CHAPTER 5

MAPREDUCE FOR FUNCTIONAL DEPENDENCY EXTRACTION

5.1 INTRODUCTION

Over the last few years, large volume of data is generated at a faster rate. There has been a remarkable growth in the need for large scale data processing systems. As the data grows larger in size, the data quality is many times compromised. Functional dependencies representing semantic constraints in data are important for data quality assessment. Executing functional dependency discovery algorithms on a single computer is hard and laborious with large data sets. One approach to work with such huge amounts of data is to rely on a parallel database system. This approach, broadly considered for decades, includes well-known techniques developed and enhanced over time. Parallel database systems feature sophisticated query optimizers, and a rich runtime setting that supports efficient query execution and at the same time, they run only on expensive high-end servers. When the data volumes to be stored and processed reach a point where clusters of hundreds or thousands of machines are required, parallel database solutions become prohibitively expensive (Abouzeid., et.al, 2009). Still, worse part of it is that, at such scale, many of the primary assumptions of parallel database systems (e.g., fault tolerance) begin to fail, and the conventional solutions are no longer feasible without considerable extensions.
MapReduce provides an enabling technology for large scale data processing. The open-source Hadoop implementation of MapReduce has provided researchers a powerful tool for tackling large-data problems in a distributed manner. The objective of this study is to extract functional dependencies between attributes from large datasets using MapReduce programming model. The Information Theory measure called entropy is used to measure the inter attribute correlations, and exploited to discover functional dependencies hidden in data.

The remaining part of the chapter is organized as follows. Section 5.2, Section 5.3, and Section 5.4 describe the theoretical background required for understanding the proposed work. The proposed approach is explained in Section 5.5. Experimental results are discussed in Section 5.6.

5.2 THE MAPREDUCE PROGRAMMING MODEL

A MapReduce program is composed of two user-specified functions, map and reduce. The core idea is that many types of tasks that are computationally intensive involve doing a map operation with a simple function over each record in a large dataset and emitting key/value pairs as results. The map operation itself can be easily distributed by running it on different machines, each processing their own subset of the input data. The output from each of these map operations is then collected and merged into the desired results by reduce operations. MapReduce specifies effective methods for processing large amounts of data by distributing work tasks over multiple processing machines in a shared nothing cluster. A large part of the power of MapReduce comes from its simplicity. In addition to preparing the input data, the programmer needs only to implement the mapper, the reducer, and optionally, the combiner and the partitioner. All other aspects of execution are handled transparently by the execution framework on clusters ranging from a single node to a few thousand nodes, over datasets ranging
from gigabytes to terabytes. The map and reduce functions in Hadoop MapReduce have the following general form:

\[
\text{map: } (K_1, V_1) \rightarrow \text{list}(K_2, V_2) \\
\text{reduce: } (K_2, \text{list}(V_2)) \rightarrow \text{list}(K_3, V_3)
\]

where \((K_i, V_i)\) always represent the (key, value) pairs that act as input and output for map and reduce functions. In general, the map input key and value types \((K_1 \text{ and } V_1)\) are different from the map output types \((K_2 \text{ and } V_2)\). However, the reduce input must have the same types as the map output, although the reduce output types may be different again \((K_3 \text{ and } V_3)\) (Dean & Ghemawat 2008). Programs written in this functional style are automatically parallelized by the Map/Reduce framework and executed on a large cluster of commodity machines. The run-time system takes care of the details of data distribution, scheduling various map and reduce functions to run in parallel across the set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any prior experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

There are additional elements that complete the programming model, which includes partitioners, combiners and configuration object. Partitioners are responsible for dividing up the intermediate key space and assigning intermediate key-value pairs to reducers. In other words, the partitioner specifies the task to which an intermediate key-value pair must be copied. Combiners are an optimization in MapReduce that allow for local aggregation before passing the intermediate key-value pairs to reducers. A configuration object of type JobConfig can be created to set the parameters required by the map and reduce tasks in run-time. Configuration objects can be used to store values that are shared among multiple MapReduce tasks.
5.3 THE HADOOP FRAMEWORK

Hadoop is an open source implementation framework that provides libraries for distributed computing using simple map/reduce interface and Hadoop distributed file system called HDFS. The Hadoop architecture shown in Figure 5.1 is from http://hadoop.apache.org/. HDFS is a block-structured file system managed by a central NameNode. Individual files are broken into blocks of a fixed size and distributed across multiple DataNodes in the cluster. The NameNode maintains metadata about the size and location of blocks and their replicas. SMS Planner is the query converter.

Figure 5.1 Hadoop Architecture

The Hadoop Framework follows simple master-slave architecture. The master is a single JobTracker and the slaves or worker nodes are TaskTrackers. The JobTracker handles the runtime scheduling of MapReduce jobs and maintains information on each TaskTracker’s load and available
resources. Each job is broken down into Map tasks based on the number of data blocks that require processing, and Reduce tasks. The JobTracker assigns tasks to TaskTrackers based on locality and load balancing. It achieves locality by matching a TaskTracker to Map tasks that process data local to it. It load-balances by ensuring all available TaskTrackers are assigned tasks. TaskTrackers regularly update the JobTracker with their status through heartbeat messages. The InputFormat library represents the interface between the storage and processing layers.

InputFormat implementations parse text/binary files (or connect to arbitrary data sources) and transform the data into key-value pairs that Map tasks can process. It facilitates scalability and takes care of detecting and handling failures. Hadoop has components which takes care of all complexities and by using a simple map reduce framework it is possible to harness the power of distributed computing without having to worry about complexities like fault tolerance and data loss. It has replication mechanism for data recovery and job scheduling. Hadoop provides the application programmer the abstraction of map and reduce. Hadoop is a highly scalable computing and storage platform.

5.4 DESIGN PATTERNS IN MAPREDUCE

Designing a good MapReduce program is a challenging task because, it is important to design a clean, maintainable code that performs well on tasks distributed over hundreds of machines computing terabytes or petabytes of data. There are a number of design patterns that are used for designing efficient mapReduce algorithms discussed by (Lin & Dyer, 2010; White 2012; Miner & Shook, 2012 ). In this Section, the design patterns used in the proposed FD discovery methods are discussed. In-Mapper Combiner with Stripes( Lin & Dyer, 2010) is the design pattern used in the proposed work. The In-Mapper Combiner pattern is used to substantially reduce both
the number and size of key-value pairs that need to be shuffled from the mappers to the reducers.

An associative array (Map in Java) is introduced inside the mapper to locally aggregate key-value pairs. The associative array is the data structure called stripes which is used to store a complex value. The mapper emits this associative array(set of stripes) instead emitting each key-value pair as output reducing the number and size of key-value pairs passed from mapper to reducer. The In-mapper Combiner pattern includes the combiner code in the map function instead of a separate Combiner class. The reason for not using combiner as a separate class from mapper is that even if its explicitly set for a MapReduce job, Hadoop may or may not run the combiner. With In Mapper Combiner design pattern, the programmer can explicitly and deterministically control how to reduce the number of intermediary key-value pairs (Lin & Dyer, 2010).

5.5 PROPOSED WORK

In this Section, the parallel Information Theory based FD extraction algorithm designed using Hadoop-Mapreduce model (ITFDMR) is presented. The input data set is saved on the Hadoop Distributed File System, because HDFS can hold huge chunks of data and HDFS can help data localization for MapReduce task. The path of the input files can be specified during the run time. The input data is divided into parts and allotted to the mapper. The pruning rules stated in Chapter 4 are used for pruning edges and candidates from the search space.

The ITFDMR algorithm is a 2-phase algorithm, where the first phase includes InitalMapReduce task, performing inference of candidate keys, equivalent attributes and non-dependants by search through the initial candidate list of the attribute semi-lattice. The initial candidate list includes
level 1, 2, n and n-1 candidates, where n is the number of attributes. The second phase of the algorithm includes IterativeMapReduce task, that has k passes, where k = n-3. The IterativeMapReduce task generates level k candidates from level k-1 candidates generated during previous iteration. The presence of FDs is checked by comparing the entropy of level k candidates with that of their subsets in level k-1.

**Initial Map task //one for each split**

```
Input : Si // input split i, each line in the split is a record.
Output: < key1, value1> key- Initial candidates from level1,2,n,n-1
        Value- partial frequency count for the values of candidates.
1. map( S_i, record)    // map function
2. for each record in Input Split i
3.    GenerateInitialCandidateList(L_1,L_2,L_n,L_n-1);
4.    for each candidate c_j \in L_k where j=1,2,..sizeOf(L_k) and k=1,2,n-1 and n
5.        HashMap Hm_{c_j} = new HashMap();
6.        recordSplit[ ] = record.Split(“,”)
7.        for each value v \in recordSplit[c_j]
8.            count = Hm_{c_j}.Get(v);
9.            if (count = = 0)
10.                Hm_{c_j}.set( v,1);
11.            else
12.                Hm_{c_j}.set( v, count+1);
13.            End for each;
14.        Emit( c_j, Hm_{c_j});
15.    End for each;
16.    End for each;
17.    End map;
```

The IterativeMapReduce task is terminated after n-3\(^{th}\) pass or when there are no candidates in the search space for further comparisons. The pseudocode for Initial map task is shown above. The combiner code is added
as a part of mapper, in the proposed implementation and it is called as in-mapper combiner (Lin & Dyer, 2010). The in-mapper combiner finds the partial frequency count for the values of initial candidate list.

The split chunk of the data file is fed to Mapper. The Map reads one record at a time and splits the records into tokens. The tokens are grouped based on the current candidate being processed. The local aggregation of count of value of each candidate is found. The output from the in-mapper-combiner is the candidates (representing the keys of key-value pair) and the partial frequency count for the candidate values (representing the values of the key-value pair). The mapper output is written in the temporary files of HDFS. The result from the in-mapper-combiner is then taken in by the reducer for further processing.

The reducer combines the partial frequency count for the values of a candidate to compute the total frequency count for the values of the candidates in the initial candidate list. After getting the frequency count for the values of a candidate, its entropy is computed. The procedure pruneCandidates(), apply pruning rules 1 and 2 to eliminate candidate keys and equivalent attributes from level 1 and level 2 candidates. Only those unpruned candidates and their entropy are written into the output file. The candidate keys and the equivalent attributes identified are added to configuration object. The procedure CheckFDs() use theorem 1 to check the presence of FDs. The discovered FDs are added to the configuration object, which are used in latter passes of the algorithm. The CheckNonDependants() procedure uses pruning rule 4 to check the presence of attributes that are not dependant on any other attributes in the relation. The level 2 candidates and their corresponding entropy are finally written into temporary files in the HDFS, and is used for generating level 3 candidates for further iteration.
Initial Reduce task // one for each intermediate key from map

**Input:** <key 2, value 2>
   Key 2 - candidates of Initial Candidate List
   Value 2 - HashMap of partial frequency count for the values of Candidates in Initial Candidate List.

**Output**<key3, value 3>
   Key 3 - candidates c_j of level 2 for j=1,2, sizeOf(Level2)
   Value 3 - Entropy(c_j) // H(c_j)

1. Reduce(key2, value2)
2. for each intermediate key key2 // for each candidate c_j ∈ L_k where
   j=1,2,..sizeOf(L_k) and k=1,2, n-1 and n
3. While (value2.hasNext())
4.   Hm_cj = (Map)value2.getNext();
5.   for each value v in Hm_cj
6.     pofv[] = findPofv(Hm_cj.get(v));
7.     computeEntropy(pofv)
8. end While;
9. pruneCandidates(); // check for keys and equivalent attributes and add to configuration object.
10. CheckFDs(); // Add FDs to configuration object.
11. CheckNonDependants(); // add non-dependants to configuration object
12. Emit(c_j ∈ L_2, Entropy(c_j));
13. end for each;
14. end Reduce;

Following the InitialMapReduce task, the IterativeMapReduce task is started. The iterative MapReduce task is run for n-3 times, to discover FDs existing between level 3 to level (n-2) candidates level by level. Iterative Map task computes partial frequency count for the values of level k candidates that are supersets of level k-1 candidates stored in the temporary file of HDFS.
Iterative Map Task

| **Input**: Si  // input split i, each line in the split is a record. |
| **Output**: <key, value> key- Candidates of level $L_k$  
  Value- partial frequency count for the values of $L_k$ candidates. |

1. map(Si, record)  // map function
2. Read $L_{k-1}$ candidates and their entropy from Distributed Cache.
3. if(CandidateList.isEmpty())
   System.exit();
4. GenerateCandidates($L_k$)
5. for each record in Input Split i
6.   for each candidate $c_j \in L_k$ where $j=1,2,...\text{sizeOf}(L_k)$
7.      HashMap Hm$c_j$ = new HashMap();
8.      recordSplit[j] = record.Split(“,”)
9.      for each value $v \in \text{recordSplit}[c_j]
10.         count = Hm$c_j$.Get(v);
11.         if (count == 0)
12.             Hm$c_j$.set(v,1);
13.         else
14.             Hm$c_j$.set(v, count+1);
15.     End for each;
16.     Emit($c_j$, Hm$c_j$);
17.     End for each;
18.     End for each;
19.     End map;

The iterative map function carries out the same operations as that of Initial Map function, except calling GenerateCandidates() method in step 4. This method generates level $k$ candidates from level $k-1$ candidates stored in temporary HDFS file. The iterative map task emits level $k$ candidates as intermediate keys and their partial frequency count for the values as list of hashmaps (values). The iterativeMapper function is included as a member function of IterativeMapper class. The iterative reduce task, computes the
entropy of level k candidates using their partial frequency count for the values returned by map task.

The procedures pruneCandidates(), pruneComparisons(), CheckFDs() are same as that of the initial reduce task. The level k candidates that remain after pruning, are written to the temporary HDFS file, and are used for next iteration. The iterative reduce task terminates when the candidate list has no more candidates to verify. The level k candidates and their entropy are stored within output folder under a sub-folder with an iteration number. The output file forms an input file for the next iteration for generating next higher level candidates.

**Iterative Reduce Task**

<table>
<thead>
<tr>
<th>Input: &lt;key k, value k&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key k - candidates of L_k</td>
</tr>
<tr>
<td>Value k - HashMap of partial frequency count for the values of L_k candidates</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output &lt;key k, value k+1&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key k - candidates c_j of level L_k for j=1,2,...sizeOf(L_k)</td>
</tr>
<tr>
<td>Value k+1 - Entropy(c_j)/ H(c_j)</td>
</tr>
</tbody>
</table>

1. Reduce(key-k, value-k)
2. for each intermediate key key-k // for each candidate c_j \in L_k where j=1,2,...sizeOf(L_k) and k=3,4.. n-2
3. While (value-k.hasNext())
4. Hm_cj = (Map)value-k.getNext();
5. for each value v in Hm_cj
6. povf[v] = findPovf(Hm_cj.get(v));
7. end for each;
8. computeEntropy(povf);
9. end While;
10. pruneCandidates(); // check for keys and equivalent attributes and add to configuration object.
11. if(CandidateList.isEmpty());
12. System.exit();
13. pruneComparisons();
14. CheckFDs(); // Add FDs to configuration object.
15. Emit(c_j \in L_H, Entropy(c_j));
16. end for each;
17. end Reduce;
The discovered candidate keys and equivalent attributes are added to configuration objects and are used by the pruneCandidates() method. The FDs discovered are stored in another configuration object and is used by pruneComparisons() method to discover only minimal FDs. The pruneComparisons() method also uses the non-dependants list stored in configuration object to prune comparisons that has the non-dependants as RHS attribute of a FD.

The iterative mapper and reducer functions does the same job as initial mapper and reducer functions, but for candidates in level 3 to level n-2. The algorithm converges when there are no candidates to process further or when the last iteration is over, i.e level n-2 candidates are generated and their entropy is compared with that of level n-1 candidates. In summary, the benefits of the proposed algorithm are as follows:

- There is no need to centralize all data into one database. Instead, data is partitioned and stored in multiple nodes.
- The time taken for FD inference is dramatically reduced, especially for large datasets.
- The pruning rules are effective in filtering out unnecessary comparisons from the FD discovery search space.
- MapReduce is the most suitable approach to deal with today’s large scale distributed systems and large amount of data.

Using entropy as a dependency measure helps in identifying the degree of dependency and the proposed algorithm can be slightly modified to discover conditional (Fan et al 2011) and approximate functional dependencies (Huhtala et al 1999).
5.6 EXPERIMENTS

The experiments were carried out on Hadoop cluster of five nodes, having Intel® Core™ 2 Duo, 2.54 GHz processors with 2GB RAM running Ubuntu operating system with Hadoop 1.0.3 and Java 6 installed. There are four different types of cluster setups used for carrying out the experiments and they are described in Table 5.1.

<table>
<thead>
<tr>
<th>Type</th>
<th>No. of Data Nodes</th>
<th>No. of Name Node</th>
<th>No. of Job Tracker</th>
<th>Total No. of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>II</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>III</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>IV</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

Typically one machine in the cluster is designated as the NameNode and another machine as the JobTracker, exclusively. These are the actual master nodes. The rest of the machines in the cluster act as both DataNode and TaskTracker. These are the slaves or worker nodes. Type I is a single node cluster, where a single node acts as data node, name node, TaskTracker and JobTracker. The single node Hadoop was run on the virtual machine to represent the Hadoop environment. Type II is a 2 node cluster, where one node is configured as master and the other node is configured as slave.

The master node will run the “master” daemons for each layer: NameNode for the HDFS storage layer, and JobTracker for the MapReduce processing layer. Both machines will run the “slave” daemons: DataNode for
the HDFS layer, and TaskTracker for MapReduce processing layer. Basically, the “master” daemons are responsible for coordination and management of the “slave” daemons while the latter will do the actual data storage and data processing work. Type III is a 4 node cluster, where one node acts as master and 3 nodes act as slaves. Type IV cluster has one master node and 4 slave nodes.

The experiments were run on both synthetic data and real life data sets from UCI repository (Frank and Asuncion, 2010). A sequential implementation called ITFD algorithm is implemented using Java without Hadoop framework. Initially, synthetic data sets are used to test the performance of the proposed ITFDMR algorithm (run on cluster with two nodes) with that of ITFD algorithm. The number of tuples in the data set is fixed as 100 K and the number of attributes is varied from 5 to 25. The attribute correlation, which represents the degree of dependency between attributes, is kept at 25%. It is seen from Figure 5.2 that both the ITFD and ITFDMR algorithms execute at the same rate for smaller datasets.

![Figure 5.2 Execution Time on Synthetic Datasets (ITFD Vs ITFDMR)](image)
Only when the number of attributes is greater than 15, ITFDMR shows significant performance improvement, compared to ITFD method. The execution time taken by ITFDMR method is below 1000 seconds even when the number of attributes increases. To test the effectiveness of the pruning rules, the time is measured by running ITFDMR with pruning and ITFDMR without pruning. Figure 5.3 shows the execution time measured when the ITFD algorithm is run without pruning rules(ITFDMR) and with pruning rules(ITFDMR-PRUNE). The number of records is fixed as 100K and the number of attributes is fixed at 15. The attribute correlation is varied from 0% to 25%. When attribute correlation is 0%, there is no functional dependency in the dataset. Pruning rule 4 detects that all the attributes of the dataset are in the non-dependent list and no more search is required to check the presence of FDs. When the attribute correlation is below 10%, most of the comparisons between the candidates at different levels of the semilattice are pruned and hence the execution time gets reduced at a large extent.

![Figure 5.3 Execution Time on Synthetic Datasets (ITFDMR Vs ITFDMR-PRUNE)](image-url)
It is seen from Figure 5.3 that the time taken by the ITFDMR method with pruning rules is significantly less compared to the one without pruning rules.

The ITFDMR algorithm is run on multi-node cluster with the configurations shown in Table 5.5. As Figure 5.4 shows, when the number of nodes in the cluster is increased, the execution time of the ITFDMR algorithm reduces. The sythetic dataset is used to test the performance of the proposed algorithm on multi-node cluster. The number of attributes is fixed at 15 and the number of tuples are varied from 100K to 500K.

![Figure 5.4 Number of Nodes Vs Execution Time](image)

The execution time of the ITFDMR algorithm remains less than 2000 seconds when it is run on single node cluster. Executing the proposed algorithm on two node cluster, takes lesser time compared to single node cluster. For multi-node cluster with 4 nodes and 5 nodes, the execution time of ITFDMR is not above 100 seconds. The ITFDMR algorithm is also tested with real life datasets from UCI machine learning repository and the results are presented in Table 5.2.
Table 5.2 Execution Time of ITFDMR on UCI Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>No. of Attributes</th>
<th>No. of FDs</th>
<th>No. of Equivalent Attributes</th>
<th>No. of Non-Dependents</th>
<th>Time (seconds)</th>
<th>Non-Hadoop Node</th>
<th>Single Node Hadoop</th>
<th>2-node Cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abalone</td>
<td>8</td>
<td>60</td>
<td>8</td>
<td>0</td>
<td>40</td>
<td>54</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Chess</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>30</td>
<td>34</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Nursery</td>
<td>9</td>
<td>1</td>
<td>0</td>
<td>8</td>
<td>28</td>
<td>30</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Census</td>
<td>9</td>
<td>0</td>
<td>0</td>
<td>9</td>
<td>15</td>
<td>18</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Glass</td>
<td>10</td>
<td>27</td>
<td>1</td>
<td>1</td>
<td>31</td>
<td>42</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Hardware</td>
<td>10</td>
<td>33</td>
<td>22</td>
<td>0</td>
<td>34</td>
<td>50</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>Bridge</td>
<td>13</td>
<td>61</td>
<td>48</td>
<td>0</td>
<td>50</td>
<td>67</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>Echo</td>
<td>13</td>
<td>536</td>
<td>30</td>
<td>0</td>
<td>46</td>
<td>63</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>Adult</td>
<td>14</td>
<td>5</td>
<td>0</td>
<td>13</td>
<td>18</td>
<td>20</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Zoo</td>
<td>18</td>
<td>6151</td>
<td>617</td>
<td>2</td>
<td>625</td>
<td>840</td>
<td>415</td>
<td></td>
</tr>
<tr>
<td>Hepatitis</td>
<td>20</td>
<td>7381</td>
<td>4862</td>
<td>0</td>
<td>600</td>
<td>980</td>
<td>454</td>
<td></td>
</tr>
<tr>
<td>Mushroom</td>
<td>23</td>
<td>1</td>
<td>0</td>
<td>22</td>
<td>80</td>
<td>92</td>
<td>43</td>
<td></td>
</tr>
</tbody>
</table>

The number of tuples in the datasets taken from UCI repository is fixed as 50K by duplicating the records multiple times. This is done to measure the efficiency of the MapReduce version of ITFD algorithm on large datasets. The TFDMR algorithm is run on single node and two node cluster and the results are measured. The sequential version of ITFD algorithm takes longer time to complete, since the number of tuples in the datasets is very large.

An interesting observation made is that, when the datasets have 0% attribute correlation, the sequential version of ITFD algorithm takes almost the same time as that of ITFRMR. The ITFD algorithm works at a faster rate for datasets with poorly correlated attributes. None of the existing algorithms prune the non-dependents at the earlier state of FD discovery process. For
datasets with higher than 20 attributes and high attribute correlation, both the ITFD and ITFDMR algorithms take longer duration to complete. From the experimental results, it is proved that the MapReduce version of ITFD algorithm works extremely well on datasets with very large number of tuples. The ITFDMR algorithm also takes advantage of poor attribute correlation among attributes of the dataset to prune the non-dependents at the earlier state of FD discovery search operation.

5.7 SUMMARY

Parallel and distributed computing provides cost-efficient solutions of storing and analyzing mass data. In this study, functional dependency discovery using MapReduce strategy in the Hadoop environment is described. The improved algorithm can scale up to large data set with comparatively less cost. Moreover, this distributed algorithm can accommodate the distributed nature of the input data. Furthermore, details of the management of the distributed systems, such as data transferring among nodes and node failures are taken care by Hadoop, which adds a great deal of robustness and scalability to the system. Based on the experimental evaluation, it is found that the improved FD discovery algorithm achieves the best performance for large datasets with sparsely correlated attributes. Quality of very large datasets can be assessed based on the constraints extracted from data.