Chapter 7

Concept Mining Model for Distributed Environment

Although the proposed method in chapter 6 has satisfying progress on dynamic environment, it has not been equipped to challenge problems involving big data sets with small file size or requiring huge computation [77] in a distributed environment. Due to this, particles of the multi swarm cooperative FPSO may fly discretely in the dynamic environment, and there might exists the chance to miss the narrow area where the global optimum escapes due to chances of decentralization of huge amounts of data. And as the problem is getting complex, the computational cost increases. So we implement the “MapReduce enabled parallel frequent pattern based fuzzy particle swarm optimization (MRPFPSO)” [107] in parallel and distributed environment by extending the algorithm on MapReduce model. By embracing the power of parallel computing, we can generate optimal cluster centroids and obtain faster convergence with MapReduce.

7.1 Related Work

Typically, conventional traditional frequent pattern mining algorithms works in two dimensions mainly the breadth-first techniques like Apriori [7] and the depth first techniques like FP-growth [47, 48]. To further enhance the efficiency in mining, some optimized methods are resorted to improve the aforementioned algorithms [46, 85, 138]. Furthermore, to overcome the issues of mining on a single machine, several parallel or distributed algorithms are presented in [152, 135, 144]. However, by using the traditional methodologies especially sequential algorithms for mining large scale data in a stand-alone environment involves issues of memory and I/O cost. These issues degrades the performance of the sequential algorithms in comparison to parallel counter parts. Various works on distributed document clustering have been discussed in the related works about shifting the computation to the location of the large data rather than moving huge data to the central node. From various studies by researchers we can identify that distributed clustering faces challenges mainly from two aspects: algorithmic issues and implementation issues. Authors in [27] introduced a P2P K-means algorithm for dynamic synchronization of nodes that are connected to that node directly. The algorithm is seen to fail in situations of dynamic peer distribution. While the authors in [111] uses a probabilistic approach to increase scalability by using distributed hash table for clustering documents but fails in aspects of speed up with increase in data sets. Thangamani.et.al [63, 136]
proposed a fuzzy semantic clustering scheme for decentralized network using ontologies. But this method too has to face scalability issues. From the literature study done we can conclude that the distributed clustering algorithms faces challenges of scalability, speedup and distribution of input data to deal with future needs of information retrieval. To meet these odds Hadoop MapReduce can be used to significantly improve the performance of the algorithm. Recently MapReduce has gained significance especially in data mining domains. In recent years to meet the rapidly growing demands of large scale data processing, a Parallel FP-growth (PFP) algorithm [152, 154] based on MapReduce was proposed by authors in [83]. By incorporating load balancing feature by authors in [163], the performance of parallel frequent pattern approach was enhanced.

Clustering finds new scopes in the era of knowledge economy and gained access to numerous applications where the amount of data increase is a challenge, from profiling pages in social network analysis, search engine optimization to bio-informatics. With expensive storage constraints and time complexities being the limitation for sequential clustering, we discuss in this chapter work related to parallel clustering algorithms that deploy MapReduce system [161, 82, 109, 37, 44, 70, 61, 26].

Zaho et al. in [161, 11, 10] modeled a K-Means parallel algorithm based on MapReduce. In this \( k \) initial centroids are distributed randomly and uses Map function to ultimately find the weighted average of the items within clusters and uses Reduce function to update the cluster centroid positions as they move across search space.

Li et al. in [82, 11, 10] uses the ensemble learning method with MapReduce K-means clustering algorithm to solve the outliers. The MapReduce framework for co-clustering large data set was introduced by Sur et.al [109] and discussed the diverse application of co-clustering mining for collaborative filtering as well as mining of the text [11].

The data sampling and constant factor approximation was introduced by authors [37, 10] for shrinking data. Shrinkage of data helped in faster clustering but the algorithm gave a minimal performance with respect to k-median problem [11]. An efficient sub surface clustering, “Best of Both Worlds” to reduce I/O and network cost of clusering in Hadoop framework was discussed by [44, 10].Particle swarm optimization using MapReduce is proposed by authors in [11]. Here the authors finds a scalable solution with increased data size.

To the best of my knowledge, none of the algorithms considered the hybrid use of MapReduce, and “Frequent Pattern Growth and Fuzzy Particle Swarm Optimization” [107, 108] to improve the clustering algorithm in distributed environment. In this chapter clustering is defined as an optimization task to find the best cluster solutions in a distributed environment based on the minimum distance between documents and its corresponding relevant clusters.

7.2 Preliminaries

Most of the sequential clustering algorithm suffer from the problem of scalability, expensive memory storage and time complexities when clustering large data sets. For these reasons parallelization of clustering algorithm is essential for big data to make the system scalable with large number of clusters within an acceptable amount of time. The approach discussed in chapter 6 fares well in dynamic environment. In this the amount of clusters generated is proportional to the frequent sets retrieved from frequent pattern growth algorithm. So to scale up the algorithm to big data, in this chapter we discuss the use of
MapReduce technique to enhance the clustering algorithm.

7.2.1 Hadoop

Hadoop is the most commonly used implementation of the MapReduce technique [10, 13, 28]. Google faced the problem of scaling with massive volumes of data when they wanted to perform web indexing, log analysis and such other operations on larger datasets. For this Google devised a framework for large-scale data processing deriving from the “map” and “reduce” functions of the functional programming paradigm. Apache Hadoop [10, 13] is the commonly used open source MapReduce implementation, for data-intensive distributed applications licensed under Apache. The two main component of Apache Hadoop infrastructure that enables processing of yottabytes of data includes: Hadoop Distributed File System (HDFS) and MapReduce. HDFS does perform two functions: creates replicas of the target blocks to make the system fault tolerant. Second function provides a high-throughput access to the data. Next major component is the processing factor, MapReduce enabling the computation at the location of the data rather than taking the data to the centralized location [10, 11].

7.2.2 MapReduce

In normal parlance every disk access for any change in content makes expensive calls to disk and increases the cost and disk throughput. So by using MapReduce we move the computation to the location of documents and process data sequentially [10, 11]. The use of MapReduce very much speeds up the proposed algorithm. In MapReduce, all data are in the form of keys, $K^*$ with its corresponding values, $V^*$. In the word count problem for example we consider words as key and frequency as associated values. This is done using two function: Map and Reduce. The Map operation is called for each input record and outputs any number of output records. Since the Map function takes records one by one for each call, all operations are independent and fully parallelized. The Map function is defined as follows:

$$map : (K_1^*, V_1^*) \rightarrow list(K_2^*, V_2^*)$$  \hspace{1cm} (7.1)

In the second stage, these outputs is sorted and grouped by key and finally Reduce operation is applied for each key and outputs a new list of reduced set of values. Even though Reduce operation are independent in calling its function but it can begin only when all Map operations are completed.

$$reduce : (K_2^*, list(V_2^*)) \rightarrow list(V_2^*)$$ \hspace{1cm} (7.2)

7.3 Proposed Methodology

In this section, we propose a “MapReduce enabled Parallel Frequent Pattern based Fuzzy Particle Swarm Optimization clustering” [107, 108] algorithm to implement the parallelism of association rule mining in big dataset with massive small files on a Hadoop...
platform by incorporating massive small file processing strategies. The use of MapReduce paradigm for mining frequent item sets in parallel is incorporated to overcome the inherent defects of Hadoop in handling big data sets associated with large-scale short texts, thereby generating fuzzy clusters matching user needs. In this chapter focus is on finding best cluster solutions for the massive short texts documents spread across in distributed environment using the new method discussed in this chapter. We have formulated clustering as an optimization problem trying to minimize the distance between data items and its cluster centroids.

Overview of the proposed approach is show in Figure 7.1 below:

![Proposed Approach](image)

**MapReduce enabled Parallel Frequent Pattern Growth based Fuzzy Particle Swarm Optimization**

The massive small file processing strategy is incorporated into MRFPFPSO which is implemented with the MapReduce paradigm for mining frequent item sets in parallel to overcome the inherent defects of Hadoop in handling big datasets associated with large-scale small files, thereby generating interestingly strong association rules. On the other hand, Hadoop was originally designed to cope with large streaming files and thus stores and manages massive small files inefficiently. Memory consumption and access cost increases in cases where huge amount of small chunks of data are stored in HDFS [10, 31]. Moreover the computational efficiency of Hadoop depends on the performance of two key components (i.e., HDFS and MapReduce) [163]. For these reasons, the processing of massive small file will reduce the overall performance of association rule mining on a Hadoop platform, which is mainly shown in the following two aspects:

- The access efficiency of HDFS is reduced. Namenode is responsible for manag-
Concept Mining Model for Distributed Environment

ing and scheduling huge amounts of metadata stored in each Datanode node, and each file, directory, and block is encapsulated into a 150-byte space stored in the memory of Namenode, which needs to query and retrieve the requested file blocks between Datanodes with lots of searches and hops. To this end, massive small files will occupy a large amount of Namenode limited available memory space and consume a lot of execution time, which will result in inefficient data access of HDFS significantly. Storing and managing massive small files pose a big challenge to HDFS.

- MapReduce jobs consist of multiple Map tasks and Reduce tasks, and the Map tasks commonly execute an input block at a time. If the file is too small in size and numerous in number, a large number of Map tasks will be generated and each Map task will only handle very small data, which will lead to an additional increase of MapReduce computational tasks inevitably. For example, a 2GB file is equally divided into 32 files (i.e., each file with a size of 64MB) or $2^{21}$ files (i.e., each file with a size of 1 KB). We can draw a conclusion that the execution time of the latter will be significantly longer than the former by comparative analysis. The reason is that the former requires only 32 Map tasks to complete the data processing task, while the latter needs $2^{21}$ Map tasks to operate.

The proposed MapReduce methodology provides the ability to distribute the documents to one or more storage spaces and perform the computation at its locations. The proposed algorithm consists of three stages: firstly, combining all small chunks of big data into single sequence file for HDFS to efficiently utilize the data. Secondly, particle centroid updation using MapReduce enabled parallel frequent growth approach. Third stage includes fitness evaluations using MapReduce enabled fuzzy particle swarm optimization. The Hadoop Distributed File System (HDFS) stores document vectors in the form of transaction database as discussed below.

To process massive small files of size less than 64MB we have incorporated three methods: Hadoop Archives (HAR), Sequence Files (SF), and CombineFileInputFormat (CFIF) [152].

### 7.3.1 Stage 1

**Hadoop Archives**

Every Hadoop Archive has a *.har extension, which can be used as an input to MapReduce. In stage 2, all local frequent itemsets and global frequent itemsets obtained are stored in Hadoop Archives using archive command [152, 143]. In the HAR file system, the process of reading files happens in three stages:

1. Access the Master Index to gain the Index stored in memory index
2. Access the Index to get the Store Index of the working file.
3. Access the Store Index to obtain the content of the working file.
Sequence Files

Sequence File (SF) is a method that integrates massive small files into a large sequence file via the following three classes [95].

1. WholeFileInputFormat Class: the `isSplitable()` method within the class overloads and returns the value, “false”, to maintain the input file not to be partitioned into splits. The `getRecordReader()` method returns a customized RecordReader.

2. WholeFileRecordReader Class: the `FileSplit` is converted into a record, where the key of the record is the document name/file name and the value is the content of the file. In view of the existence of only one record after the conversion, WholeFileRecordReader only deals with the record or is discarded. However, it uses a Boolean variable, `processed` to indicate whether the record was processed. If the `next()` method is called, it indicates that the file is not processed. At the same time the file will be opened and then generate a byte array with a size of the file length. Subsequently, it calls the Hadoop’s `IOUtils` class and takes the content of the file into the byte array. Finally an array is created on the `BytesWritable` instance that is passed to the `next()` method. If the return value is true, it demonstrates that the record has been successfully read [152].

3. SmallFilesToSequenceFileConverter Class: massive small files are integrated into a sequence file, and this class consists of the `Map()` and the `Reduce()`. The input format of data is `WholeFileInputFormat`, while the output format is `SequenceFileOutputFormat`.

CombineFileInputFormat

CombineFileInputFormat (CFIF) [152] is a new type based on the `InputFormat` method which packages multiple files into a split to handle more data by each `Map()`, thereby avoiding the drawbacks of traditional `FileInputFormat` that must generate a split for each file. Unlike the SF method, the CFIF method does not integrate massive small files but only produces less `InputSplit`, to achieve the packaging of massive small files through employing the following two classes.

1. CombineFileLineRecordReader Class: packaging multiple files then will be processed by the Map tasks.

2. MyMultiFileInputFormat Class: the `CreateRecordReader()` method is implemented and the abstract class of CFIF is inherited, and then the implementation of the customized Recorder is returned.

As a result of these processing strategies we could finally aggregate all small files stored in HDFS into one transaction database, ready for input to “MapReduce enabled frequent pattern growth approach based fuzzy particle swarm optimization” [107, 108].

---

2 The details of three classes are reprinted from the excerpts of works published by authors in [95]
7.3.2 Stage 2

In stage two, we focus our goal on dividing the transaction database obtained as discussed above into many sub transaction databases. These are then assigned to different nodes of Hadoop cluster. We employ the *balance* command to enable its file system to fulfill load balancing, when necessary. The Map function divides the transaction database into equal chunks of 64MB each. Frequent pattern growth based Map function stores the data objects with its co-occurrent terms in frequent pattern tree as discussed in chapter 3. For each node many tree data structures are evolved for the document sets stored in each node. Frequent growth algorithm generates all conditional patterns matching to user query from the tree by setting the threshold for dimensional reduction to Mean Squared Residual Error (MSRE) from the tree data structure. The reduce operation accepts the outputs from map function and then calculates the centroids as the mean of the frequent items under each frequent item sets as discussed in chapter 4. Later using MapReduce operation we sum up all the local frequent item sets under each frequent set on a Hadoop cluster. This operation ends giving the set of global frequent items which will act as master and slave inputs for the map reduce enabled multi swarm adaptive fuzzy particle swarm optimization. These optimal centroids obtained are further reduced using Random Indexing.

7.3.3 Stage 3

In this stage main focus is on particle centroid updation and fitness evaluation. This is done using two phases. The tentative particle centroids obtained from stage two are updated based on the swarm particles as they move around in a dynamic and distributed environment in this stage. In traditional centralized approach the particle centroid updation takes long time with larger data sets with smaller files. In third phase the fitness function evaluates the position of the particle and assigns a fitness value as discussed in chapter 4.

**Phase 1: Particle Swarm Centroid Updation**

The Map function receives the particle swarm or document terms with Freq.setID’s and dependent list of particles containing information of all neighbors or co-occurant terms as Value from MapReduce enabled frequent pattern growth approach. Since the swarms obtained fails to communicate among particles, we have opted MapReduce operation. While in the Reduce operation it facilitates global best by taking information from different swarms. The proposed map reduce enabled fuzzy particle swarm optimization clustering conforms to the MapReduce model while performing the same operations and calculations as discussed in chapter 4. Every time a message is sent from one particle to another as part of MapReduce operation, in the Reduce operation the recipient reads the individual best and updates its global best accordingly as seen in Algorithm 2.

**Phase 2: Fitness Function Evaluation of Particles**

In this second phase of stage three, MapReduce is relaunched to update the fitness values of the updated swarms from phase one. The Map function starts with taking the particles as RecordID, extracting centroid vector information from the distributed cache and then
Algorithm 2 Phase 1: Particle Swarm Centroid Updation

function MAP(Key : Freq.setID, Value : FreqitemSets) ▷ Key and Value obtained from MapReduce enabled FP Growth

Initialization :
Freq.setID = Key
FreqitemSets = Value
extractInfo(Cluster Centroid, X, Y, p_best, s_best)

6: for each X_i in cluster centroid do ▷ Update particle velocity
for each j in Dimension do
\[ V(ij + 1) = w \odot V(ij) \oplus c_1 r_1 (i_{best}(ij) \ominus X(ij)) \oplus c_2 r_2 (s_{best}(ij) \ominus X(ij)) \]
\[ X(ij + 1) = X(ij) \oplus V(ij + 1) \]
end for
update (particle, V(ij + 1), X(ij + 1))
end for
Output (Freq.setID, FreqitemSets)
end function

function REDUCE(Key: Freq.setID, DepList:FreqitemSets)
for each Value in DepList do
Output(Key, Value)
end for
end function

computes fitness value which is set to distance computation between centroid vector and the records. The Reduce operation assigns the values with same key to calculate the membership degrees of particles from the position matrix and formulates new fitness values. Finally output the relevant clusterID with particles membership degrees as Values.

7.4 Performance Evaluation

7.4.1 Experimental Setup

For this experiment we ran the experiment on Hadoop platform of 1 Master and 8 Slaves with Intel Xeon (R) E7-4820V4 2.00GHz CPU (8-cores) and 16.00 GB RAM. All experiments are conducted on Cent OS 6.5 with Hadoop version 2.7.3 for the MapReduce framework, and JDK 1.6.0.

7.4.2 Data Sets

Three real world data sets namely Frequent Itemset Mining Dataset Repository (FIMD) denoted by F1, F2, F3; BBC News data sets namely N1, N2, N3; [152] are chosen along with 3 synthetic data sets, T1, T2, T3 respectively. Each of them containing large number of small files as shown in Table 7.1.
7.4.3 Evaluation of the Proposed Methodology

We have evaluated the efficiency and scalability of the proposed methodology against the existing Parallel Frequent Pattern Growth approach in terms of speedup, sizeup and scaleup [114, 152, 155].

Efficiency

The efficiency of the proposed methodology outperforms the parallel frequent growth approach with increasing nodes as well as increasing size of the dataset as shown in Figure 7.2 to 7.4. Both real and synthetic data sets are used for the efficiency evaluation and the results show that for every increase in the number of nodes the execution time is relatively decreasing linearly in comparison to the state of the art techniques used for evaluation.

Table 7.1: Data Sets Used

<table>
<thead>
<tr>
<th>Data sets</th>
<th>Number of small files</th>
<th>Size of each small file</th>
<th>Size of all files</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBC News (N1)</td>
<td>843</td>
<td>= 1.1 MB</td>
<td>1GB</td>
</tr>
<tr>
<td>BBC Entertainment (N2)</td>
<td>1961</td>
<td>= 1.1 MB</td>
<td>3GB</td>
</tr>
<tr>
<td>BBC Sports (N3)</td>
<td>3988</td>
<td>= 1.1 MB</td>
<td>5GB</td>
</tr>
<tr>
<td>FIMD (F1)</td>
<td>2233</td>
<td>&lt; 64 MB</td>
<td>128 MB</td>
</tr>
<tr>
<td>FIMD (F2)</td>
<td>4566</td>
<td>&lt; 64 KB</td>
<td>256 MB</td>
</tr>
<tr>
<td>FIMD (F3)</td>
<td>8766</td>
<td>&lt; 64 KB</td>
<td>512 MB</td>
</tr>
<tr>
<td>Synthetic (T1)</td>
<td>856</td>
<td>&lt; 1.1 MB</td>
<td>1GB</td>
</tr>
<tr>
<td>Synthetic (T2)</td>
<td>1954</td>
<td>&lt; 1.1 MB</td>
<td>3GB</td>
</tr>
<tr>
<td>Synthetic (T3)</td>
<td>3945</td>
<td>&lt; 1.1 MB</td>
<td>5GB</td>
</tr>
</tbody>
</table>

Figure 7.2: Execution Time Comparison for Proposed Methodology and PFP with Synthetic Data Sets
7. Concept Mining Model for Distributed Environment

Figure 7.3: Execution Time Comparison for Proposed Methodology and PFP with BBC Data Sets

Figure 7.4: Execution Time Comparison for Proposed Methodology and PFP with FIMD Data Sets

**Speedup**

This measure indicates how much faster the parallel algorithm, $T_{pr}$, performs on $p$ nodes in comparison to its sequential counterpart $T_{sq}$ on single node. The speedup is calculated as below:

$$\text{Speedup} = \frac{T_{sq}}{T_{pr}}$$  \hspace{1cm} (7.3)

The speedup metric is evaluated on a varying range of cluster nodes from 1 to 10 with 6 data sets whose size ranges from 128 MB to 5 GB as shown in Figure 7.5. From the Figure 7.5 its evident that the proposed methodology linearly increases with increase in the nodes as well as with increase in the size of the data. When the number of nodes reaches 10, the speedup of the $N1$ data set reaches 8.00 (i.e., 80 % of the ideal speedup). The visualization of the results of the comparison for speed up are shown in Figure 7.5.
Scaleup

Scaleup metric [152] gauges the parallel execution time of the algorithms when they have larger data sets with more number of nodes. \( T_s \) indicates the sequential execution time of the algorithm with the given data set on one node. While \( T_p^* \) denotes the parallel execution time of the algorithm for processing \( p \)-times data sets on \( p \)-times larger data sets. Scaleup is given by:

\[
\text{Scaleup} = \frac{T_s}{T_p^*}
\] (7.4)

To validate speedup we increased the nodes in proportion to the size of the datasets as shown in Figure 7.6.

Sizeup

The sizeup validates how much further the parallel algorithm extends on a given node when the size of the data sets is varied. Here we have varied the size from 128 MB to 10

---

Figure 7.5: Speedup Comparison for Proposed Methodology

Figure 7.6: Scaleup Comparison for Proposed Methodology

This results indicates that proposed methodology scales well with growing data sets.
GB on 10 nodes as shown in Figure 7.7. Sizeup is given by:

\[ Sizeup = \frac{T^*}{T_0} \]  

\[ T^* \] is the execution time of the algorithm with a given size of the data set on a given node while \( T_0 \) gives the processing time of \( p \times \) times larger dataset on a constant node as chosen for \( T^* \) [152]. Here only the data sets size varies while node remains constant.

![Figure 7.7: Sizeup Comparison for Proposed Methodology](image)

### 7.5 Result Analysis

Based on the evaluation conducted it could be identified that the proposed methodology performs better in a distributed and parallel environment more efficiently and effectively on a MapReduce framework. The massive small size file processing strategies incorporated to the base model resolves the common issues of memory consumption and I/O cost faced by Hadoop in handling large scale datasets with massive small files. The following are the main achievements of the new methodology:

#### 7.5.1 Memory Consumption Reduced

For evaluation we have used number of files and indexes in HDFS as discussed by authors in [152, 149]. We calculated the memory consumption of Namenode after processing the BBC Data sets with HAR, SF and CFIF as discussed in section 7.3.1 respectively.

![Figure 7.8: Memory Consumption](image)
From Figure 7.8 we could find that while using HAR method, the memory space consumed by the Namenode varies with two indexes, Master Index and Slave Index. In the case of SF method where small files are aggregated to large sequence file, the memory space consumed by the Namenode depends upon the index of the Index stored in Namenode. Lastly in the case of CFIF method massive small files are integrated into a InputSplit package and as a result of this the memory consumption is dependent on the number of files.

7.5.2 Execution Time Reduced

In HAR method since the memory consumption is reduced, the loading time is proportionately reduced and because of these reasons execution time decreases considerably. In the case of SF method, the execution time is proportional to the scheduling time where all files are integrated to make it a single file and thus increases execution time. While in the case of CFIF method packages the whole small files into single InputSplit used by a single Map function. This reduces the execution time.

7.6 Summary

In this chapter we have proposed “MapReduce enabled parallel frequent pattern growth based fuzzy particle swarm optimization clustering” [107] using massive small file processing strategies. The algorithm was applied to study the effectiveness of clustering in text documents with massive small size files on a Hadoop framework. With the extensive experiments with real world data sets the performance of the proposed methodology in terms of efficiency and scalability fares well to existing parallel frequent growth approach [152]. The inherent defects of Hadoop to manage massive small files has been overcome with reduced memory consumption and reduced I/O cost. The experimental results also shows that the proposed methodology enhances the performances in terms of efficiency, scaleup, sizeup and speedup.