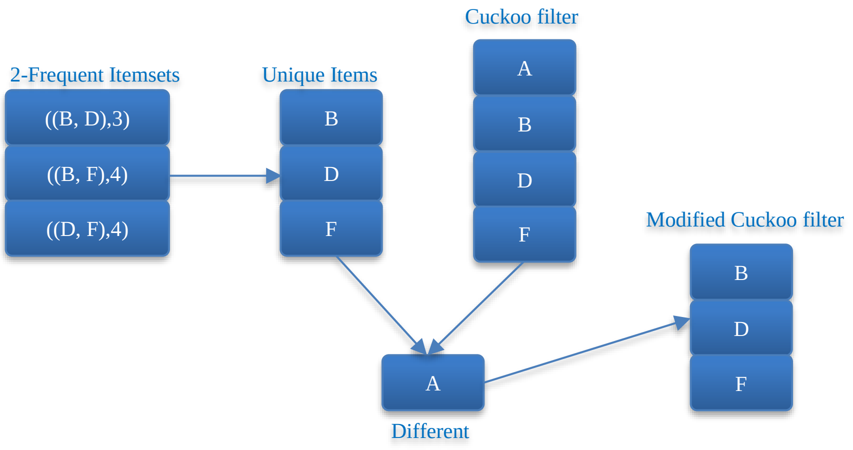

Fig. 5(c) Cukofilter store items from two most common sets

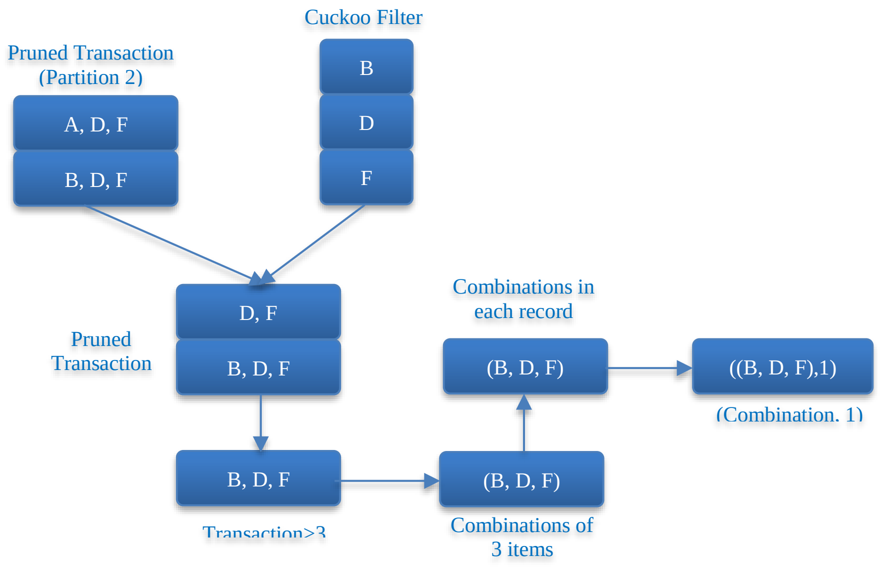

Fig. 5(d) Output of partition 2

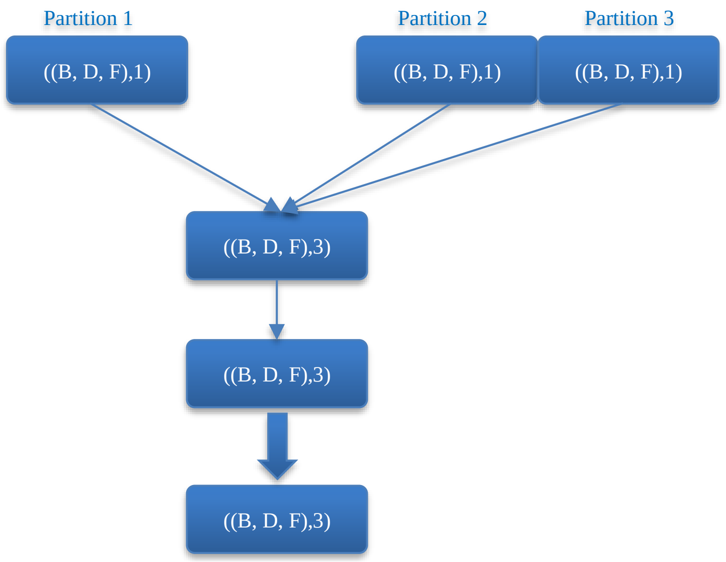

Fig. 5(e) The calculation of the frequency of each combination following the application of the reduceByKey() function to all partitions

4. Results and Discussions

This part evaluates the CS-OPBL efficiency. We compare CS-OPBL with three Spark-based algorithms—HFIM, YAFIM, and EAFIM. A Spark cluster, which CS-OPBL uses, consists of four nodes. With three CPU cores and six GB of RAM, every node can do its job. All of the nodes are running the latest versions of Hadoop, Spark, and Python.

4.1. Dataset Utilized

The CS-OPBL algorithm operates on three distinct datasets. IBM's information generator produced the first dataset, T10I4D100K. The second dataset, known as the commercial database, is made up of market basket information that records distinct transactions from a mall. The third dataset, known as the chess dataset, features end-game scenarios involving kings and rooks in chess. Key statistics for these datasets are detailed in Table 2.

4.2. Performance Assessment

In each second-phase repetition, the Apriori algorithm creates a huge set of candidates. It then compares these candidates to each transaction record to find the k-frequent item sets. This process takes the most time and room, especially for large files.

CS-OPBL works on the three datasets listed on a collection of four nodes. Each node has its own three CPU cores and 6 GB of RAM so that it can do its own thing. EAFIM works with groups of five nodes. There are 4 CPU cores and 16 GB of RAM for each node. The first iteration creates k-frequent item sets; the second iteration creates two frequent item sets, and so on.

Figure 7 shows the time it takes to run each version of CS-OPBL, HFIM, and YAFIM. In Figure 7(a), you can see how long the T10I4D100K dataset takes to run each time with 0.25% minimum support. CS-OPBL goes through nine steps to create eight frequent item sets. In every version, it does better than HFIM and YAFIM. In the chess dataset experiment, CS-OPBL requires 9 runs to identify 8 frequent item sets with a minimum support level of 85%. It also does better than HFIM and YAFIM, as seen in Figure 7(b). Figure 7(c) shows it does a better job. Table 3 shows how long it took for each method to run on all of the datasets.

We use the minimum support numbers mentioned above to test the performance of CS-OPBL on A group of three interconnected nodes representing all three datasets. The T10I4D100K dataset takes 72.2 seconds to run, with chess taking 28.8 seconds and retail taking 11.5 seconds. This means that CS-OPBL runs faster as the number of nodes grows. With at least 85% confidence, we compare the CS-OPBL running time to EAFIM on the chess dataset. The total time it took to run EAFIM was 70s. Figure 8 shows that CS-OPBL outperforms EAFIM in all trials.

Database	**Number of items**	**Number of transactions**
T10I4D1OOK	870	100000
Retail	16470	87988
Chess	75	3196

Fig. 6 Important dataset statistics

4.3. Discussion

The HFIM method begins by aggregating transaction IDs (TIDs) for each item, organizing data into (item, TID) pairs using the groupByKey() function, which is time-consuming due to the need for data movement across the network. HFIM also requires significant time to analyze vertical data and determine itemset frequencies. YAFIM, on the other hand, struggles with efficiency when faced with numerous candidate combinations, as it consumes substantial space and time to scan the transactional data on each node and store candidate itemsets in a hash tree for subsequent rounds. EAFIM attempts to optimize by identifying frequent items and updating the input database by removing redundant items and transactions. However, this introduces additional costs when reloading the revised input RDD for future iterations. OPCS surpasses HFIM, YAFIM, and EAFIM by leveraging a cuckoo filter to store frequent items, which are then used to prune transactions, allowing for the generation of candidate itemsets of size k or larger.

In contrast to Rathee et al., who improve efficiency using a Bloom filter for 2-frequent itemset creation—though at the cost of needing to rebuild the entire filter for data deletion and with lookup time depending on the number of hash functions—OPCS utilizes a cuckoo filter for direct deletion operations, achieving O(1) lookup performance with less than 3% space complexity. Moreover, while Rathee et al., like YAFIM, use a hash tree to store candidate (k+1)-itemsets and scan the entire dataset in each iteration, OPCS avoids this inefficiency by pruning transactions and retaining only those with lengths k+1 or less.

Dataset	**CS-OPBL (s)**	**HFIM (s)**	**YAFIM (s)**
T10I4D1OOK	59.8	158	200
Retail	27.8	105	137
Chess	12.5	197	225

Fig. 7 Total implementation period for CS- OPBL, HFIMs, and YAFIMs

4.4. Time Complexity

A dataset containing t items, n operations, and m components; the biggest transaction is represented by the letter D. First, we must analyze every interaction to find m rarely used things. Under worst-case conditions, this process may take O(n × m) time. The cuckoo filter is used to store up to ten items, with the insertion process averaging O(1) time due to the filter's known O(1) average insertion time [20, 23]. After completing this initial phase, we prepare for the second phase by pruning transactions to retain only those containing items with a frequency of at least k, where k ranges from 2 to n. We then update each transaction to include only items present in the cuckoo filter, which is the most time-consuming step, taking O(n) time.

Since Phase 2 is incremental, we will determine the time required to complete each ith cycle. To form groups of k items from transactions that have >= k, we first need to reduce each transaction. This reduction process will take no more than O(N) time. After this, the number of transactions will be reduced to N′, which is less than or equal to N. Each transaction will then produce a distinct set of k items. The potential number of item combinations within each transaction is O(N^k). If the operation includes items a, b, and c, the available permutations are O(N^k). To create k-frequent sets of items with the fewest possible assistance counts, these sets are then converted into (key, value) pairs. This operation will not take more than O(N′ × C) time if there are C potential pairs of k items per operation. We must confirm if the frequent objects in each of the many k-frequent sets of items are different when compared to those in the Cuckoo filter.

Next, we will change the cuckoo filter by getting rid of the items that are not in the k-frequent item sets. The cuckoo filter requires an O-time to delete and check values. We prune each transaction, k = k+1, to retain only those that exceed k. We remove all items that occur infrequently from each transaction, ensuring that the next repeat begins in no more than O(N′) time. In this case,O(N)+O($N^{\prime\wedge k}$)+O(N′×C)+O(F×k)+O(N′), says the time complexity of the Phase 2The next repetition will then begin. The amount of time needed for phase 2 will be O(N)+O(N′^k)+O(N′×C)+O(F×k). Yes, the total difficulty of CS-OPBL is equal to phase 1 plus the step of getting ready for Phases 1 and 2.

5. Conclusion

This work improves Apriori by adding a Cuckoo Search algorithm using opposition parameters-based learning. This method optimizes the original Apriori technique as the database's size or quantity of items increases. It consists of two parts. The initial stage is in charge of making 1-frequent item sets. Phase two involves the iterative creation of k-frequent item sets. We can summarize the CS-OPBL's efforts as follows:

- ➢ The cuckoo filter structure is used to prune transactions. The cuckoo filter trims each transaction to include only frequently used items. This decreases the quantity of items in each purchase.
- ➢ Every time, CS-OPBL cuts down on the total amount of transactions by getting rid of all transactions with a length of less than k, in which k represents the length of the regular item sets that will be made. Instead of making all possible candidate sets in each cycle, it makes candidate sets from each trimmed transaction whose length is atleast k. This cuts down on the number of potential sets.
- ➢ The algorithm saves the trimmed transaction information and the k-frequent sets of items, which may be employed to create k+1-frequent itemsets in the next cycle. This process notably accelerates subsequent operations.
- ➢ We put into practice the parallel-distributed Spark's technology. With several benefits, using in-memory processing. Because of these features, Spark outperforms other environments when it comes to iterative algorithms.
- ➢ When running on a cluster of four nodes, CS-OPBL, with minimum support of 0.75%, completes 5.8% of the HFIM on the commercial dataset. On the chess dataset, it only gets 25% of HFIM and 38.1% of EAFIM, with a minimum support of 85%. With a minimum support threshold of 0.25% on the T10I4D100K dataset, the HFIM algorithm achieves only 3.3%Testing with various datasets confirms that CS-OPBL enhances the Apriori algorithm's performance.

The findings indicate that CS-OPBL consistently surpasses both HFIM and YAFIM across all datasets, regardless of the minimum support levels. By substituting the candidate selection process with a method for choosing the optimal candidates in each iteration, CS-OPBL streamlines the computations. Applications of CS-OPBL include market basket analysis, class selection on e-learning sites, and stock management. Future research in biology will utilize CS-OPBL, as the majority of tasks in this field require identifying frequently connected elements. Additionally, these applications are constantly faced with a vast array of features. It will also be tested against additional algorithms for mining group regulations, such as FP-growth and Buddy Prima.

References

[1] Made Leo Radhitya et al., "Product Layout Analysis Based on Consumer Purchasing Patterns Using Apriori Algorithm," *Journal of Computer Networks, Architecture and High-Performance Computing*, vol. 6, no. 3, pp. 1701-1711, 2024. [CrossRef] [Google Scholar] [Publisher Link]

[2] D. Padmini Bai, and P. Preethi, "Security Enhancement of Health Information Exchange Based on Cloud Computing System," *International Journal of Scientific Engineering and Research*, vol. 4, no. 10, pp. 79-82, 2016. [Google Scholar] [Publisher Link]

[3] M. Supriyamenon, and P. Rajeswari "A Review on Association Rule Mining Techniques with Respect to their Privacy Preserving Capabilities," *International Journal of Applied Engineering Research*, vol. 12, no. 24, pp. 15484-5488, 2017. [Google Scholar] [Publisher Link]

[4] P. Preethi, and R. Asokan, “Modelling LSUTE: PKE Schemes for Safeguarding Electronic Healthcare Records Over Cloud Communication Environment,” *Wireless Personal Communications*, vol. 117, pp. 2695-2711, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[5] R. Agrawal, and J.C. Shafer, “Parallel Mining of Association Rules,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 6, pp. 962–969, 1996. [CrossRef] [Google Scholar] [Publisher Link]

[6] Ning Li et al., “Parallel Implementation of Apriori Algorithm Based on MapReduce,” *International Journal of Networked and Distributed Computing*, vol. 1, pp. 89-96, 2013. [CrossRef] [Google Scholar] [Publisher Link]

[7] Brijendra Singh, and Rohit Miri, “An Efficient Parallel Association Rule Mining Algorithm Based on MapReduce Framework,” *International Journal of Engineering Research*, vol. 5, no. 6, pp. 236–240, 2016. [Google Scholar] [Publisher Link]

[8] Hongjian Qiu et al., “YAFIM: A Parallel Frequent Itemset Mining Algorithm with Spark,” *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, Phoenix, AZ, USA, pp. 1664–1671, 2014. [CrossRef] [Google Scholar] [Publisher Link]

[9] Sanjay Rathee, Manohar Kaul, and Arti Kashyap, “R-Apriori: An Efficient Apriori Based Algorithm on Spark,” *Proceedings of the 8th Workshop on Ph.D. Workshop in Information and Knowledge Management*, New York, NY, USA, pp. 27–34, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[10] Shashi Raj et al., “EAFIM: Efficient Apriori-Based Frequent Itemset Mining Algorithm on Spark for Big Transactional Data,” *Knowledge and Information Systems*, vol. 62, pp. 3565–3583, 2020. [CrossRef] [Google Scholar] [Publisher Link]

[11] Krishan Kumar Sethi, and Dharavath Ramesh, “HFIM: A Spark-Based Hybrid Frequent Itemset Mining Algorithm for Big Data Processing,” *The Journal of Supercomputing*, vol. 73, pp. 3652–3668, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[12] Sanjay Rathee, and Arti Kashyap, “Adaptive-Miner: An efficient Distributed Association Rule Mining Algorithm on Spark,” *Journal of Big Data*, vol. 5, pp. 1–17, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[13] Fei Gao, Ashutosh Khandelwal, and Jiangjiang Liu, “Mining Frequent Itemsets Using Improved Apriori on Spark,” *Proceedings of the 2019 3rd International Conference on Information System and Data Mining*, Houston, TX, USA, pp. 87–91, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[14] Eduardo P.S. Castro et al., “Review and Comparison of Apriori Algorithm Implementations on Hadoop-MapReduce and Spark,” *The Knowledge Engineering Review*, vol. 33, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[15] Shashi Raj, Dharavath Ramesh, and Krishan Kumar Sethi, “A Spark-Based Apriori algorithm with Reduced Shuffle Overhead,” *The Journal of Supercomputing*, vol. 77, pp. 133–151, 2021. [CrossRef] [Google Scholar] [Publisher Link]

[16] Sunil Kumar, and Krishna Kumar Mohbey, “A Utility-Based Distributed Pattern Mining Algorithm with Reduced Shuffle Overhead,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 1, pp. 416–428, 2023. [CrossRef] [Google Scholar] [Publisher Link]

[17] Sunil Kumar, and Krishna Kumar Mohbey, “A Review on Big Data Based Parallel and Distributed Approaches of Pattern Mining,” *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 5, pp. 1639–1662, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[18] Mohamed A. Gawwad, Mona F. Ahmed, and Magda B. Fayek, “Frequent Itemset Mining for Big Data Using Greatest Common Divisor Technique,” *Data Science Journal*, vol. 16, pp. 1-10, 2017. [CrossRef] [Google Scholar] [Publisher Link]

[19] Sunil Kumar, and Krishna Kumar Mohbey, “UBDM: Utility-Based Potential Pattern Mining Over Uncertain Data Using Spark Framework,” *5th International Conference, Emerging Technologies in Computer Engineering: Cognitive Computing and Intelligent IoT*, Jaipur, India, pp. 623–631, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[20] Sunil Kumar, and Krishna Kumar Mohbey, “Memory-Optimized Distributed Utility Mining for Big Data,” *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 8, pp. 6491– 6503, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[21] Subramanian Kannimuthu, and Kandhasamy Premalatha, “Stellar Mass Black Hole Optimisation for Utility Mining,” *International Journal of Data Analysis Techniques and Strategies*, vol. 11, no. 3, pp. 222–245, 2019. [CrossRef] [Google Scholar] [Publisher Link]

[22] S. Kannimuthu, and D. Gowtham Chakravarthy, “Discovery of Interesting Itemsets for Web Service Composition Using Hybrid Genetic Algorithm,” *Neural Processing Letters*, vol. 54, no. 5, pp. 3913–3939, 2022. [CrossRef] [Google Scholar] [Publisher Link]

[23] Kannimuthu Subramanian, and Premalatha Kandhasamy, “UP-GNIV: An Expeditious High Utility Pattern Mining Algorithm for Itemsets with Negative Utility Values,” *International Journal of Information Technology and Management*, vol. 14, no. 1, pp. 26–42, 2015. [CrossRef] [Google Scholar] [Publisher Link]

[24] Le Hoang Son et al., “ARM–AMO: An Efficient Association Rule Mining Algorithm Based on Animal Migration Optimization,” *Knowledge-Based Systems*, vol. 154, pp. 68–80, 2018. [CrossRef] [Google Scholar] [Publisher Link]

[25] Aghila Rajagopal et al., “A Novel Approach in Prediction of Crop Production Using Recurrent Cuckoo Search Optimization Neural Networks,” *Applied Sciences*, vol. 11, no. 21, pp. 1-13, 2021. [CrossRef] [Google Scholar] [Publisher Link]