MapReduce Network Enabled Algorithms for Classification Based on Association Rules

A Thesis submitted for the Degree of

Doctor of Philosophy

By

Suhel Hammoud

Electronic and Computer Engineering

School of Engineering and Design

Brunel University

May 2011

Abstract

There is growing evidence that integrating classification and association rule mining can produce more efficient and accurate classifiers than traditional techniques. This thesis introduces a new MapReduce based association rule miner for extracting strong rules from large datasets. This miner is used later to develop a new large scale classifier. Also new MapReduce simulator was developed to evaluate the scalability of proposed algorithms on MapReduce clusters.

The developed associative rule miner inherits the MapReduce scalability to huge datasets and to thousands of processing nodes. For finding frequent itemsets, it uses hybrid approach between miners that uses counting methods on horizontal datasets, and miners that use set intersections on datasets of vertical formats. The new miner generates same rules that usually generated using apriori-like algorithms because it uses the same confidence and support thresholds definitions.

In the last few years, a number of associative classification algorithms have been proposed, i.e. CPAR, CMAR, MCAR, MMAC and others. This thesis also introduces a new MapReduce classifier that based MapReduce associative rule mining. This algorithm employs different approaches in rule discovery, rule ranking, rule pruning, rule prediction and rule evaluation methods. The new classifier works on multi-class datasets and is able to produce multi-label predications with probabilities for each predicted label. To evaluate the classifier 20 different datasets from the UCI data collection were used. Results show that the proposed approach is an accurate and effective classification technique, highly competitive and scalable if compared with other traditional and associative classification approaches.

Also a MapReduce simulator was developed to measure the scalability of MapReduce based applications easily and quickly, and to captures the behaviour of algorithms on cluster environments. This also allows optimizing the configurations of MapReduce clusters to get better execution times and hardware utilization.

Acknowledgements

I would like to thank all those who have given me academic and moral support for my research work over the last years. I would like to thank the department of Electronic and Computing Engineering, in particular to my supervisor, Dr. Maozhen Li for his guidance and valuable advice.

I would like to thank my wife Zena Ibrahim for her support. I dedicate this work for her and for all my family members.

I would like to thank my friends Dr. Fadi Thabtah, Dr. Ivan Rankin, Yang Liu and Nasullah Khalid Alham for their help and advice in my research.

I would like to thank my friends Adam Daowd, Ahmad Daowd, and Ruaa Hasan for their support.

Declaration

The work described in this thesis has not been previously submitted for a degree in this or any other university and unless otherwise referenced it is the author's own work.

Statement of Copyright

Publications

The following papers have been published for publication or under review as a direct or indirect result of the research discussed in this thesis.

Journal Papers

F. Thabtah, P. Cowling, and **S. Hammoud**, "Improving rule sorting, predictive accuracy and training time in associative classification," *Expert Systems with Applications*, vol. 31, Aug. 2006, pp. 414-426.

N.K. Alham, M. Li, **S. Hammoud**, L. Yang, and M. Ponraj, "A distributed SVM for image annotation," *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, Aug. 2010, pp. 2983-2987.

Y. Liu, M. Li, N.K. Alham, and S. Hammoud, "HSim: A MapReduce simulator in enabling Cloud Computing," *Future Generation Computer Systems*, May. 2011.

N. K. Alham, M. Li, Y. Liu and **S. Hammoud**, Parallelizing Multiclass Support Vector Machines for Scalable Image Annotation, Neurocomputing, Elsevier Science 2010. (under review).

N. K. Alham, M. Li, Y. Liu and **S. Hammoud**, A Resource Aware Parallel SVM for Scalable Image Annotation, Parallel Computing, Elsevier Science. 2010. (under review).

Y. Liu, M. Li, N. K. Alham, **S. Hammoud**, A Resource Aware Distributed LSI for Scalable Information Retrieval, Information Processing and Management, Elsevier Science. 2010 (under review).

Conference Papers

Y. Liu, M. Li, **S. Hammoud**, N.K. Alham, M. Ponraj, Distributed LSI for Information Retrieval, Proc. of IEEE FSKD'10, pp. 2978-2982.

N. K. Alham, M. Li, **S. Hammoud**, Y. Liu, M. Ponraj, MapReduce-based Distributed SMO for Support Vector Machines, Proc. of IEEE FSKD'10, pp. 2983-2987.

S. Hammoud, M. Li, Y. Liu, N. K. Alham, Z. Liu, MRSim: A Discrete Event based MapReduce Simulator, Proc. of IEEE FSKD'10, pp. 2993-2997.

N. K. Alham, M. Li, S**. Hammoud** and H. Qi, Evaluating Machine Learning Techniques for

Automatic Image Annotations, Proc. of IEEE ICNC09-FSKD09, pp.245-249, 2009 (invited paper).

Table of Contents

List of Figures and Illustrations

List of Tables

Chapter 1

Introduction

This chapter briefly describes the background to the problems investigated in this thesis, the motivations of the work, major contributions and the structure of the thesis.

1.1 Background

Data mining, a technique to understand and convert raw data into useful information, is increasingly being used in a variety of fields like marketing, business intelligence, scientific discoveries, biotechnology, Internet searches, and multimedia. Data mining is an interdisciplinary field combining ideas from statistics, machine learning, and natural language processing.

Advances in computing and networking technologies have resulted in many distributed computing environments. The Internet, intranets, LANs, WANs, and peer-to-peer (P2P) networks are all rich sources of vast distributed databases. These distributed data sets allow large-scale data-driven knowledge discovery to be used in science, business, and medicine. Data mining in such environments requires a utilization of the available resources. Conventional data mining algorithms are developed with the assumption that data is memory resident, making them unable to cope with the exponentially increasing size of data sets. Therefore, the use of parallel and distributed systems has gained significance.

Generally, parallel data mining algorithms work on tightly coupled custom-made shared memory systems or distributed-memory systems with fast interconnects. Other algorithms designed for clusters like loosely coupled systems are connected over a fast Ethernet LAN or

WAN. The main differences between such algorithms are scale, communication costs; interconnect speed, and data distribution. MapReduce is an emerging programming model to write applications that run on distributed environments. Several implementations such Apache Hadoop are currently used on clusters of tens of thousands of nodes [1]. This thesis focuses on MapReduce design and the implementation of two new data mining techniques relating to associative rules and associative classification. This trend to use distributed, complex, heterogeneous computing environments has given rise to a range of new data mining research challenges. This work explores the different methods and trade-offs when designing and implementing distributed data mining algorithms. Particularly, it discusses data partition/replication and workload distribution, and data formats. Also, this work aims to investigate the hardware utilization when running MapReduce algorithms on the infrastructure. This helps to study the behaviour of algorithms on simulated large clusters. This helps rapid optimizing and rapid developing efficient algorithms that use the MapReduce framework.

1.1.1 MapReduce Framework

MapReduce [2] is a linearly scalable programming model. The programmer writes two functions—a map function and a reduce function—each of which defines a mapping from one set of key-value pairs to another. These functions are oblivious to the size of the data or the cluster that they are operating on, so they can be used unchanged for a small dataset and for a massive one. More importantly, if you double the size of the input data, a job will run twice as slowly. But if the size of cluster is doubled, a job will run as fast as the original one. This is not generally true of SQL queries.

One widely used implementation of MapReduce is Apache Hadoop [3] which is a collection of related subprojects that compose an infrastructure for distributed computing. These projects are open-source ones hosted by the Apache Software Foundation. Hadoop is known for MapReduce and its Hadoop Distributed File System HDFS [4], Hadoop provides complementary services, such as Core, MapReduce, HDFS, and HBase. Hadoop Core is a set of components and interfaces for distributed file systems and general I/O (serialization, Java RPC, persistent data structures). Hadoop MapReduce is distributed data processing model and execution environment that runs on large clusters of commodity machines. Hadoop HDFS is distributed file system that runs on large clusters of commodity machines.

Finally Hadoop HBase is distributed, column-oriented database. HBase is designed after Google Bigtable [5] and uses HDFS for its underlying storage, and supports both batch-style computations using MapReduce and point queries (random reads).

Using the MapReduce framework to build machine learning algorithms is investigated in [6] and Mahout [6] [7] to provide libraries for classification clustering and other machine learning algorithms.

1.1.2 MapReduce Simulation

Several simulation environments are available to simulate batch systems running on clusters of machines [8] [9] and others. MapReduce is a paradigm which is only few years old. MRPerf [10] is an available tool simulator. It uses network simulator NS2 [11] to simulate the network traffic and uses a call back feature to simulate CPUs and hard drives using average processing speed and average IO speed respectively.

Mumak [12] is an open source project aim to provide a tool for researchers and developers to prototype features (e.g. pluggable block-placement for HDFS, Map-Reduce schedulers etc.) and predict their behaviour and performance with a reasonable amount of confidence. Mumak takes as an input a job trace from jobs executed on real clusters.

MRPerf is a MapReduce simulator based on C++, TCL and Python. As they described that they presented the design of an accurate MapReduce simulator, MRPerf, for facilitating exploration of MapReduce design space. MRPerf captures various aspects of a MapReduce setup, and uses this information to predict expected application performance.

1.1.3 Mining Frequent Items and Associations

Frequent items are patterns of itemsets or sequences that frequently appear in a data set. For example, a set of items in shopping basket, such as milk and bread that appear frequently together in a transaction data set is a frequent itemset. Finding such frequent item plays an essential role in mining associations, correlations, and many other interesting relationships among data. Moreover, it helps in data classification as well. Several classifiers [13] [14] [15] [16] are built based on association rules. Thus, frequent pattern mining has become an

important data mining task and a focused theme in data mining research. Patterns represented in the form of association rules are human readable. For example, the association rule below:

$$\text{milk} \rightarrow \text{bread [support = 2\%; confidence = 60\%]}$$

Rule support and confidence are two measures of rule interestingness. They respectively reflect the usefulness and certainty of discovered rules. A support of 2% for an Association Rule means that 2% of all the transactions under analysis show that milk and bread are purchased together. A confidence of 60% means that 60% of the customers who purchased milk also bought bread. Association rules are considered interesting if they pass both a minimum support threshold and a minimum confidence threshold. Such thresholds can be set by users or domain experts.

Many algorithms have been developed for frequent itemset mining, from which association rules can be derived. Some of these algorithms are Apriori-like algorithms. Others are using growth-based algorithms [17] in addition to algorithms that use the vertical data format [18] [19].

The Apriori algorithm is one of the first algorithms used for mining frequent itemsets to get association rules. It employs the mining Apriori property that subsets of a frequent itemset are also frequent items. It iterates over the data to generate frequent k-itemset candidates based on the frequent (k-1)-itemsets. Variations involving hashing [20] and transaction reduction can be used to make the procedure more efficient. Other variations include partitioning the data [21] (mining on each partition and then combining the results) and sampling the data [22] [23] (mining on a subset of the data). These variations can reduce the number of data scans required.

Frequent pattern growth (FP-growth) [17] is a method of mining frequent itemsets without candidate generation. It constructs an FP-tree data structure to compress the original transaction database. It tries to achieve greater efficiency by focusing on frequent pattern growth, thus there are no candidate items generation steps.

Mining frequent itemsets using a vertical data format [19] (ECLAT) is a method that transforms a given data set of transactions in the horizontal data format of TID-itemset into the vertical data format of an item-TID set. It employs Apriori properties and mines the

transformed data set by TID set intersections for generating lower degree frequent items and vrepeating set intersections till it gets higher degree candidate rules. Additional optimization techniques are used in this method such as diffset [18]. Other extensions include using multiple supports thresholds [24] defined for each level of abstraction.

1.1.4 Classification and Prediction using Association Rules

Association rules are not used directly for prediction without further analysis or domain knowledge. If one of the attributes is used as class label and the other attributes are used as conditions then associated rules can indicate causality used in classification. A classifier is usually built in two steps consisting of discovery of frequent itemset mining, as before, but narrows the search to items of targeted class attribute. The second step is to generate strong associations between frequent patterns and class labels. Several studies indicated that associative classification has been found to be more accurate than some traditional classification methods, such as C4.5 [25]. Classification Based on Associations (CBA) [13], Classification Based on Multiple Class-Association Rules (CMAR) [26], Multi-class Classification based on Association Rule (MCAR) [15] and Classification based on Predictive Association Rules (CPAR) [27] that adopt methods of frequent itemset mining to generate candidate association rules.

CBA (Classification-Based Association) [13] is one of the earliest and simplest algorithms for associative classification. CBA uses an iterative approach to frequent itemset mining, similar to that described for Apriori. Multiple passes are made over the data and the derived frequent itemsets are used to generate and test longer itemsets

CMAR [26] (Classification based on Multiple Association Rules) differs from CBA in its strategy for frequent itemset mining and its construction of the classifier. It also employs several rule pruning strategies with the help of a tree structure for efficient storage and retrieval of rules. CMAR adopts a variant of the FP-growth algorithm to find the complete set of rules satisfying the minimum confidence and minimum support thresholds.

CMAR employs another tree structure to store and retrieve rules efficiently and to prune rules based on confidence, correlation, and database coverage. Rule pruning strategies are triggered whenever a rule is inserted into the tree.

MCAR [15] improves the efficiency of the rule discovery phase by employing a method that extends the tid-list intersection methods.

1.1.5 High Performance Computation in Mining Association Rules

The count distribution [28] algorithm is a distributed formulation of the Apriori algorithm, in which each processor generates its own version of the candidate hash tree. The counts of the candidates are estimated by performing a single pass over the local database, and a global reduction operation is then performed to estimate the global support of the candidate itemsets. When the globally frequent itemsets at level-k have been discovered, each processor generates the k+1-candidate itemsets in parallel, and repeats the process till all frequent itemsets have been found.

The Parallel Data Mining for association rules (PDM) algorithm [29] is a parallel implementation of the serial Direct Hashing and Pruning (DHP) algorithm [20]. While it is similar in nature to count distribution, the major difference is the use of a parallel hash table. The database is distributed among the processors, and each processor generates disjoint candidate itemsets. As in DHP, the hash table is built during the candidate counting phase and used to prune candidates in the subsequent generation phase. For this to happen, each processor needs to have a copy of the global hash table. Since PDM maintains parallel hash tables, this requires communicating the counts of each location in the hash table by a global exchange. Communicating the entire hash table is inefficient [30].

Task distribution: Another parallelization paradigm is to replicate the candidate generation process on each processor, and parallelize the support counting process. The processors will then perform different computations independently, but will need access to the entire database.

The Data Dist [28] algorithm addresses the memory problem of the count distribution algorithm by splitting the candidate generation process among the processors. The candidate itemsets are distributed among the processors in a round robin manner, and each processor computes the global counts of the candidates assigned to it. In order to do this, each processor needs to scan the transactions belonging to all the other processors. This is done by an asynchronous send–receive policy, wherein each processor allocates P buffers to read and write transactions from other processors. While this scheme allows each processor to handle a

greater number of candidates, the communication overheads make it an order of magnitude slower than the count distribution algorithm. There are a number of other algorithms such as Intelligent Data Distribution (IDD) [31], Hybrid Distribution (HD) [31], and Parallel Eclat [32] are based on the task distribution paradigm. These algorithms allow each processor to handle a subset of the candidate itemset and attempt to minimize communication and achieve load balance. The manner in which the database is replicated also impacts performance. A detailed summary of these techniques is available in [33] [32].

1.2 Motivations of the Work

The MapReduce framework [2] is proved to be reliable low cost, data intensive framework used by giant enterprises such as Yahoo and Google [34] on clusters of tens of thousands of machines. Although a number of machine learning techniques are available for association and classification, very few are designed to be MapReduce aware algorithms. This work aims to introduce two new algorithms designed to naturally fit with the MapReduce programming model to benefit from the high scalability of MapReduce applications. Also, it aims to simulate the performance of MapReduce jobs using new general purpose MapReduce simulator MRSim [35]. The proposed algorithms are MRApriori for association rules and MRMCAR for associative classification.

The basic function of the MapReduce model is to iterate over the input, compute key/value pairs from each part of input, group all intermediate values by key, then iterate over the resulting groups and finally reduce each group. The programmer can abstract from the issues of distributed and parallel programming. The MapReduce implementation deals with issues such as load balancing, network performance, fault tolerance etc [36]. The Apache Hadoop [3] project is an open-source implementation of Google's MapReduce written in java for reliable, scalable, distributed computing. Recently there have been many applications adapted to the MapReduce model; however these applications are tested with small and medium size clusters of participating nodes. Having such clusters of nodes it is difficult to measure the scalability of the algorithms. Scalability is a measure of efficiency of an algorithm with a much larger cluster i.e. hundreds of nodes as well using a much larger training dataset. It is almost impractical to set up a very large cluster consisting hundreds or thousands of nodes to measure the scalability of an algorithm. The Hadoop environment set-up involves alterations of a great number of parameters which are crucial to achieve best performances. An obvious

solution to the above problems is to use a simulator which can simulate the Hadoop environment; a simulator on one hand allows us to measure scalability of MapReduce based applications easily and quickly, on the other hand to determine the effects of different configurations of the Hadoop set-up on MapReduce based applications behaviour in terms of speed. Hadoop is offered as a service on Amazon Elastic Compute Cloud EC2 [37] where users can launch and terminate instances on demand and pay by the hour for active instances. Also a Hadoop cluster can be integrated in Sun Grid Engine. Thus, MapReduce simulators will be very useful utilities to allow users to estimate times and the costs for the jobs before submitting it to EC2 or Sun Microsystems Grid Compute Utility.

For finding association rules, it has been widely recognized that finding frequent items is computationally intensive when the size of a training dataset is large. An Apriori-like algorithm usually involves repeat scanning of datasets. To speed up training for associative classification, distributed computing paradigms have been investigated to partition a large training dataset into small data chunks and process each chunk in parallel utilizing the resources of a cluster of computers [21] [33] [29]. The approaches include those that are based on the Message Passing Interface (MPI). However, MPI is primarily targeted at homogeneous computing environments and has limited support for fault tolerance. Furthermore, inter-node communication in MPI environments can create large overheads when shipping data across nodes. Although some progress has been made by these approaches, existing distributed apriori algorithms usually partition large datasets into smaller parts with the same size which can be used efficiently only in homogeneous computing environments in which the computers have similar computing capabilities. Currently heterogeneous computing environments are increasingly being used as platforms for resource intensive distributed applications. One major challenge in using a heterogeneous environment is to balance the computation loads across a cluster of participating computer nodes. Using the MapReduce framework, the two algorithms developed are naturally load balanced to heterogeneous environments.

Multi-Class Association Rule (MCAR) [15] and Multi-Class Multi-Label (MMAC) [38] are two algorithms developed for associative classification. Both use the vertical dataset representation introduced by Eclat [19]. Finding frequent itemset is done by doing set intersections between sets that hold the occurrences of lower degree frequent itemsets. This

work tries to consider the steps of MCAR [15] and MMAC [38] to develop a generalized associative classifier MRMCAR that produces multi-label rules with probabilities attached to each predicted class.

1.3 Major Contributions

This work has produces a design and an implementation for MapReduce simulator (MRSim) to simulate the behaviour of newly developed data mining MapReduce based algorithms in a Hadoop environment. With the real implementation of the algorithms, this helps to find the best values of parameters to tune the cluster for the best performance. MRSim is to be used later to evaluate the behaviours and the scalability of newly developed algorithms for MapReduce environments.

MRSim extends a discrete event engine used SimJava [39] to accurately simulate the Hadoop environment. Using SimJava MRSim simulates interactions between different entities within clusters. The GridSim [8] [40] package is also used for network simulation. It is written in the Java programming language on top of SimJava. Evaluation of MRSim is performed using a number of MapReduce based applications which have been implemented recently. Evaluation results show a high level of accuracy from different aspects. MRSim is modelled with several layers. This makes it easier to plug-in more components to it, such as new types of job schedulers. Network topology and hardware specifications are introduced to the simulator in text files of JSON format. Also, the definitions of algorithms are done in JavaScript Object Notation (JSON) format. Source code for MRSim is available for the community on [41].

MRApriori, a MapReduce aware distributed associative algorithm for finding frequent items has been implemented. MRApriori builds on the apriori algorithm for high efficiency in training and employs MapReduce. Two implementations of MRApriori are introduced; one is in-memory sequential application which is pluggable to WEKA [42] machine learning software and the second implementation was done using the Apache Hadoop distributed system platform. The Hadoop implementation [3] of MapReduce was used. MRApriori uses both vertical and horizontal dataset representations to find all frequent item sets. Including the implementation in WEKA machine learning software makes it available to the community.

This work also introduces MRMCAR (MapReduce Multi-Label Classifier based on Associative Rules). MRMCAR is a MapReduce based generalization of MCAR [15] classifier. Several ranking methods are plugged into the system allowing different classifiers for different datasets. Two implementations of MRMCAR are introduced; one is an in-memory application which is pluggable to WEKA machine learning software and the second implementation is done using Apache Hadoop [3]. MRMCAR uses both vertical and horizontal dataset representations to find all frequent item sets. Including the implementation in WEKA machine learning software makes it available to the community. Also, the incremental learning capability of MRMCAR is discussed and prototype was implemented using Google Bigtable distributed data structure. The performance of MRMCAR is evaluated and compared with several classification algorithms such as C4.5 [25], J48 [43] , RIPPER, CBA [13], and MCAR [15]. Results are discussed to appreciate the cost of MRMCAR classification to make it ready to be used in later machine learning analysis and specific classification applications.

The MapReduce framework facilitates a number of important functions such as partitioning the input data, scheduling MapReduce jobs across a cluster of participating nodes, handling node failures, and managing the required network communications. A notable feature of the Hadoop implementation of MRApriori and MRMCAR is the ability to support heterogeneous environments. This was utilized to design MRApriori and MRMCAR for effective load balancing scheme for resources with varied computing capabilities. Source code for MRApriori and MRMCAR with both Weka and Hadoop implementations is available for the community [44].

1.4 Structure of the Thesis

The rest of this thesis is organized as follows:

Chapter 2 reviews two separate subjects; MapReduce framework, and associative classification in machine learning algorithms. The first part of the chapter introduces MapReduce and using it in large scale intensive data applications. It explains the MapReduce programming model and introduces several of its implementations. Then it reviews available MapReduce simulator and some of Grid System simulators. The second part of the chapter introduces the problem of mining frequent itemsets. It addresses common association rules

techniques and data formats used. This part reviews several commonly used classification algorithms and it concentrates on classification algorithms that are based on association rules.

Chapter 3 introduces the design of MRSim MapReduce Simulator used to evaluate the behaviour of Map Reduce jobs on Hadoop software, System Architecture, Core entities, Map-Reduce entities, MRSim user level and input specifications, validating MRSim, design of core entities, validation on real cluster environments. Validation, Job Execution Times, IO data sized used locally and shuffled among cluster nodes comparing MRSim with results obtained from terasort benchmark are discussed.

Chapter 4 describes the design and the implementation of MRApriori, an association rules distributed algorithm based on MapReduce framework. Features and constraints of the algorithms are discussed. Sample run results are presented in this chapter in addition to MRSim configuration and results to simulate MRApriori for a higher number of machines.

Chapter 5 is dedicated to the designing of the MRMCAR a multi-label associative rule classifier based on the MapReduce framework. Features of the algorithm are discussed. Challenges of distributions and constraints of incremental learning. and also scalability features are discussed.

Chapter 6 presents two implementations of the MRMCAR for training multiclass datasets; Weka plug-in sequential implementation and Hadoop parallel implementation. Several types of evaluations of the algorithm are presented. Firstly, the prediction accuracy of the algorithm is presented in experiments and compared with several existing classifiers. Also, other label-based measurements were calculated evaluating the cost of predication per predicted class.

Performance results, scalability results, using MRSim are also discussed in the chapter.

Finally, Chapter 7 summarizes the contributions of the thesis and proposes directions for future work.

Chapter 2

Literature Review

2.1 Introduction

The first part of this section summarizes the basic principles of the MapReduce model and discusses available implementations of the MapReduce framework. Also, it concentrates on available simulators for MapReduce and likewise environments.

In the second part of this chapter, the association rule mining problem is presented with a review of published research works conducted on it. Specifically, it discusses popular association rule mining approaches like Apriori [45], FP-growth [46], partitioning [47] and others. In the third part of this chapter, the classification problem in data mining is briefly defined with a review of well-known traditional classification approaches like decision trees, rule induction, and Naïve Bayes. Last part specially focuses on reviewing classification algorithms that uses associative classification such as CBA [13], CPAR [27], CMAR [26], CACA was proposed in [14], BCAR [48], and MCAR [15] [49].

2.2 Map Reduce Framework for Scalable Intensive Data Applications

There is increasing interests to use MapReduce [2] in distributed machine learning algorithms such as Apache Mahout [6] [7]. Several algorithms were paralleled using MapReduce framework such as DisCo clustering [50], Locally Weighted Linear Regression (LWLR), K-

Means, Logistic Regression (LR), Naive Bayes (NB), SVM, ICA, PCA, Gaussian Discriminant Analysis (GDA), Back Propagation, and several more in [51].

2.2.1 MapReduce Programming Model

Jeffrey and Sanjay [2] introduced the easy and abstracted programming model, MapReduce. Many computation problems can be expressed using this model. It is inspired by functional programming languages. The input and output data have a specific format of key/value pairs. The users express an algorithm using two functions: the Map functions and the Reduce function. The Map function is written by the application developer. It iterates over a set of the input key/value pairs, and generates intermediate output key/value pairs. The MapReduce library groups all intermediate values by key and introduces them to the reduce function. The Reduce function is also written by the application developer, it iterates over the intermediate values associated by one key. Then it generates zero or more output key/value pairs. The output pairs are sorted by their key value.

(input) <k1, v1> -> map -> <k2, v2> -> reduce -> <k3, v3> (output)

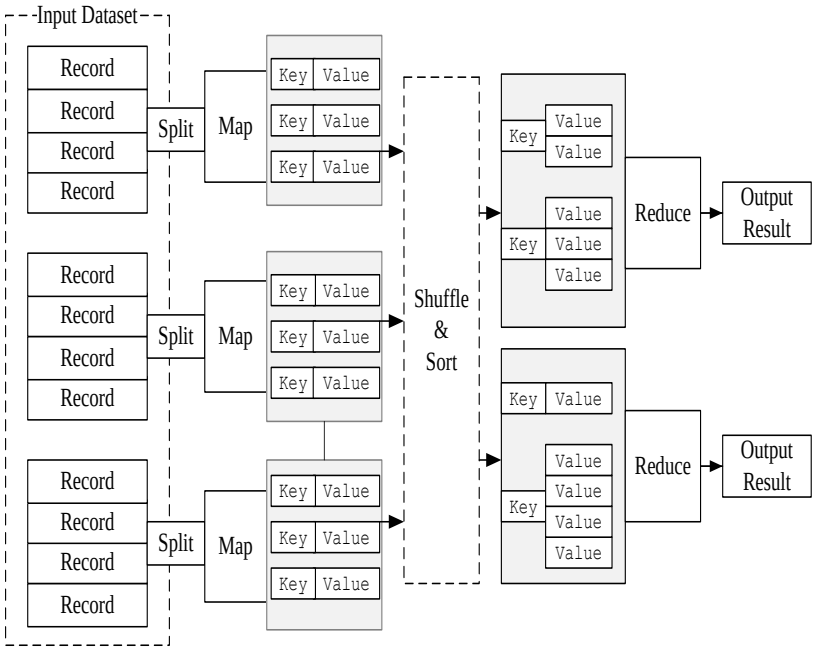


Figure 2-1: MapReduce model abstraction

Figure 2-1 shows the split of the input into logical chunks and each chunk is processed independently by a map task. The results of these processing chunks can be physically

partitioned into separate sets, which are then sorted. Each sorted chunk is passed to a reduce task.

2.2.2 MapReduce Implementations

While the programming model is abstracted, it is the job of the implementation to deal with the details of parallelization, fault tolerance, data distribution, load balancing, etc. Several implementations of MapReduce have been proposed some of them provided by academic research as MR-J [52] which used Java multithreading and Phoenix [53] [54] which uses c threads on shared memory systems to implement MapReduce. Other implementations used by enterprises include e.g. Microsoft Dryad [55] [56] , Greenplum MapReduce [57] Aster data SQL-MapReduce [58] , Hadoop [3] and Google MapReduce [59].

The Apache Hadoop project [3] is the most popular and widely used open-source implementation of Google's MapReduce. It is written in Java for reliable, scalable, distributed computing. The code is available as the Apache License Version 2.0 [60]. Hadoop is being used by known enterprises e.g. Facebook, Yahoo, Amazon and many others [34] .

2.2.3 Map Reduce Framework Simulator

MapReduce is an emerging model. Yet, not much research on simulating the performance of MapReduce cluster has been done. To the best of our knowledge MRPerf [61] [10] and Mumak [12] are the only simulators targeting the MapReduce framework. However, there is a closely related large-scale distributed computing paradigm, Grid computing [62]. Grid computing is a well known paradigm used to solve large-scale problems on distributed systems. Several simulators have been developed to simulate the performance of Grid systems including Bricks [63], MicroGrid [64], SimGrid [9]. Another approach to understanding how MapReduce behaves is to develop tracing tools to build a comprehensive view of the system. The Chukwa project [65] and X-trace [66] are examples of tools used to collect different measurements in real clusters.

2.2.4 Grid System Simulators

Bricks [63] simulation system simulates client-server applications. It follows a centralized global scheduling methodology.

The MicroGrid simulator [64] is modelled after Globus. It allows the execution of applications constructed using the Globus toolkit in a controlled virtual Grid-emulated environment. The results produced by emulation can be precise but it is very time-consuming because it runs on emulated virtual resources. Applications modelled in MicroGrid should be fully implemented as applications ready to run in a real environment. So developing a model to be emulated by MicroGrid takes more effort than models designed to run on other simulators.

The SimGrid toolkit [9] is a C language based toolkit for the simulation of application scheduling. It supports modelling of resources that are time-shared and the jobs can be submitted dynamically. SimGrid is restricted to a single scheduling entity and time-shared systems; it is difficult to simulate multiple users, applications and schedulers, with different policies. This needs a substantial extending of the toolkit.

The GridSim [8] simulator is close to SimGrid [9] but GridSim is implemented in the Java programming language and extensively uses a SimJava [67] discrete event simulation infrastructure. GridSim allows users to extend scheduling policies easier than SimGrid.

2.2.5 Limitations of Grid Simulators

Bricks, MicroGrid, SimGrid, GridSim and similar grid computing simulators cannot truly simulate the MapReduce framework. They model jobs submitted to the system as batch jobs where each job has fixed computational cost or CPU hours and has predefined input and output specifications. In the MapReduce framework, the interaction between hardware resources such as CPUs, memory buffers, local hard drives, and network adapters is more complicated and cannot be simplified as batch processes without losing a large amount of accuracy. For example, the reduce phase in any job is highly coupled and affected by the behaviour of all tasks executing at the map phase and no patch system can simulate this interaction accurately. The following simulators are dedicated for MapReduce frameworks.

2.2.6 Mumak MapReduce Simulator

Mumak [12], MRSim and MRPerf [10][61] focused on modelling the specifics of MapReduce framework and not grid systems, so it does not worry about reservations, resource brokering and wide-area scheduling used in the Grids.

Mumak [12] is an open source project aimed to provide a tool for researchers and developers to prototype features (e.g. pluggable block-placement for HDFS, Map-Reduce schedulers, etc.) and predict their behaviour and performance with a reasonable amount of confidence. Mumak takes as input a job trace from jobs executed on a real cluster. Then it simulates resubmitting the job on a Mumak virtual cluster. The output would be a detailed job execution trace recorded in virtual simulated time. Analyzing an output will provide more understanding of the effect of using different schedulers such as the effect of jobs' turnaround time, throughput, fairness, capacity guarantee, etc. Mumak accurately simulates the conditions of the actual system which would affect the scheduler's decision. This is because Mumak is unique in plugging in the real JobTracker and scheduler used in Hadoop middleware [12]. However, Mumak does not simulate tasks-sharing resources at a lower level. It does not simulate actual map/reduce tasks themselves. It merely takes the job history for jobs run on real clusters and keeps the run-time for each task as it is. It only re-allocates the tasks based on new schedulers and a new cluster environment. But tasks – definitely – will have a different execution time when using a new scheduling policy. This will generate unrealistic predictions if the same jobs were simulated on different clusters or when jobs were simulated on same cluster with different configurations. This is a major and the main limitation of Mumak. An example of wrong prediction is by doubling the number of tasks that each node can handle concurrently. Mumak will keep predicting the same time for each task as before, and this will reduce the total job time to half. In a real cluster, this scenario will roughly double the execution time for each task because now more processes are sharing the same hardware (CPUs, hard disks, and network adapters), but the total job time will not change very much from the previous run.

2.2.7 MRPerf MapReduce Simulator

Guanying Wang and et al. [10] proposed their MapReduce simulator MRPerf which is based on C++, TCL and Python. They presented the design of an accurate MapReduce simulator, MRPerf, for facilitating exploration of the MapReduce design space. MRPerf captures various aspects of a MapReduce setup, and uses this information to predict expected application performance. They designed MRPerf which can serve as a design tool for the MapReduce infrastructure, and as a planning tool for making MapReduce deployment far easier via reduction in the number of parameters that currently have to be hand-tuned using

rules of thumb. They validated their simulator using the data collected from medium-scale production clusters. The results showed that the simulator is able to predict application performance accurately, and thus can be a useful tool in enabling cloud computing.

Overview	Actual	*MRPerf*
Number of map tasks	480	476
Number of reduce tasks	16	16
Total input data	32G	32G
Total output data	32G	32G
Phases	**Actual**	***MRPerf***
Map	220.0	220.8
Shuffle	7.4	5.4
Sort	0.5	3.4
Reduce	137.9	135.9
Map break-down	**Actual**	***MRPerf***
map	2.14	2.10
sort	1.12	1.19
spill	4.22	4.58
merge	4.52	4.26
overhead	1.79	1.61
sum	13.80	13.75

Data locality	Actual		*MRPerf*	
	num	time	num	time
Data-local	468	13.77	468	13.66
Rack-local	6	13.60	3	14.67
Rack-remote	6	16.10	5	21.64

Figure 2-2: Detailed characteristics of a TeraSort job using MRPerf simulator.[10]

From the published testing results [10][61] as partly showed in Figure 2-2 MRPerf shows its high accuracy in simulating the impacts of changing the network topologies. This kind of accuracy is based on two points. The first point is that MRPerf introduced a network simulator ns-2 [11] to form its network component. The network simulator ns-2 has been developed for several years and has proved that it can provides support for simulation of TCP, routing, and multicast protocols over wired and wireless (local and satellite) networks so that the ns-2 simulator involved can produce high accuracy when MRPerf simulates the behaviours of networks [68]. The second point is that they involved several benchmarks including TeraSort, Search and Index, which are presented as being able to represent the standard MapReduce applications and the results of the tests are quite convincing.

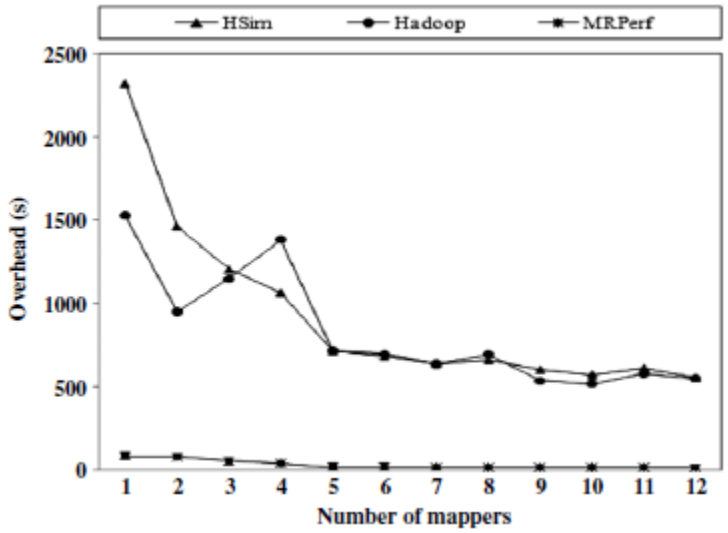


Figure 2-3: MR-LSI algorithm on HSim and MRPerf simulators vs. Actual Hadoop experiment. [69]

However, MRPerf shows less accuracy when used to simulate other algorithm of MR-LSI [69] as in Figure 2-3. This limitation maybe caused showed that realistic behaviours of the framework are based on a number of interactions of hardware and system components. For in Map phase, the performances of Map instances are very tightly coupled to the current states of processor, buffer, hard drive and networks. When certain thresholds are reached, according to the working mechanisms certain components may be interrupted to guarantee the performance and synchronizations. In Reduce phase, the performance of Reduce instances is highly dependent on the current IO states. The copying, shuffling and sorting procedures are quite dynamic according to the current system states. MRPerf does not simulate these real time interactions accurately. Instead, it employs a number of rough estimations to estimate the overhead of the Hadoop system. These approximations in terms of parameterization can not reflect real world Hadoop implementations. In TeraSort, Search and Index validations, none of these three algorithms involved complex behaviours of a Hadoop framework when the tests were carried out. So these rough estimations may generate small errors when MRPerf simulates these simple-behaviours-involved algorithms. However, whether the simulator can adapt to complex Hadoop behaviours is quite critical.

2.3 Mining Association Rule

Association rule mining was introduced by Agrawal, Imieliński, & Swami in 1993 [70]. It still has an active research area in the data mining and machine learning. Association rule mining finds correlations between items in a database. The classic application for association

rule mining is market basket analysis [45][70], in which business experts aim to investigate the shopping behaviour of customers in an attempt to discover regularities. The aim is to find groups of items that are frequently sold together in order that marketing experts can develop strategic decisions concerning shelving, sales promotions and planning. Association rule mining has been widely used in various industries beside supermarkets such as mail order [45], telemarketing [70][71], and e-commerce [72].

2.3.1 Association Rule Discovery Problem

Definition 2.1: The task of association rule discovery can be defined [70] as follows: Let D be a database of sales transactions, and $I = \{i_1, i_2, \ldots i_m\}$ be a set of binary literals called items. A transaction T in D contains a set of non empty items called an itemset, such that T $\subseteq$ I.

Definition 2.2: The support of an itemset is defined as the proportion of transactions in D that contain that itemset.

Definition 2.3: An association rule is an expression $X \rightarrow Y$, where $X, Y \subseteq$ I and $X \cap Y = \emptyset$

Definition 2.4: The confidence of an association rule is defined as the probability that a transaction contains Y given that it contains X, and given as

$$Confidence\ \left(X \Rightarrow Y\right) = \frac{support(X \cup Y)}{support(x)}$$

Given a transactional dataset D, the association rule problem is to find all rules that have supports and confidences greater than certain user-specified thresholds, denoted by minimum support and minimum confidence, respectively.

Finding rules from in dataset D consistes of two steps [45]: step one is to generate all frequent itemsets. Frequent itemsets are itemsets that have support greater than minimum support threshold. Step two is for each frequent itemset generated in Step one, produce all rules that pass the minimum confidence threshold those rule are considered interesting or strong rules. For example if itemset XYZ is frequent, then the confidence of rules XY $\rightarrow$ Z, XZ $\rightarrow$ Y and YZ $\rightarrow$ X can be evaluated using equation. The overall performance of mining

association rules is determined by the first step because it is relatively harder problem that requires extensive computation and storage [33][73].

Resulting rule pattern for shopping basket is of type:

$$\text{milk} \rightarrow \text{bread [support = 2\%; confidence = 60\%]}$$

A support of 2% for previous association rule means that 2% of all the transactions under analysis show that milk and bread are purchased together. A confidence of 60% means that 60% of the customers who purchased milk also bought the bread.

A major challenge in mining frequent itemsets from a large data set is the fact that such mining often generates a huge number of itemsets satisfying the minimum support threshold. This is because if an itemset is frequent then each of its subsets is frequent as well. A one candidate long frequent itemset of size 100 should contains $\binom{100}{1}$ frequent itemsets of size 1 and $\binom{100}{2}$ frequent itemsets of size 2 and so on total possible sub frequent itemsets is:

$$\binom{100}{1} + \binom{100}{1} + \cdots + \binom{100}{1} = 2^{100} - 1$$

This is too huge a number of itemsets for any computer to compute or store. Many researchers have extensively investigated the problem of efficiently finding frequent itemsets in association rule discovery in the last decade for the purpose of improving its efficiency [20][24][74][75].

2.3.2 Association Rule Data Layouts

There are several representations of a target database in association rule mining, these are the horizontal [45] , vertical layouts [33][76][77], and tree format for growth [46][78]. In the horizontal layout, the database consists of a group of transactions, where each transaction has transaction identifier (TID) followed by a list of items contained in that transaction. In the vertical layout on the other hand, the database consists of a group of items where each item is followed by its tid-list [47] transaction identifiers list that contains the item.

2.4 Common Association Rules Techniques

2.4.1 Apriori

Apriori is an algorithm that has been proposed in [45]. The discovery of frequent itemsets is accomplished in several iterations. In each scan, a full scan of training data is required to count new candidate itemsets from frequent itemsets already found in the previous step. Apriori uses the "apriori" property to improve the efficiency of the search process by reducing the size of the candidate itemsets list for each iteration.

DB : Transactional database

Output O set of all frequent items

F_n: Set of n-items that pass the minsupp threshold (frequent itemsets)

C_n : Set of n-candidate itemsets that are possibly frequent

1. F_1={frequent 1-itemsets};
2. for (n=2; $F_{n-1} \neq \emptyset \neq \emptyset$; n++) Do
3. C_n=generate_candidates(F_{n-1});
4. for each transaction $t \in DB$ Do
5. $P_t = \text{subset}(C_n, t)$
6. for each candidate $c \in P_t$
7. $c.\text{count} = c.\text{count} + 1$;
8. end //for
9. $F_n = \{c \in C_n \text{ if } c.\text{count} \geq \text{min support}\}$
10. $O = O \cup F_n$
11. end// for

Figure 2-4: Pseudo code for Apriori algorithm

Function Generate_Candidate (F_k):

1.For all $f_1, f_2 \in F_k$ where $F_k = \{i_1, i_2, \ldots, i_k\}$ do:

2. if and $i_k < i'_k$ Do

$f := f_1 \cup f_2 = \{i_1, i_2, \ldots, i_{k-1}, i_k, i'_k\}$

if $\forall_i \in f : f - \{i\} \in F_k$

$C := C \cup \{f\}$;

end if

end

return C

end

Figure 2-5: Pseudo code for generating candidate frequent items

The Apriori algorithm for finding frequent itemsets is shown in Figure 2-4, where the generate candidate function shown in Figure 2-5 , is used to produce $\boldsymbol{C_n}$ from $\boldsymbol{F_{n-1}}$ by merging $\boldsymbol{F_{n-1}}$ with $\boldsymbol{F_{n-1}}$, and discarding all itemsets in $\boldsymbol{C_n}$ that do not pass the support threshold. Once these candidate itemsets are identified from $\boldsymbol{C_n}$, then their supports are incremented (line 6-7). The algorithm terminates whenever there are no frequent itemsets $\boldsymbol{F_n}$ in the nth iteration.

2.4.2 Dynamic Itemset Counting

Dynamic Itemset Counting (DIC) [21] was developed to speed up the discovery of frequent itemsets in a database.DIC splits the database into several partitions marked by start points. Then, it calculates the supports of all itemsets counted so far, dynamically adding new candidate itemsets whenever their subsets are determined to be frequent, even if their subsets have not yet been seen at all transactions. The main difference between DIC and Apriori is that whenever a candidate itemset reaches the support during a particular scan, DIC starts producing additional candidate itemsets based on it, without waiting to complete the scan as Apriori does.

To accomplish the dynamic candidate itemsets generation, DIC employs a prefix-tree where each item counted so far is associated with a node. One of the drawbacks of DIC algorithm is

its sensitivity to how homogeneous the data is. Particularly, if the database to be mined is correlated,

2.4.3 Frequent Pattern Growth

Han, et al. [45] [46,78] presented a new association rule mining approach that does not use candidate rule generation called FP-growth that generates a highly condensed frequent pattern tree (FP-tree) representation of the transactional database. Each database transaction is represented in the tree by at most one path. FP-tree is smaller in size than the original database the construction of it requires two database scans, where in the first scan, frequent itemsets along with their support in each transaction are produced; and in the second scan, FP-tree is constructed.

Once the FP-tree is built, a pattern growth method is used to mine association rules by using patterns of length one in the FP-tree. For each frequent pattern, all possible other frequent patterns co-occurring with it in the FP-tree (using the pattern links) are generated and stored in a conditional FP-tree. The mining process is performed by concatenating the pattern with the ones produced from the conditional FP-tree. One constraint of FP-growth method is that memory may not fit FP-tree especially in dimensionally large database.

2.4.4 Partitioning

To reduce the number of database scans in association rule mining Savasere et al. [47] proposed an algorithm that divides the database into small partitions such that each partition can fit in the main memory and discovers frequent itemsets locally using a step-wise approach, e.g. Apriori, in the first pass. A tid-list structure for each itemset in a partition is then constructed. The tid-list of an itemset identifies rows in a partition that contain that itemset. The cardinality of an itemset tid-list divided by the total number of the transactions in a partition gives the support of that itemset.

In the second pass, the algorithm performs union operations on local frequent itemsets found in each partition to discover frequent itemsets in the database as whole. One of the drawbacks of the partitioning algorithm is that it prefers a uniform data distribution. For an unevenly distributed database, the majority of the itemsets in the second pass may be infrequent, causing extra I/O overhead. Furthermore, when the number of partitions increases, the

number of local frequent itemsets increases as well, consuming processing time and increasing redundant computation, especially when these partitions overlap in several frequent itemsets [19].

Performance comparison between Apriori and the partitioning algorithm using 6 market basket analysis data sets [45] revealed that the execution time of both algorithms increase when the support is reduced. A comparison using different number of partitions against the 6 benchmark problems indicate that the execution time decreases when less number of partitions is used due to the size of the candidate set normally becomes smaller.

2.4.5 Direct Hashing and Pruning

Generally, the computational cost of association rule mining is largely determined by the speed of discovery of frequent one and two itemsets. Empirical results from [45] suggest that the computational cost in the initial iterations dominates most of the execution time for the candidate generation phase. When the number of frequent itemsets during iteration 1 is large, the expected number of candidate itemsets at iteration 2 is also large, and thus, reducing the size of the candidate itemsets at early iterations may result in huge savings of processing time and memory. A hash-based technique, called Direct Hashing and Pruning (DHP), has been proposed in [20] to efficiently reduce the size of candidate itemsets at early iterations.

DHP works as follow: While scanning the database to find frequent one-itemsets, a hash tree, H1, is built for candidate one-itemsets to ease the search. The algorithm evaluates during the scan whether an item exists in the hash table, if so, the count of the item is incremented by one. Otherwise the item is inserted into the hash table and is given a count of one. Also, when the occurrences of all one-itemsets are counted for each transaction, all two-itemsets are produced and hashed into another hash table, H2, where a count is initialised to one for each itemset. Once the database is scanned, The possible candidate two-itemsets from H2 can be obtained.

Pruning occurs to reduce the database size during the scan in which not only a transaction is trimmed but also some of the transactions are removed. DHP trims an item in a transaction t if it does not have a certain number of occurrences in t's candidate itemsets. For example, If the support is set to 2, t = XYZWP and four two-subsets, (XZ, XW, XP, WP), exist in the hash tree constructed for candidate two-itemsets, H2, the number of frequencies according to

each item in t is 3, 0, 1, 2, 2, respectively. For frequent three-itemsets, only three items in t, e.g. (X, W, P), have occurrences above the support threshold. Consequently, these three items are kept in t and items Y and Z are removed.

Empirical study indicates that DHP reduces the execution times not only in the second iteration, when the hash table is employed by DHP to facilitate the production of candidate two-itemsets, but also in later iterations [20]. Particularly, the execution time required to produce candidate two-itemsets by DHP is orders of magnitude smaller than that of Apriori. However, the execution time of DHP is slightly larger than Apriori in the first iteration due to time required for building the hash table for candidate two-itemsets.

2.4.6 Multiple Supports Apriori

The support constraint is the most important factor that controls the number of association rules produced [70][74][32]. Setting the support to a high value results in discarding some useful rare items in the database. To capture such rare items, lower support thresholds is used. But this will also capture many un-interesting rules [24] [75].

To overcome such a problem,[24] proposed a multiple support Apriori-like approach, which represents the dataset in hierarchical concepts, then assigns different support values for each level. This enables users to express different support requirements for different rules. The candidate generation steps is still similar to the generate function in Apriori algorithm.

An evaluation study from [45] reveals that this method generates smaller number of candidate itemsets than that of Apriori for real world data sets. However, the execution time spent to find frequent itemsets for both algorithms is roughly the same.

2.4.7 Confidence-Based Approach

Confidence-based approach was proposed Li et all [75] to solve the problem of discarding rules with high confidence and low support. This method abandons the support threshold and mines only top confidence rules. Given a database, the end-user has to set an itemset target, which represents the consequent of the desired outcome (rules). The problem of mining high confidence rules is to find all a [79] association rules where the target is the consequent. In doing that, the algorithm divides the problem of mining confidence rules into two steps. Step

1 involves splitting the original database into two sets, one set that holds transactions containing the target itemset, T1, and the other holds the rest of the transactions, T2. The algorithm discards all items of the target from transactions in T1 and T2, therefore, the set of items in the original database I, becomes $I' = I - target$.

In the second step, all itemsets, X, which appear in T1 but do not appear in T2 are discovered, and rules such as $X \rightarrow tg$, is produced, where *tg* is the target consequent. These itemsets have a zero support in T2 but non-zero support in T1 and are called Jumping Emerging Patterns (JEP). The authors of [75] have adopted two border methods from [80] to discover itemsets whose support is zero in one sub-set, but non-zero in the other sub-set. The first border algorithm finds all itemsets with non-zero support in a data set and names them horizontal borders. When taking two horizontal borders produced from two sets of data, as an input, the second border algorithm can derive all itemsets whose support in one is zero, but non-zero in the other one.

Confidence-based approach can produce some high confidence rules that cannot be found by traditional association rules approaches. However, the candidate itemsets generated are many times larger than the original database. Therefore, a disk-based implementation is often preferred when pruning the search space using only the confidence threshold [79].

2.4.8 Tid-List Intersection

The Eclat algorithm has been presented in [19] and [18], which requires only one database scan. Eclat uses a vertical database transaction layout, where frequent itemsets are obtained by applying simple tid-lists intersections, without the need for complex data structures.

The recent variation of the Eclat algorithm, called dEclat, has been proposed in [18] which uses new vertical layout representation approach called a diffset. dEclat [19] stores only the differences in the transactions identifiers (tids) of a candidate itemset from its generating frequent itemsets. This considerably reduces the size of the memory required to store the tids. Experimental results in [18] revealed that dEclat and other vertical techniques like Eclat usually outperform horizontal algorithms like Apriori and FP-growth with regards to processing time and memory usage. Furthermore, dEclat outperforms Eclat on dense data, whereas the size of the data stored by dEclat for sparse databases grows faster than that of

Eclat. Thus, for dense databases, it is better to start with a diffset representation, and for sparse databases, it is better to start with a tid-list representation then switch to a diffset at later iterations [18].

2.4.9 Constraint-Based Association Mining

Often, users have a good sense of which direction of mining may lead to interesting patterns and the form of the patterns or rules they would like to find. Thus, a good heuristic is to have the users specify such intuition or expectations as constraints to confine the search space. This strategy is known as constraint-based mining. This can include Knowledge type constraints, data constraints, dimension/level constraints, interestingness constraints, rule constraints.

2.5 Classification in Data Mining

The goal of classification is to build a model (a set of rules) from a labelled training data set, in order to classify new data objects, known as test data objects, as accurately as possible. Figure 2.4 shows classification in data mining as a two-step process, where in the first step, a classification algorithm is used to learn the rules from a training data set. The second step involves using the rules extracted in the first step to predict classes of test objects.

There are many classification approaches for extracting knowledge from data such as divide-and-conquer [81], separate-and-conquer [82] [83] (also known as rule induction), covering [84] and statistical approaches [85] [86] [87]. Numerous algorithms have been based on these approaches such as decision trees [81] , PART [88], RIPPER [89] Prism [84] and others. Here is brief description of classification techniques related to the work of this thesis:

2.5.1 Simple One Rule

One of the simplest classification algorithms is One Rule 1R [90], which constructs a one-level decision tree and derives rules for training instances associated with most frequent classes. Two main challenges for classification algorithms are missing values and real-valued attributes [88] [91] . An experimental study [90] showed that, in most classification cases, simple techniques such as 1R generate reasonably accurate classifiers.

2.5.2 Decision Trees

A popular approach for classification and prediction is that of decision trees [92] [93]. In constructing a decision tree, a candidate record will enter the root node, and a branch for each possible value for the candidate is built. The same process is applied recursively until all the records in a node end up with the same class or the tree cannot be split any further [92] . After the tree has been constructed, each path from the root node to each of the leaf nodes represents a rule. The antecedent of the rule is given by the path from the root node to the leaf node, and the consequent is the majority class that is assigned by the leaf node.

Several pruning methods are used to simplify the rules and to discard unnecessary ones. Pruning the tree will involve either replacing some sub-trees with leaf nodes (sub-tree replacement) or raising some nodes to replace the nodes higher in the tree (sub-tree rising) [25]. Both of these operations are examples of post-pruning techniques [91]. One effective pruning method is to estimate the error rate at the internal and leaf nodes and then compare the error rates for the nodes with their replacement leaves [81].

2.5.3 ID3 Algorithm

ID3 is a decision tree algorithm introduced in [92]. ID3 utilises a statistical property called information gain to assess which attribute goes into a decision node. ID3 makes the selection of the root based on the most informative attribute and the process of selecting an attribute is repeated recursively at the so-called child nodes of the root, excluding the attributes that have been chosen before, until the remaining training data objects cannot be split any more [94]. Information gain measures how well a given attribute divides the training data objects into classes.

The basic ID3 is to be modified to handle missing attribute values and continuous attributes [25]. Also, there are different pruning methods proposed to produce a smaller subset of rules, such as replacing a sub-tree by a leaf node [25]. This replacement occurs if the expected error rate in the sub-tree is greater than that in the leaf node.

2.5.4 C4.5 Algorithm

C4.5 algorithm is an extension of the ID3 algorithm and was created by Quinlan [25] accounts for missing values, continuous attributes and pruning of decision trees. A commercial version that adds some minor modification to C4.5 named "C5" has been developed by Quinlan [95].

As for the ID3 algorithm, C4.5 uses information gain to select the root attribute. It calculates the Entropy for all attributes in order to select one as a root. The same process is repeated on the remaining attributes.

Missing values are treated by C4.5 using probabilities that are computed based on the frequencies of the different values for an attribute at a particular node in the decision tree [91]. Continuous attributes are discretized using a discretisation method such as [96]. One of the major extensions of the ID3 algorithm that C4.5 proposed is that of pruning. Two known pruning methods used by C4.5 to simplify the decision trees constructed are sub-tree replacement and pessimistic error estimation [97] [98]. Sub-tree replacement may be performed when a sub-tree has an expected error larger than its replacement leaf. At that point, the decision tree will be pruned by replacing a whole sub-tree by a leaf node [25]. J48 is an implementation of C4.5 under the WEKA [42] data mining platform

2.5.5 Statistical Approach (Naïve Bayes)

Unlike the 1R algorithm [90], statistical modelling uses all available attributes to make a prediction. One of the well-known statistical classification algorithms is Naïve Bayes [86], which computes the probability of each class for a data object using the joint probabilities of attribute values in that data object given the class. This algorithm assumes that the conditional probability of a data object given a class is independent of the probabilities of other data objects given that class. This naïve assumption is too optimistic since attributes in real world data sets are dependent on each others and could have different degree of importance. However, Naïve Bayes proved to work well in practice in many experimental studies [87] [80] [99].

2.5.6 Rule Induction and Covering Approaches

2.5.6.1 Incremental Reduced Error Pruning

Furnkranz and Widmer [100] proposed a learning algorithm called Incremental Reduced Error Pruning (IREP), which integrates a separate-and-conquer approach with Reduced Error Pruning (REP) [98] .REP was introduced as method that effectively prunes and produces a small set of classification rules. IREP constructs a rule set in greedy fashion where firstly, the training data is partitioned randomly into a growing set and a pruning set, where the growing set contains 66.6% of the training data objects. Rules are constructed greedily in IREP, starting from an empty rule; a condition (attribute value) is appended to its antecedent. The choice of which condition to add is preformed using Foil-gain measure [101]. IREP continuously adds conditions that maximise Foil-gain value, to the current rule until the rule covers no data objects from the growing set. After a rule is built, IREP immediately considers pruning it backwards by removing the final sequence of conditions from it. Starting from the last condition for each generated rule, IREP considers removing one condition at a time and chooses the deletion that improves the certain function. An Empirical study on different benchmark problems in [100] revealed that IREP is faster than REP and competitive to it with reference to error rate. In comparison to C4.5 algorithm [25] on 36 data sets, IREP achieved less error rate on 16, whereas C4.5 outperformed IREP on 21.

2.5.6.2 Repeated Incremental Pruning to Produce Error Reduction

Repeated Incremental Pruning to Produce Error Reduction algorithm (RIPPER) is a rule induction algorithm that has been developed by Cohen [89]. RIPPER builds the rules set as follows: The training data set is divided into two sets, a pruning set and a growing set. RIPPER constructs the classifier using these two sets by repeatedly inserting rules starting from an empty rule set. The rule-growing algorithm starts with an empty rule, and heuristically adds one condition at a time until the rule has no error rate on the growing set.

RIPPER stops adding a rule using the minimum description length principle (MDL) [102] where after a rule is inserted, the total description length of the rules set and the training data is estimated. If this description length is larger than the smallest MDL obtained so far, RIPPER stops adding rules. The MDL assumes that the best model (set of rules) of data is the

one that minimises the size of the model plus the amount of information required to identify the exceptions relative to the model [91].

A study on 36 benchmark problems from [103] has been reported in [89] in order to compare the prediction rate of RIPPER, IREP and C4.5 algorithms. The results pointed out that RIPPER outperformed IREP on 28 data sets, whereas IREP outperformed RIPPER on only 7 occasions. In addition, RIPPER outperformed C4.5 on 20 data sets, whereas C4.5 achieved less error rate on 15 occasions.

2.5.7 Prism

Prism was developed by Cendrowska in [84] is a covering algorithm for constructing classification rules. The covering approach starts by taking one class among the available ones in the training data set, and then it seeks a way of covering all instances to that class, at the same time it excludes instances not belonging to that class. This approach usually tries to create rules with maximum accuracy by adding one condition to the current rule antecedent. At each stage, Prism chooses the condition that maximises the probability of the desired classification. The process of constructing a rule terminates as soon as a stopping condition is met. Once a rule is derived, Prism continues building rules for the current class until all instances associated with the class are covered. Once this happens, another class is selected, and so forth.

2.5.8 Hybrid Approach (PART)

Unlike the C4.5 and RIPPER techniques that operate in two phases, the PART algorithm generates rules one at a time by avoiding extensive pruning [88]. PART adopts separate-and-conquer to generate a set of rules and uses divide-and-conquer to build partial decision trees. PART avoids constructing a complete decision tree and builds partial decision trees as in C4.5. Also, in PART each rule corresponds to the leaf with the largest coverage in the partial decision tree. Missing values and pruning techniques are treated in the same way as C4.5.

Experimental tests using PART, RIPPER and C4.5 on different data sets from [103] have been reported in [88]. The results revealed that despite the simplicity of PART, it generates sets of rules, which are as accurate as C4.5 and more accurate (though larger) than those of RIPPER.

2.6 Associative Classification Mining

AC (**A**ssociative **C**lassification) mining utilises association rule discovery methods in the training step of classification. This approach was successfully used to build highly accurate classification models (i.e. [26][80][27][16][15]) in data mining and machine learning communities. Some of common algorithms in AC are CBA [13] and MCAR [104].

2.6.1 Associative Classification Problem and Common Solutions

Following the definition of [38] [105] for the AC problem. A training data set D has n distinct attributes A1, A2… An and C contains a list of classes. The number of cases in D is denoted |D|. A training case in T contains a mixture of attributes Ai and their values aij, plus a class cj. An attribute value can be described as a term name Ai and a value ai, denoted <(Ai, ai)>.

Definition 1: An AttributeValueSet is a set of disjoint attribute values contained in a training case, denoted < (Ai1, ai1), …, (Aik, aik)>.

Definition 2: A ruleitem r is of the form < AttributeValueSet, c>, where c□C is the class.

Definition 3: The support count (suppcount) of ruleitem r is the number of cases in D that match r's AttributeValueSet, and belong to the class c of r.

Definition 4: The frequency of an AttributeValueSet i (AVS_freq) is the number of cases in D that match i.

Definition5: A ruleitem r passes the MinSupp threshold if (suppcount(r)/|D|) ≥ minsupp.

Definition 6: A ruleitem r passes the minconf threshold if (suppcount(r)/ AVS_freq (r)) ≥ minconf.

Definition 7: Any ruleitem r that passes the minsupp threshold is said to be a frequent ruleitem.

Definition 8: A rule is represented in the form: $(A_{i1}, a_{i1}) \wedge \ldots \wedge (A_{ik}, a_{ik}) \rightarrow c$, where the antecedent (rule body) is an AttributeValueSet and the consequent (RHS) is a class.

A classification model is a mapping form $H : A \rightarrow C$, where A is the set of AttributeValueSet and C is the set of classes. The main task of AC is to find a classifier h ε H that maximises the probability that h (a) = c for each test case.

Unlike neural network and statistical and probabilistic based approaches, which normally produce classification models that are hard to understand or interpret by end-users, AC produces "IF-THEN" rules that are easy to understand and manipulate by end-users. This sub-section shed light on the solution scheme of AC and review some of its common algorithms.

The AC works as follow. First, all ruleitems that hold enough support values (ruleitem frequencies in the training data set above the MinSupp threshold are produced. Most of the current AC algorithms generate frequent ruleitems by making more than one scan over the training data set. In the first scan, they find the support frequency of 1-ruleitems (ruleitems consisting of a single attribute value), and then in each subsequent scan, they start with ruleitems found to be frequent in the previous scan in order to produce new possible frequent ruleitems involving more attribute values. In other words, frequent 1-ruleitems are used for the discovery of frequent 2-ruleitems, and frequent 2- ruleitems are the input for the discovery of frequent 3- ruleitems and so on.

Once all frequent ruleitems are discovered, their confidence values are computed and compared with the MinConf. When a ruleitem holds enough confidence (its confidence value is larger than or equal to the MinConf) then it will be produced as a rule. To cut down the number of rules generated most AC algorithms employ rule pruning procedures to discard redundant or noisy rules. Lastly, the most significant rules (those with high confidence and support) that survive the pruning phase will form the classifier that is later utilised to predict test cases. Each classifier must have a default rule which is applied when no other classifier rule is used. For example, with a MinSupp of 30%, the frequent 1-ruleitems in Table 1 are < (<AT1, z1)>, p1), (< AT2, w1>, p1) with support frequencies of 3/10 and 3/10, respectively.

2.6.2 CBA

The first AC algorithm is called CBA and was proposed in [13]. It consists of three main steps where in the first step any continuous attribute in the training data set gets discretised.

Step 2 involves frequent ruleitems discovery and rule generation. Then CBA selects high confidence rules to represent the classifier. Finally, to predict a test case, CBA applies the highest confidence rule whose body matches the test case. Experimental results designated that CBA derives higher quality classifiers with regards to accuracy than rule induction and decision tree classification approaches.

2.6.3 CPAR

A greedy AC algorithm called CPAR which employs FOIL-Gain in generating the rules from data sets was proposed in [27]. CPAR looks for the highest attribute value gain among the available attributes in the training data set to add it in a rule body. Once this attribute value is identified, the weights of the positive examples associated with it will be deteriorated by a multiplying factor, and the process will be repeated until all positive examples (examples that the rule body matches) in the training dataset are covered. In the rule generation process, CPAR produces not only the best attribute value but also all similar ones since there are often more than one attribute values with a similar gain. Results showed that CPAR improves the speed of the rule discovery process when compared with popular methods like CBA [13] and CMAR [26].

2.6.4 CACA

Another AC algorithm called CACA was proposed in [14], which first scans the training data set, stores data vertically like the MCAR algorithm, and then counts the frequency of every attribute value and sorts them in a descending manner according to their frequencies. All frequent disjoint attribute values' TIDs are intersected to reduce the search space of frequent patterns. A TID of a frequent attribute value holds the row numbers where these attribute values occur in the training data set. Lastly, for each attribute in a class group that passes the MinConf, it gets inserted in the Ordered Rule Tree (OR-Tree) as a path from the root node and its support, confidence and class are stored at the last node in the path. CACA classifies the unseen data like the CBA algorithm. Experimental results suggested that CACA performs better with reference to accuracy and computation time than other associative algorithms on the UCI data sets [103].

2.6.5 BCAR

Yoon and Lee [48]proposed an AC algorithm called BCAR which generates a large number of rules. BCAR prunes the derived rules using a Boosting-like approach [106]. This pruning method is a modification of the database coverage pruning of CBA (Liu et al., 1998). It has been claimed by the authors that the BCAR algorithm can be utilised in large-scale classification benchmarks like unstructured textual data. Experiments using various text collections showed that BCAR achieves a good prediction rate when compared with the Harmony classification approach [107].

2.6.6 MCAR

The MCAR algorithm introduced by [15] uses an intersection technique for discovering frequent ruleitems. MCAR consists of two main phases: Rule generation and a classifier builder. In the first phase, the training data set is scanned once to discover frequent 1-ruleitems, and then MCAR combines ruleitems generated to produce candidate ruleitems involving more attributes. Any ruleitem with support larger than MinSupp is created as a candidate rule. In the second phase, rules created are used to build a classifier by considering their effectiveness on the training data set. Only rules that cover a certain number of training cases are kept in the classifier.

The frequent ruleitems discovery method employed by MCAR scans the training data set to count the frequencies of 1-ruleitems, from which it determines those that hold enough support. During the scan, frequent 1-ruleitems are determined, and their occurrences in the training data (rowIds) are stored inside an array (TID list) in a vertical format. Also, classes and their frequencies are stored in the same way. Any ruleitem that fails to pass the support threshold is discarded. MCAR uses a function “Produce” to find frequent ruleitems of size k by appending disjoint frequent ruleitems of size k-1 and intersecting their rowIds. The result of a simple intersection between rowIds of two ruleitems gives a set which holds the rowIds where both ruleitems occur together in the training data. This set can be used to compute the support and confidence of the new ruleitem resulting from the intersection.

2.7 Issues in Classification

2.7.1 Overfitting

Overall, overfitting is considered one of the reasons why classification task in data mining is so hard [108]. Over fitting in classification occurs when performance of classifier increases on the training dataset while it deteriorates on the test dataset. Several reasons can cause the over fitting like limited number of training data objects or noise among the training objects [108]. Therefore, in decision trees , pruning approaches like pre-pruning and post-pruning [97] [25] have been widely used during building decision trees in order to avoid fitting the training data very well and to provide accurate performance on test data. Several methods used to evaluate the classifiers in a way that avoids over fitting effect such as cross-validation [91] and MDL principle [102].

2.7.2 Inductive Bias

An inductive bias can be defined as a set of assumptions that guide the selection of hypothesises (classification rules) [109] . Classification algorithms are able to generalise their performance on test data objects by inductive biases since they have implicit assumptions of favouring one rule over another. For instance, a decision tree algorithms like ID3 [92] and C5 [95] have a bias for the best attribute decision node increases information gain. Since classification algorithms have a bias, the resulting accuracy depends heavily on the training data features. Freitas [110] pointed out that when someone says algorithm X is better than algorithm Y, this should always be directed to the application domain used on the experiments.

2.8 Summary

This chapter presented association rule discovery and classification tasks in data mining. In the first part of the chapter gave a general overview on challenges in association rule mining and surveyed common association rule mining algorithms. The second part of the chapter discussed popular classification approaches such as decision trees, rule induction and probabilistic approaches. It concentrated particularly on classifiers based on association rules. At last it presented using MapReduce framework and its implementations in machine learning algorithms. Using MapReduce aware algorithms allows scaling the developed

algorithms to hundreds of machines and to process huge sizes of datasets. In addition, this chapter surveyed several simulators that maybe used to evaluate versions algorithms on distributed application environments. It addressed the need of simulators that particularly targets MapReduce environments due to the lack of such simulators currently.

Chapter 3

MRSim: MapReduce Simulator

3.1 Introduction

The primary cause to develop MapReduce simulator (MRSim) is the lack of general purpose MapReduce simulators that allow studying the behaviours and scalability of MapReduce algorithms on several on heterogeneous environments. MRSim introduced in this chapter is used later in chapters 4 and 6 to evaluate the scalability and hardware utilization of two new algorithms for mining association rules and for associative classification.

MRSim [35] aims to simulate Hadoop MapReduce implementation in order to evaluate the behaviour of later developed algorithms in chapters 4 and 5 on Hadoop clusters. MapReduce Hadoop has been around for a while and is open-source, feature rich, and the most widely used implementation among researchers and enterprises. The following description is how MRSim models and the MapReduce framework. Also several evaluations of MRSim are presented in this chapter.

3.2 MRSim: MapReduce Simulator for Apache Hadoop

Hadoop is a large distributed system platform of several hundreds of classes and hundreds of thousands of code lines. To simulate it the level of abstraction should be decided to reduce the complexity without losing the needed accuracy. MRSim adopted a layered structure design

(Figure 3-1). It starts from abstracting the work of CPU, Hard Drive, and network adapters in three entities CPU, HDD, and NetEnd as essential components used in shared resources in any distributed application. MRSim names these resources as "*core entities*" in MRSim. MRSim also abstracted the huge data sizes that are read, written, processed, or transferred over the network. Core MRSim entities do not simulate the data transformations resulted by each of these operations. It is the job of the application developer to consider this. For example, it is the map task to hold information of sizes and number of records passed to the "map()" function and resulted by it. MRSim processes uses the shared resources of CPU, HDD, NetEnd entities to estimate the times of such operations and to synchronize the task process with all other related processes in the cluster.

MRSim highly simulates dynamic systems where thousands of processes of different types are intended to share cluster resources concurrently .Thus, the problem of synchronization has to be addressed in the design. MRSim should be free from any risk condition that could arise between any running processes, especially the dependent processes. There were two options to solve the concurrency issue while designing MRSim. The first option was to simulate the processes using parallel threads or agent-based models, and then use the concurrency control methods provided by the programming language (e.g. using synchronized methods, lock objects, concurrent collections, thread pools, atomic variables, etc. in Java) and use other design methodologies to maintain the consistency of components operating in the system. The second option was that using discrete event simulation (DES) where operations in the system are represented as events and the DES ensures that events are always sorted in chronological order. Events can mark state changes in the system and are consumed in order by the system components resulting in the advance of the virtual simulation clock. MRSim adopted the discrete event simulation method.

3.2.1 MRSim Features

- Modelling of CPUs of different speeds and number of cores. CPU capability is defined by MIPS (Mega Instructions Per Second)
- Resources are modelled in time-shared mode.
- The design is layered and core functionality is defined as interfaces or abstract classes. Developers can implement or subclass new components to test new features in the system.

- Cluster resources can be heterogeneous.
- Application tasks can be heterogeneous and they can be CPU or I/O intensive.
- More than one application job can be submitted to the cluster at the same time. Multiple user entities can submit tasks for execution simultaneously to the same resource.
- The topology and network traffic between cluster nodes are specified and simulated using GridSim [8][40].
- Statistics of all or selected operations can be recorded and analyzed later.

3.2.2 System Architecture

MRSim component has clear interface that allows the other in-layer component and the components of upper layers to use it in formal way. Figure 3-1 shows MRSim components in layered design.

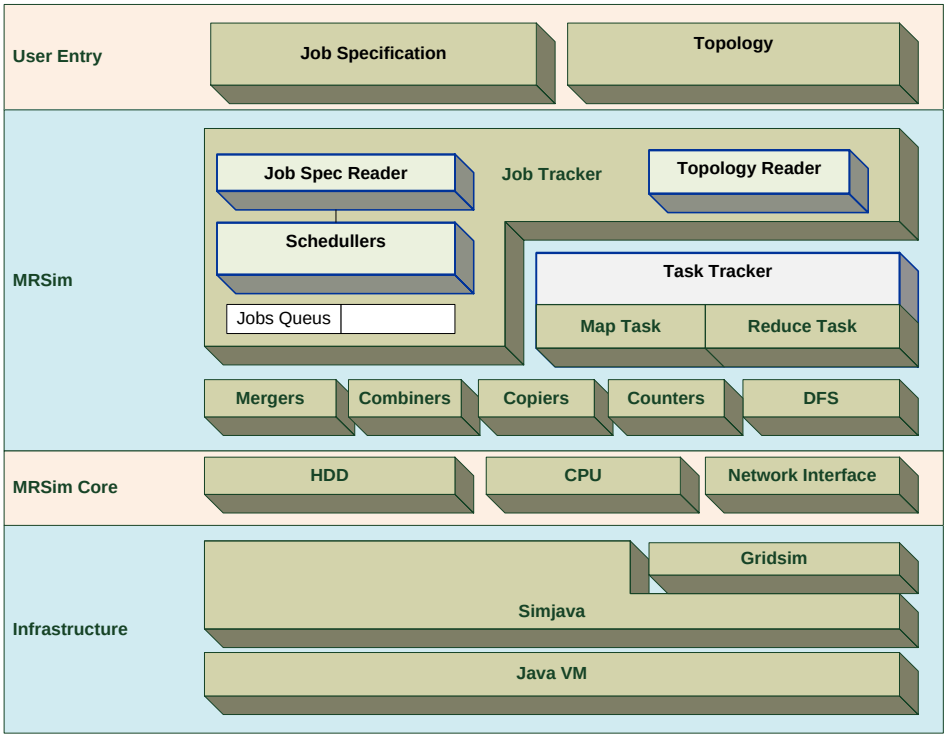


Figure 3-1: System architecture

3.2.2.1 MRSim Infrastructure

MRSim contains more than 20 thousand lines of code written in the Java programming language. System components are written using SimJava [39][67] which is a general-purpose discrete event simulation package implemented in Java. The simulation model in SimJava

contains a number of entities, each of which runs in parallel in its own thread. An entity's behaviour is encoded in Java using its "body()" method. Entities have access to a small number of Simulation primitives:

- sim_schedule() sends event objects to other entities via ports.
- sim_hold() holds for some simulation time.
- sim_wait() waits for an event object to arrive.

System entities communicate with each other by sending and receiving passive event objects efficiently.The sequential discrete event simulation algorithm in SimJava is as follows. A central object Sim_system maintains a timestamp ordered queue of future events. Initially all entities are created and their body() methods are put in run state. When an entity calls a simulation function, the Sim_system object halts that entity's thread and places an event on the future queue to signify processing the function. When all entities have halted, Sim_system pops the next event off the queue, advances the simulation time accordingly, and restarts entities as appropriate. This continues until no more events are generated. If the JVM supports native threads, then for all entities starting at exactly the same simulation time may run concurrently.

MRSim is using GridSim for network traffic simulation. GridSim was first preferable choice because it is built on SimJava and thus is easy to integrate into the system. MRSim – by using GridSim – is able to define the network topology of all the links between system entities. This includes defining the link type and baud rate for each node, defining routers used to interconnect the nodes, and defining routing schedulers and traffic type. MRSim abstracts the usage of the Gridsim.net package in its NetEnd core entity.

3.2.2.2 MRSim Core Entities

These are used to build other system entities in a composite way, or used by other system entities as shared resources. For example, a simulated cluster node is a group of one or more CPU, HDD, and NetEnd components. All other components which simulate computing functionality in the node must share the node's CPU, HDD, and NetEnd resources. Core entities must simulate with good accuracy the behaviour of shared resources used simultaneously by different processes in different applications. MRSim core entities focus on

the predicting – at certain point of the cluster run – how much time it will take to complete a CPU process estimated by Mega Instructions per second (MIPS), how much time to complete Hard Drive read/write operation of data sizes in MBs, and how much time to complete network transfer of MBs of data in a certain network topology. It is up to calling processes to decide how the data format is transformed after being processed by the core entity.

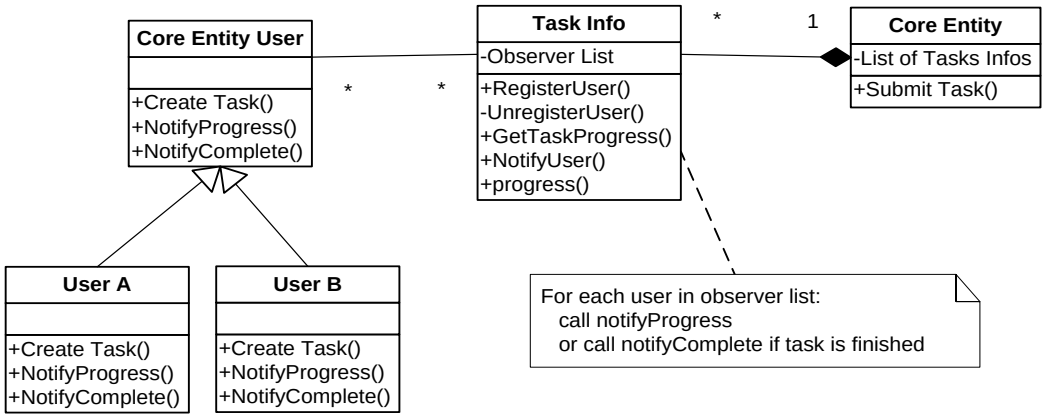


Figure 3-2: UML diagram of core entity that uses Observer Pattern

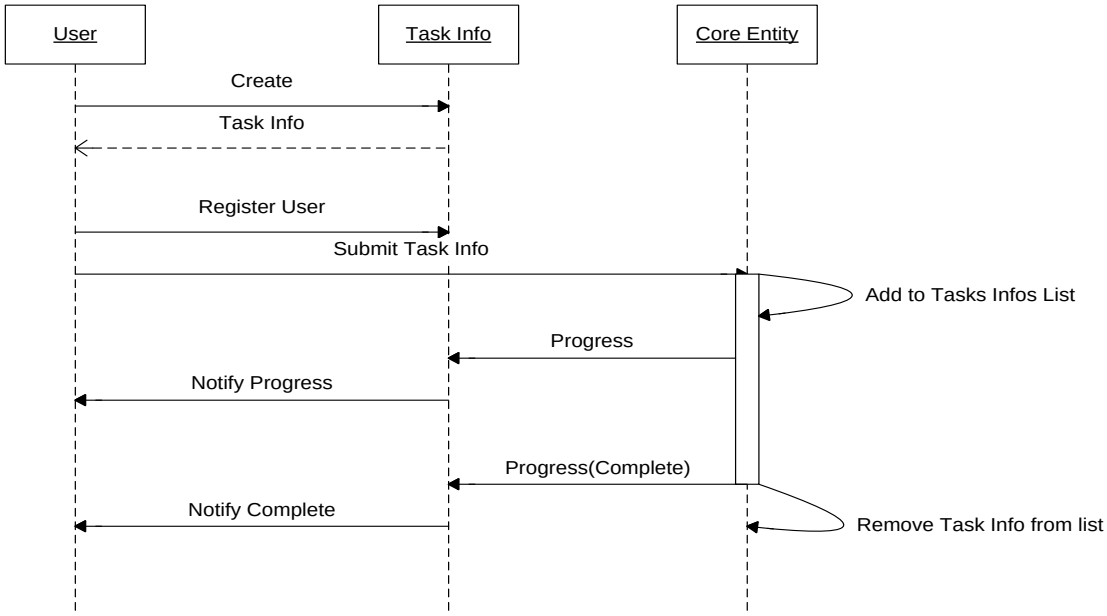


Figure 3-3: UML sequence diagram of observer pattern used in core entities

MRSim core entities also have the important role of synchronizing all system processes that use them. This is essential to get rid of risk of race conditions between dependent processes in the system. Synchronization at this level is much easier than trying to do it in upper levels. To achieve it, MRSim core entities implement observer patterns used in software design. An example of how a Core entity uses observer patterns let's take the CPU entity. The CPU entity allows a list of interested processes (observers) to subscribe to a certain CPU task, and then processes are notified automatically of the task state change (progress or completion).

Figure 3-2 and Figure 3-3 show the relation and the interaction between a core entity and core entity user (other entities in the system).

3.2.2.2.1 CPU Model

The CPU in MRSim comprises the following: number of cores (processors), speed of the processing, internal scheduling policy (currently time shared) and job done notification mechanism. The CPU is modelled using time shared mode, job scheduling uses a weighted Round Robin algorithm. CPU capability is defined in the form of MIPS (Millions Instructions per Second)

Table 3-1: Pseudo code for internal CPU scheduler

```
While simulation  is running:
   1. get next event ev
   2. if ev tag == "add new job" then:
            a.   append job to the job's exec queue
            b.   if at least one cpu core is available then
                 o  assign job to the idle cpu core
                 o  estimate the next event for cpu core, (job weight * core
                    time slot)
                 o  schedule local event ev at estimated time
                 o  continue
   3. if ev tag is local event then:
            get the attached job with the event "job"

            a.   update job status
            b.   if job is completed:
                 o  notify all registered users for this job "job"
                 o  if number of jobs in current exec queue > 1 then:
                        • remove first job in the queue "job_0"
                        • estimate the next event time for the cpu core
                          (job_0 weight * core time slot)
                        • schedule local event ev at estimated time for job
                          "job_0"
                 o  continue
            c.   else: //job is not completed
                 o  if number of jobs in current exec queue > 1 then:
                        • append "job" to the end to exec queue
                        • remove first job  "job_0" in the queue
                        • estimate the next event time for the cpu core
```

```
                        (job_0 weight * core time slot)
                    • schedule local event ev at estimated time
                o  else
                    • estimate the next event time for the cpu core
                        (job weight * core time slot)
                    • schedule local event ev at estimated time
                o  continue
                o
```

The scheduler assigns a fixed time unit per process (job), and cycles through jobs in the execution queue. Processes with more weights will have larger time units.

3.2.2.2.2 Hard Drive Model

The HDD in the MRSim has the following: average seeks time, the average speed for the write process and the average speed for the read process of the processing, internal scheduling policy (currently time shared) and read/write done notification mechanism. The HDD is modelled using time shared mode, job scheduling uses a weighted Round Robin algorithm. Read and write speed is always adjusted by a dynamic adjustment factor. HDD keeps track of the concurrent number of read and write processes running on it, and recalculates the adjustment factor on events of submission of a new job or completion of an existing job. The calculation of the adjustment factor is derived from experiments on real hard disks (Figure 3-23).

Here is pseudo-code summarises the internal HDD scheduler:

```
While simulation is running:
    1. get next event ev
    2. if ev tag == "add new read/write job" then:
            d.  append job to the jobs exec queue
            e.  if disk is idle then
                o  estimate the next event for HDD core, (job weight * core
                   time slot)
                o  schedule local event ev at estimated time
                o  adjust current read and write speeds of HDD
                o  continue
    3. if ev tag is a local event then:
            f.  get the attached job with the event "job"
            g.  update job status
            h.  if job is completed:
                o  notify all registered users for this job "job"
```

```
        o  if number of jobs in current exec queue > 1 then:
            • remove first job in the queue "job_0"
            • estimate the next event time for the HDD core
              (job_0 weight * core time slot)
            • schedule local event ev at estimated time for job
              "job_0"
            • adjust current read and write speeds of HDD
        o  continue
    i.  else: //job is not completed
        o  if number of jobs in current exec queue > 1 then:
            • append "job" to the end to exec queue
            • remove first job  "job_0" in the queue
            • estimate the next event time for the HDD core
              (job_0 weight * core time slot)
            • schedule local event ev at estimated time
        o  else
            • estimate the next event time for the HDD core
              (job weight * core time slot)
            • schedule local event ev at estimated time
        o  continue
```

Table 3-2: Pseudo code for internal HDD scheduler

3.2.2.2.3 Network Interface Model (NetEnd)

NetEnd abstract several classes used in GridSim.net package and support observer pattern as shown before. It has several methods to allow other system entities to send variance data sizes between two nodes in the network topology. NetEnd updates the registered users with the progress of current data transactions. In current GridSim implementation, network "link" entities does not support time shared mode. Only routers support sending network packet in time (or space) shared mode. This means processes running in one machine need to acquire the NetEnd resource before it can send data through it to different nodes. Thus, new transaction processes will not start until the current data transaction is completed. MRSim NetEnd entity added small extension to GridSim.net to allow several processes to send data through the GridSim in time shared mode.

3.2.2.3 MRSim Map-Reduce Entities

3.2.2.3.1 JobTracker

The Hadoop implementation and thus MRSim consists of a single master JobTracker and one slave TaskTracker per cluster-node. JobTracker is responsible for scheduling the jobs' component tasks on the TaskTrackers, monitoring them and re-executing the failed tasks. The TaskTrackers execute the tasks as directed by the JobTracker. Files are shared on the system using Hadoop distributed file system (HDFS).

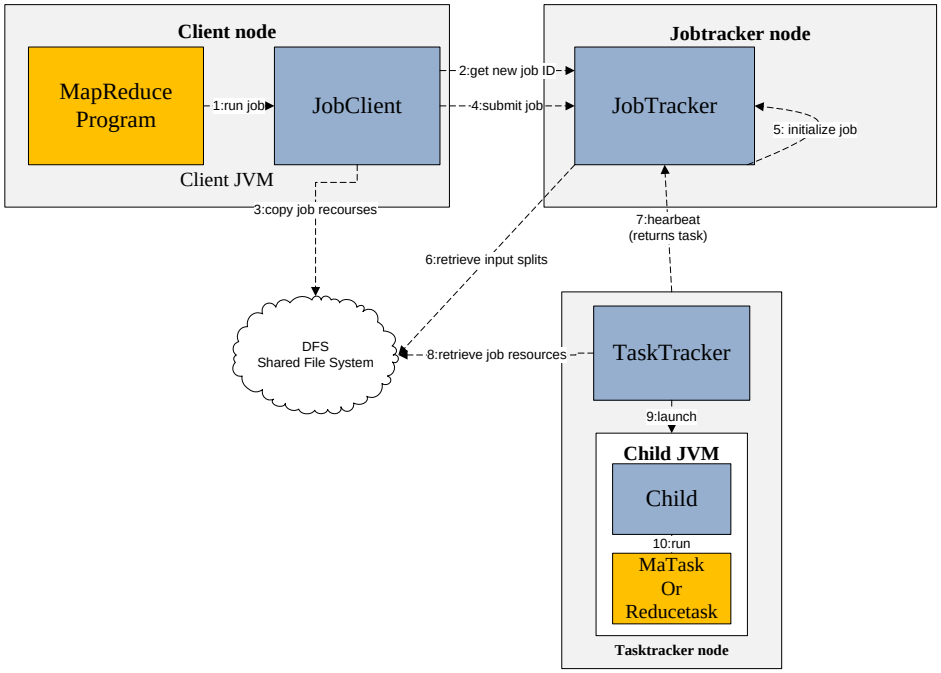


Figure 3-4: JobTracker in MRSim and Hadoop systems [36]

The process of running one job in Hadoop can be described in high level in Figure 3-4. Also, Figure 3-5 shows the workflow of between Hadoop main entities.

Job Submission: Client Asks the JobTracker for a new job ID, checks specifications of the job, calculate input splits for the job, and copy resources needed to run the job including job configuration file.

Job Initialization: Job Initialization: JobTracker it puts the submitted job into an internal waiting queue from where the job scheduler will pick it up and initialize it. Initialization involves creating an object to represent the job being run, retrieve the input splits computed by the Client. Then it creates one map task for each split. The number of reduce tasks to create is determined by job specifications. All map and reduce tasks are given Ids for tracking.

Task Assignment: A simple loop running periodically every "heartbreat" updates the JobTracker with the TaskTrackers' status. Furthermore, in every loop run, each TaskTracker will check if it is ready to run new tasks. Then the JobTraker will allocate the new tasks by using its assigned scheduler. TaskTrackers have fixed number of task slots for map tasks and for reduce tasks. This is defined in cluster configuration file. To achieve better performance, assigning map task needs more scheduling work to ensure data locality so the TaskTracker will be as close as possible to map input split. Assigning reduce task is simpler. The JobTracker simply takes the next waiting task in the queue and run it on the current available TaskTracker slot.

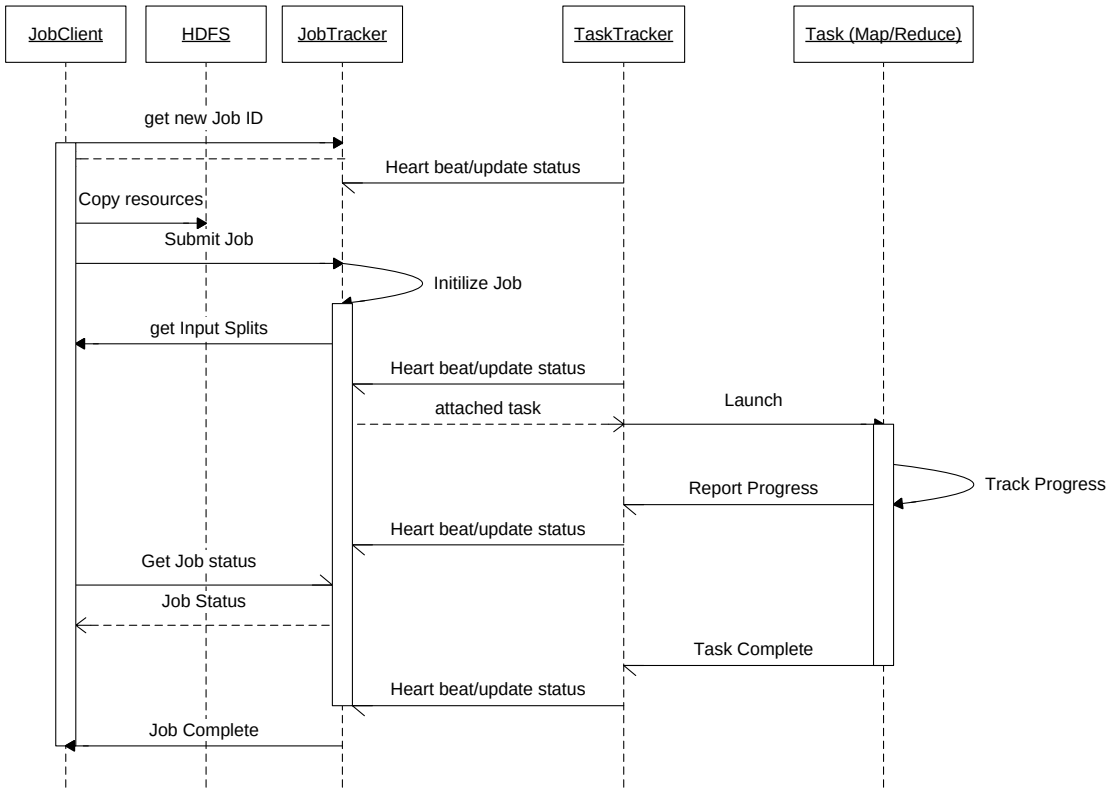


Figure 3-5: Workflow of JobTracker in Hadoop (and MRSim)

Task Execution: Now the TaskTracker has been assigned a task, it creates a local working directory for the task, in MRSim it create local log file to save task log messages. Hadoop TaskTracker will launch a new Java Virtual Machine to run each task. Similarly in MRSim, the TaskTracker will assign the task to SimJava entity to run it. Task's progress is reported every few seconds (in MRSim using simulated time) until the task is complete.

Figure 3-6: Flow control of JobTracker

Progress and status update: A job and each of its tasks have a status, which includes state of the job or task, the progress of maps and reduces, and the values of the job's or task's counters. The progress of map tasks is the proportion of the input that has been processed. The progress of reduce tasks, is divided to three phases: shuffle, sort and reduce. Tasks also have a set of counters that count various events as the task runs. The JobTracker combines these updates to produce a global view of the status of all the jobs being run and their tasks. Finally, the JobClient receives the latest status by polling the JobTracker.

Job Completion: When the last task for a job is completed, the JobTracker will changes the status for the job to indicate that is successful. When the JobClient polls for status, it learns that the job has completed successfully. In Hadoop, clients can be configured to receive callbacks by providing URL of returned call at the "job.end.notification.url" property. In MRSim, callback is implemented by providing the ID of JobClient, which is of type SimJava entity id. Finally, In Hadoop and MRSim, the JobTracker cleans up its working state for the job, remove it from running queues, and keep log history of the job its tasks on the file system.

3.2.2.3.2 Task Tracker:

Each machine at the cluster has at most one TaskTracker component. TaskTracker run tasks assigned by JobTracker master node and send progress reports back to it. In MRSim, TaskTracker has access to the machine resources of CPU HDD and NetEnd network adapter. All map/reduce tasks running in certain machine will share machine resources through the TaskTracker interface running on that machine.

3.2.2.3.3 Map Model

If data inputs are not divided by the user MRSim divide them into fixed-size pieces called "splits". MRSim – as in Hadoop – creates one map task for each split. MRSim simulates data locality optimization behaviour of the map by trying to run the map task on a node where the input split resides in DFS. However, some splits would have to be transferred across the network to the node running the map task. Map tasks generate intermediate output and write it to local HDD and not DFS.

When the map function starts producing output, the output is not simply written to disk. First it is buffered in memory buffer. Each map task has a circular memory buffer that it writes the output to. The buffer size is defined by the "ioSortMb" parameter in the job description. When the content of the buffer reaches a certain threshold size (also defined by the job description), it spills the contents to disk. Before it writes to disk, the map task first divides the data spill into partitions equal to the number of reducers. Within each partition, in memory the sort operation is performed, and if there is a combiner function, it is applied to

the output of the sort. After the map task has written its last output spill, there could be several spill files. The spill files are merged into a single partitioned and sorted output file. Combiner is applied again on the resulting file if it is defined in the job description. The configuration parameter "ioSortFactor" controls the maximum number of spills to be merged simultaneously. Compression of output data could be enabled by the job configuration. If enabled, the merged spill will be compressed before it is written to the HDD. This usually increases the performance of map tasks and reduces the task by shrinking the data sizes to be written to HDDs and to be transferred over the network. The output data are made available to the reducers over the network.

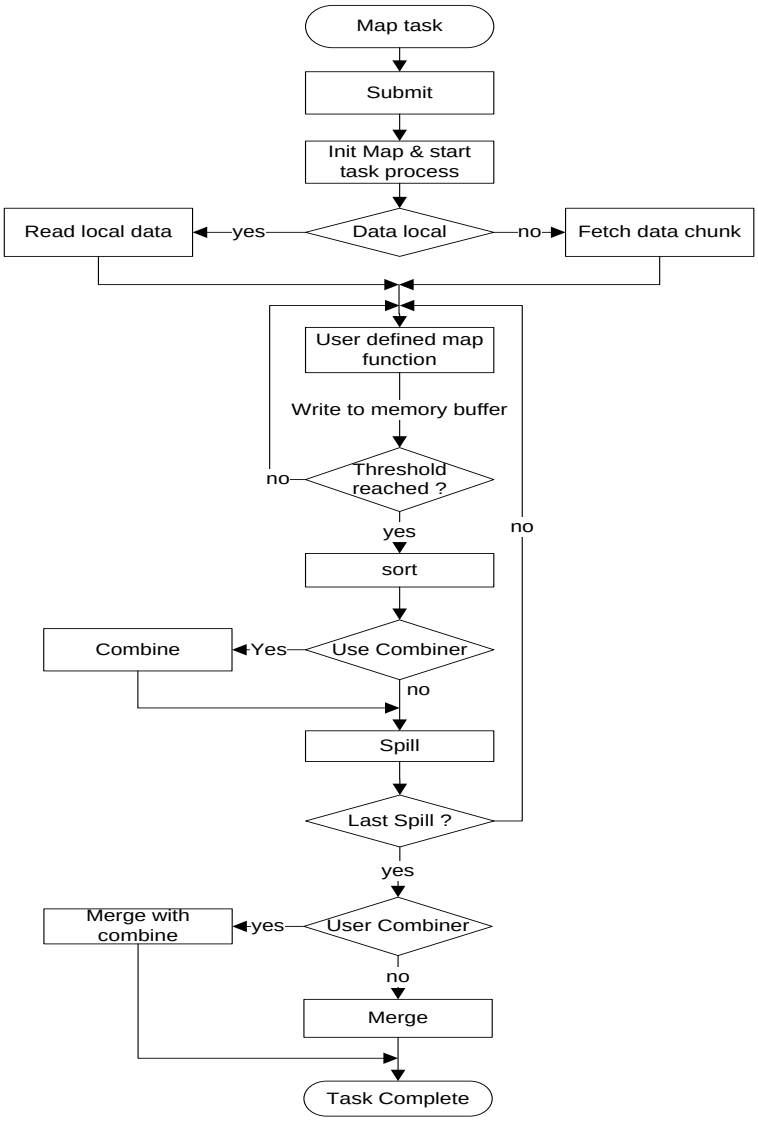


Figure 3-7: Flow control of Map task

Figure 3-7 summarizes the previous description of flow control of the map task.

Reduce Model

Figure 3-8 shows the Flow control or Reduce model in MRSim. Each reducer task normally gets its data input from the output of all Mappers. Thus usually there is no data locality in reduce tasks. The sorted map outputs are transferred across the network to the node where the reduce task is running. Then, input partitions are merged and passed to the user-defined reduce function. The output of reduce tasks is normally stored in a distributed file system. If there are multiple reducers, the map tasks divide their output, each creating one partition for each reduce task. The data flow between map and reduce tasks is known as "shuffle" as each reduce task is fed by many map tasks. The shuffle is an important phase where optimizing can have a large effect on job execution time. If there are zero reduce tasks, then map tasks write output data directly to DFS.

The shuffle phase is more complicated than described above. And it is important to model it in more detail to get a more accurate prediction of a Hadoop MapReduce cluster. The map tasks may finish at different times, so the reduce task starts "shuffling" their output partitions as soon as each map completes. This is also known as the copy phase of the reduce task. The reduce task has a small number of copier processes that fetch map outputs in parallel. This number is defined by the "mapredReduceParallelCopies" job description property. The map outputs are copied to the reduce buffer memory if they are small enough. Also the buffer's size is defined by the "mapredJobShuffleInputBufferPercent" property, which specifies the proportion of the memory heap to use for this purpose. If the map outputs are not very small, they are copied to disk. When the in-memory buffer reaches a threshold size (also defined in "mapredJobShuffleMergePercent"), or reaches a threshold number of map outputs (defined by "mapredInmemMergeThreshold"), it is merged and spilled to disk. There is also a background process that merges the spills into larger files.

When all the map outputs have been copied, the reduce task moves into the "sort phase". In the sort phase, the merging process keeps merging maps' output to larger ones and keeps the data sorted. The maximum files that can by merged at once are defined by the merge factor (ioSortFactor property).The merging process runs rounds of merges till it completes merging whole map outputs fetched to the reducer. The final merge can come from a mixture of data in-memory and data on-disk. In the last round that merges the resulting files, the merger directly feeds the reducer with the data. The reducer at this stage is in the "reduce phase",

where the user-defined reduce function is called for each key in the sorted output. The output of this phase is written directly to the output DFS.

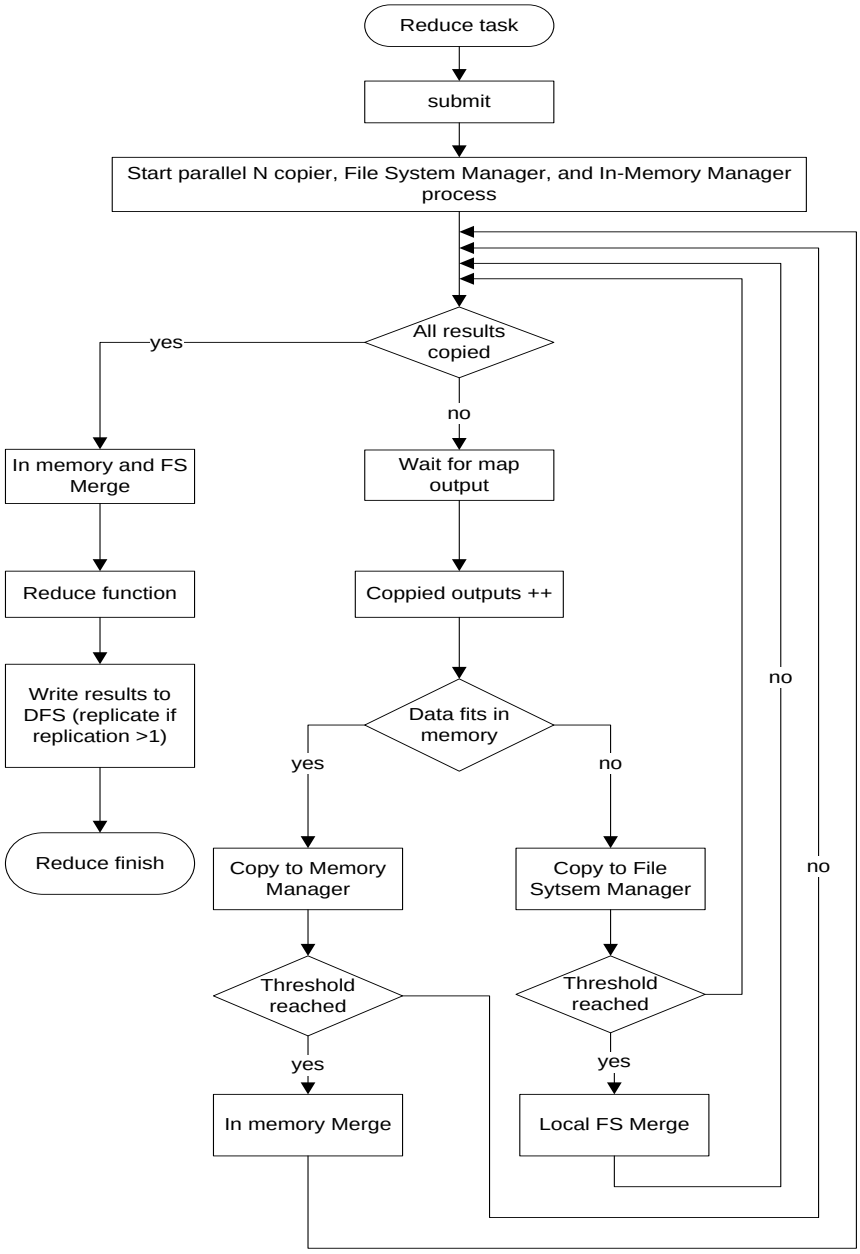


Figure 3-8: Flow control of Reduce Task

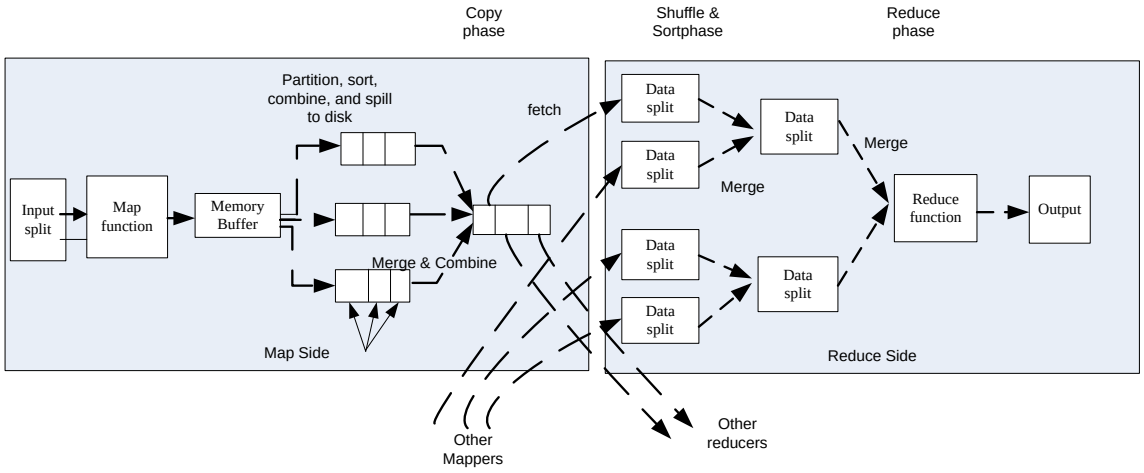


Figure 3-9: Hadoop data flow [36]

3.2.2.3.4 Combiner Model

The combiner function is run on the map output data buffered in memory, and sorted by the keys. Combiners may be run repeatedly over the input because there could be one or more data spills generated by the map task. Combiners do not affect the final result. Running combiners makes for a more compact map output, so there is less data to write to local disk and to transfer to the Reducers. Usually the combiner uses the same or similar code to the Reducer code because combiners can be used when the reduce function is mathematically aggregated. MRSim simulates the combiner behaviour in Hadoop as follows: MRSim tries to predict the key distribution in each output spill generated by the map task, and tries to predict the key distribution in spills resulting from merging spills previously combined more than once. Figure 3-10 shows the data flow of merging the map outputs of 10,000 records each, without using the combiner function.

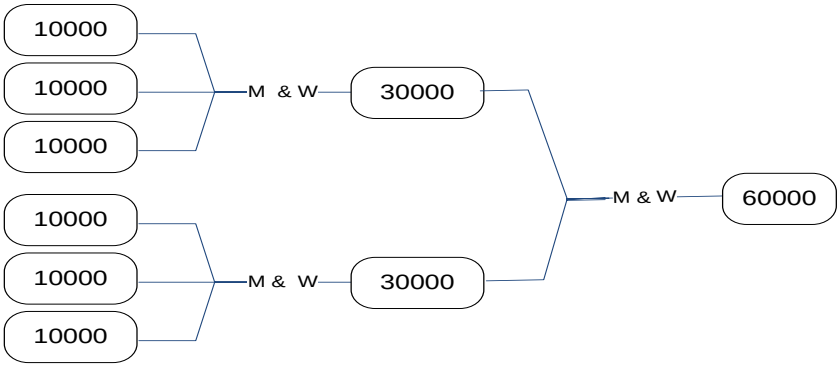


Figure 3-10: Spill writing and merging (M& W: Merge & write to file system)

$$spilled\ records = 6 \times 10000 + 2 \times 30000 + 1 \times 60000 = \ 180{,}000$$

$$read\ records = 6 \times 10000 + \ 2 \times 30000 = \ 120{,}000$$

Suppose that the keys at the map output can be grouped in 100,000 groups equally distributed. Then, MRSim can predict the merge with combine behaviour mathematically. Figure 3-11 shows using the combiner in the merge process.

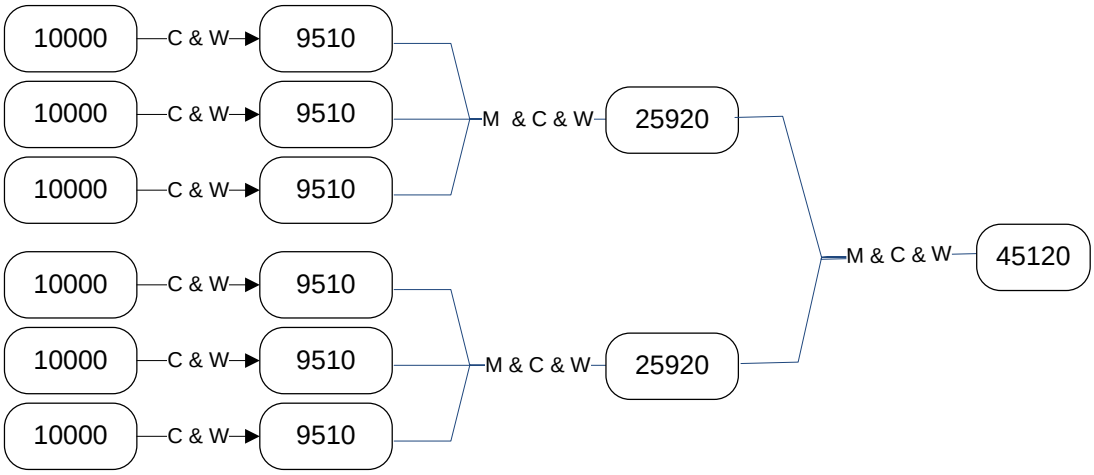


Figure 3-11: Dataflow using combiner on map outputs (C: combine, M: Merge, W write to file system)

$$spilled\ records = 6 \times 9510 + 2 \times 25920 + 1 \times 45120 = \ 154{,}020$$

$$read\ records = 6 \times 9510 + 2 \times 25920 = 108{,}900$$

Using the combiner in this example reduced the file system operations to the value:

Reduction of writings (spilled records) = 14.43 %

Reduction of readings =9.25 %

Depending on the number of output groups of keys the reduction could reach high values such as more than 99% in the k-means clustering algorithm used in [54].In the following is how MRSim mathematically calculate the records reduction when using the combiner:

This is a problem of computing the approximate probability that in a set of n records, at least two records have the same key. Suppose that the total number of possible keys is g. MRSim assumes that the g keys are equally likely. Real-life key distributions are not uniform since it depends on the input data. If $p(A)$ is the probability of an event that at least two records in

the spill have the same key, it may be simpler to calculate $p(A')$, the probability of no two records having the same key. $P(A)$ and $p(A')$ are the only two possibilities and are also mutually exclusive.

$$P(A) = 1 - p(A')$$

$p(A')$ is the probability of having n records with unique keys. $P(A')$ can be described as n independent events.

$$P(A') = p(1) \times p(2) \times ..\times p(n)$$

The n independent events correspond to the n records, and are defined in order. Each event can be defined as the corresponding record not sharing its key with any of the previously chosen records. For event 1, there is no previously chosen record. Record number 1 does not share its key with a previously chosen record Therefore,

$$p(1) = \frac{g}{g} = 1$$

For event 2, the only previously chosen record is record 1. The probability, $p(2)$, that record 2 has a different key than record 1 is : $p(2) = \frac{g-1}{g}$

Similarly, $p(3) = \frac{g-2}{g}$

Continue until record n

$$p(n) = \frac{g - n + 1}{g} \text{ , } n < g$$

$p(A')$ is equal to the product of these individual probabilities:

$$p(A') = \frac{g}{g} \times \frac{g-1}{g} \times ....\times \frac{g-n+1}{g}$$

$$p(A') = \frac{1}{g^n} \times \left(g \ \ (g-1) \times (g-2) \times ..\times (g-n+1) \right) = \frac{g!}{g^n\,(g-n)!}$$

Formula (2) calculates the probability of the nth record not sharing the same key as any of the n – 1 preceding records. The event of at least two of the n records having the same keys is complementary to all n keys being different.

$$P(A) = 1 - p(A')$$

Assume $f(n)$ is a combiner function which calculates how many unique keys in sample S of number of records equal to n, given that the total number of unique keys in the system is g. Now, add one more random record to the sample S. The probability that this record is unique is: $\frac{g-f(n)}{g} = 1 - \frac{f(n)}{g}$

The new expected number of unique keys in the sample is given as:

$$f(n+1) = f(n) + 1 \times \frac{g-f(n)}{g},\ f(n+1) = 1 + \frac{(g-1)}{g} f(n)$$

If $\alpha = \frac{g-1}{g}$ then: $f(n+1) = 1 + a.f(n)$

$$f(n+1) = 1 + a(1 + a.f(n-1))$$

$$f(n+1) = 1 + a\left(1 + a\left(1 + a\left(\ldots\ldots a(1 + a.f(1))\right)\right)\right), \qquad \text{where } f(1) = 1$$

$$f(n+1) = 1 + a + a^2 + a^3 + \cdots + a^n$$

$$f(n+1) = \sum_{i=0}^{n} a^i = \frac{1 - a^{n+1}}{1 - a}$$

$$f(n) = \frac{1 - a^n}{1 - a}, \qquad a = \frac{g-1}{g}$$

Formula (1)

This is the equation MRSim uses to estimate the number of unique keys in the subset of the data generated by partial output of the map task. Thus, MRSim can estimate how many records are reduced in each spill in each merge wave using the combine function.

Another approximation seems to have good accuracy in the current range of tests is approximation using a continuous exponential variable. Assume the maximum number of unique keys g is much greater than 1. Then MRSim can approximate that the probability density function of previous f(x) at Sample S of size x is

$$f'(x) = 1 - \frac{f(x)}{g}$$

$$g.f'(x) + f(x) - g = 0$$

This is linear differential equation. The solution is:

$$f(x) = g(1 - e^{\frac{-x}{g}})$$

Formula (2)

3.2.2.3.5 DSF Distributed File System:

MRSim focused mainly on MapReduce components. The DSF implementation is limited and very simple. If the MRSim user needs to extend some feature e.g. new policy for fetching splits for maps, then he/she needs to take care of all DSF details such as in which machines the actual replicas of splits exist, and which split replica is to be fed to the map. Writing operations in the Distributed file system in MRSim are simulated simply by storing the first replica on the local node, the other replicas are stored other nodes on and off the rack. Thus, writing to the distributed file system does consume network bandwidth. The replication factor is defined in the cluster configuration.

3.2.2.4 MRSim user level and Input Specifications

MRSim takes two types of inputs: hardware/topology specifications, and job characteristics. Input files are specified in JSON format [111], and are read by JSON-processor. The JSON processor provides seamless conversion between JSON format and POJO (Plain Old Java Object) based either on property accessor conventions or annotations. This data binding is similar to the DOM XML Tree Model. But the JSON content is converted to regular Java objects rather than the node-based model in DOM XML. MRSim uses simple data binding that only uses standard JDK container types of Lists, Maps and scalar types such as String, Boolean, Number, and nulls.

3.2.2.4.1 Topology /hardware specifications

Figure 3-12 shows a sample of a hardware/topology file of a rack of machines linked by one router:

```
{ "machines" : [


       { "baudRate" : 70000000.0,
            "cpu" : { "cores" : 4,
                "speed" : 500000000.0
              },
            "hardDisk" : { "capacity"
: 40000.0,
                "read" : 40000000.0,
                "seekTime" : 1.0,
                "write" : 20000000.0
              },
            "hostName" : "m1",


            "maxMapper" : 10,
            "maxReducer" : 5
       }
,
{ "baudRate" : 70000000.0,
            "cpu" : { "cores" : 4,
                "speed" : 500000000.0
              },
                          "hardDisk" : {
"capacity" : 40000.0,
                "read" : 40000000.0,
                "seekTime" : 1.0,
                "write" : 20000000.0
              },
       "hostName" : "m2",


            "maxMapper" : 10,
            "maxReducer" : 5
       }
       ],
"router" : "r_01",
      "heartbeat":1.0,
      "propDelay":1.0,
      "maxIM":60000,
      "deltaCPU":1000000.0,
      "deltaHDD":1000000.0,
      "deltaNEt":1000000.0,
      "flowType":false,
      "hlogLevel":"info"
     }
```

Figure 3-12: Hardware/Topology Input file

Currently Topology input can support only a simple one-rack network, and a tree of racks network. However, in principle MRSim is able to support many types of topology because the underlined network simulator (GridSim) can support a user-defined topology. In simple rack topology the cluster consists of several machines connected in on LAN linked by one

router. The distance between any two machines is equal to 1 router. In the tree rack topology, a cluster consists of several simple racks, which are connected using another router. Hardware specifications and Objects contained in each rack are shown in Figure 3-13:

Network adapter	**"baudRate"** : 70000000.0
cpu	{ **"cores"**:num_cpus, **"speed"** :mega_instruction_per_second}
hardDisk	{ **"capacity"** : 40000.0,**"read"** : 40000000.0, **"seekTime"** : 1.0, **"write"** : 2000000.0 }
Machine	{ **baudRate** : 70000000.0, **"cpu"**:{}, **hardDisk**:{} **"maxMapper"**:mapper_capacity,**"maxReducer"**: educer_capacity}
Rack	{**"machines"**:[list of machines], **"heartbeat"**:heartbeat_period, …. Cluster_step_parameters }

Figure 3-13: JSON object in Topology file

3.2.2.4.2 Job Configuration Input file

MRSim defines several parameters to describe the job characteristics. Job parameters can be grouped into three categories as shown in Figure 3-14 :

```
{
"jobName":"job_01",
"numberOfMappers":60,
"numberOfReducers":1,
"useCombiner":false,
"useCompression":false,


"ioSortFactor":10,
"ioSortMb":100.0,
"ioSortRecordPercent":0.05,
"ioSortSpillPercent":0.8,
"mapredChildJavaOpts":200,
"mapredInmemMergeThreshold":1000,
"mapredJobReduceInputBufferPercent":0.0,
"mapredJobShuffleInputBufferPercent":0.7,
```

Task Configuration Parameters

"mapredJobShuffleMergePercent":0.66, "mapReduceParallelCopies":5, "useCombiner":false, "useCompression":false "replication":3, …	
"data":{ "name":"data_1", "size":4.78627929E8, "records":1620000.0, "replica":["machine 1","machine 5"] }, "inputSplits":[], "outputSplits":[],	Job Data layout

"algorithm":{
"mapCost":10000.0, "mapRecords":50.0,
"mapOutAvRecordSize":12.0,
"combineCost":80.0, "combineRecords":1.0,
"combineGroups":100000.0,
"combineOutAvRecordSize":1.0,
"reduceCost":80.0 , "reduceRecords":0.01,
"reduceOutAvRecordSize":10.9,
….
….
},
}
	Job Algorithm Parameters

Figure 3-14 Job description File

Task configuration parameters: These parameters are used to override default job parameters. These parameters are derived from Hadoop most important job configurations. Usually they describe the amount of resources allocated for processes executing the job such as sizes of memory buffers used in Mappers and Reducers, the thresholds used to control data flows between the memory and local hard drives, the number of parallel processes running for a task, such as in memory or hard drive mergers, and map results fetchers. A full description of these parameters is listed in [3]. Task configuration parameters also help in predicting the overhead needed to initialize and finalize each task or job.

Data layout parameters: point to the locations, replicas, number of records, and sizes of data chuck for a single job. Data layout determines the data locality in each map task. The JobTracker's default scheduler tries to allocate a Mapper task on machines containing a replica of the data chunk to decrease the amount of data to be copied to map tasks, and thus make the map phase of the job faster.

Algorithm parameters: Jobs consist of map and reduce tasks. In each job the user implements the "map()", "reduce()", and "combine()" functions. All single jobs have this data sequence in the system:

(*input_data*) -> map_function -> (*intermediate data*) -> combine_function -> (*intermediate data*) ->reduce_function -> (*output_data*)

The algorithm part describes for each of the "map()", "reduce()", "combine()" functions:

- The cost of processing input data unit (Instruction per byte).
- The conversion of sizes between the input and the output.
- The conversion of the number of records between the input and outputs.
- The number of unique keys expected in the combine() and reduce() functions.

These are the minimum number of parameters needed to describe, with good accuracy, certain job behaviour. Providing algorithm parameters combined with the job configuration parameters allows simulating the lower sub-processes in each map and reduce tasks, such as: initializing the task, determine how many times the memory buffered spilled to the local hard drives, configuring mergers, sorters, and data fetcher over network. Also, they determine the

interaction between job data (initial, intermediate, and output data) with the hardware resources. This allows calculating the sizes and times needed to process each data unit in different job execution stages.

3.3 Differences between MRSim and MRPerf Simulators

MRSim shares a lot of features as in MRPerf [10] [61] and is designed for the same purpose of studying the behaviour of MapReduce jobs running with different job descriptions and in different cluster configurations. However, there are main differences which triggered the developing of MRSim:

- In MRPerf, only a few parameters are available to the user to configure the cluster settings. Other important parameters either do not exist or are fixed in the simulator's code and cannot be altered by the user.
- The current release of MRPerf code seems suitable for few algorithms such as for sort and indexing algorithms. The user can specify the number of CPU cycles per byte for sorting and merging tasks. This is a useful parameter to describe the computation needed in the task because data in a MapReduce job undertakes several sorting and merging steps whatever the algorithm is. However, a general purpose MapReduce simulator should allow the user to specify – in addition to sorting and merging – the number of CPU cycles per byte for the map() and reduce() functions in order to simulate algorithms other than sorting, especially algorithms of high computations cost in map() and reduce() functions.
- Experiments showed that – on average MapReduce jobs – hard disk I/O operations are usually the bottleneck in the system. I/O read and writes speed is affected dramatically by the concurrent number of read and write processes on the disk. This might be the main cause for less accurate results in MRPerf and in MRSim simulators. Using accurate models for the disk is highly recommended. G. Wang et al. in [10] suggested using DiskSim [112] but they did not use it. DiskSim still has drawbacks as it simulates old IDE hard disks. Most new hard disks nowadays have SATA interfaces. Because of lack of an accurate open-source hard disk modelling tool so MRSim used a simple hard disk model in MRSim simulator. MRPerf also uses a simplified disk model based on average I/O read and write speed. However, MRSim's model is more accurate as it used real benchmark that considers the effect

of the number of I/O processes running in parallel on the same disk (shown later in Figure 3-23).

- MRPerf does not overlap I/O and CPU operations assigned to a map/reduce task. It divides these operations into distinct sequential phases. This is done to simplify the design at the cost of some accuracy. However, Using Discrete-event simulation in MRSim allows us to truly simulate overlapping I/O and CPU operations without scarifying the simplicity.

3.4 Validating MRSim

Tow levels of validation are carried out. One is to validate the design of core entities and scheduler policies used in them. Core entities worth separate evaluation because they are used heavily by all other system components. The second validation is to validate the overall system.

3.4.1 Design Validation

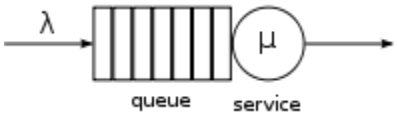
3.4.1.1 MRSim CPU validation

Several experiments were carried out to validate the CPU module. The experiment application uses a Java Thread pool of various sizes to process jobs arriving and waiting in a queue. Job arrival interval-time and CPU hour cost for a job are generated using random normal distribution. The application ran on Intel Core Duo processor T7300 CPU. Source code used is available at [41]. Each experiment generates statistics on the CPU while processing hundreds of arriving jobs. Experiments are carried out with different parallel threads capacity assigned to the CPU each time. Before showing the results, a few definitions presented in the result figures are explained:

- Submit time T_{submit} : is the job arrival time, or the time of entering the waiting queue.
- Start time T_{start} : is the time when the job is moved to the execution queue, where it is being served by the CPU cores using a Weighted Round Robin scheduleller.
- Stop timeT_{stop} : is the time when the job is completed and removed from the exec queue.

- Job execution time T_{job} : the time it took the job to be processed on the CPU. $T_{job} = T_{stop} - T_{start}$
- Waiting time T_{wait} : $T_{wait} = T_{start} - T_{submit}$
- Turnaround Time $T_{turnaround}$: $T_{turnaround} = T_{stop} - T_{submit}$

In the special case with a CPU of single core, the problem can be modeled mathematically using as single-server queue. Then if experiment run with Poisson distribution for both job arrival times and for job CPU hour cost, and with endless capacity queue buffer, then the problem can be modelled by the M/M/1 single-server queue model.



If λ is arrival rate, μ is job execution time rate, $\rho = \frac{\lambda}{\mu}$ then Expected waited time in queue is: $w = \frac{\rho}{\mu - \lambda}$. For a number of cores in the CPU greater than one, the mathematical solution is complicated. The M/M/C queue model does not fit in this case, because jobs are run concurrently on the same CPU using time shared (May need to explain more). In this case the evaluation of the CPU model is done by comparing results of simulation with the results of experiments.

- CPU utilization is predicted by the simulator. However, the results are not presented here because of lack of tools that can measure the CPU utilization of real experiments.
- Average Throughput: number of jobs that complete their execution per time unit. (Figure 3-15 and Figure 3-16).

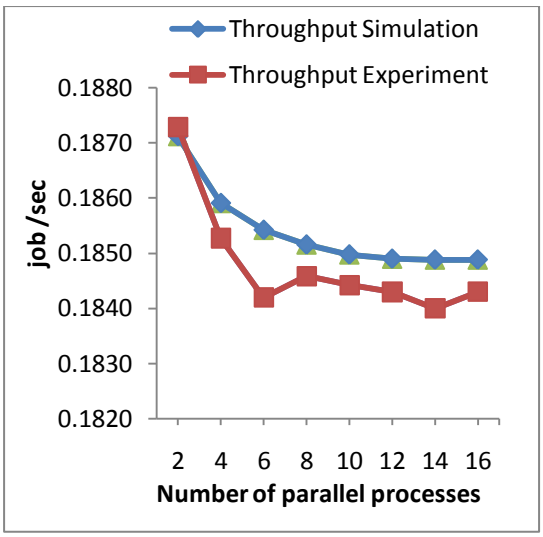


Figure 3-15: Average Job throughput vs number of maximum of parallel processes

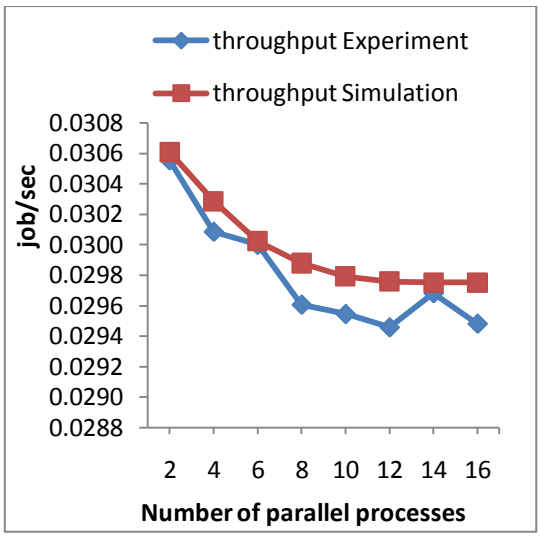


Figure 3-16: Standard deviation for job throughput vs. number of maximum parallel processes on CPU

- Average Turnaround Time: total time between submission of a job and its completion. (Figure 3-17 and Figure 3-18)

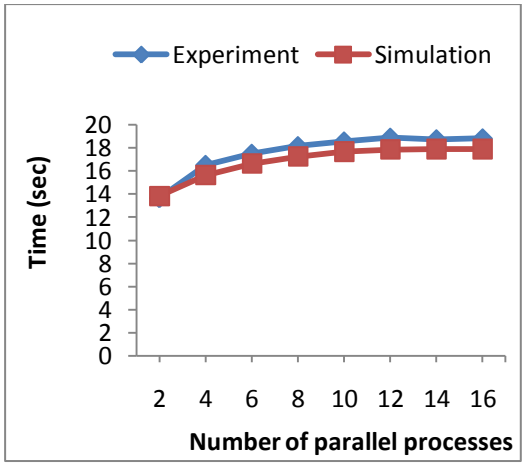


Figure 3-17: Average Job turnaround time vs. number of maximum parallel processes running on CPU

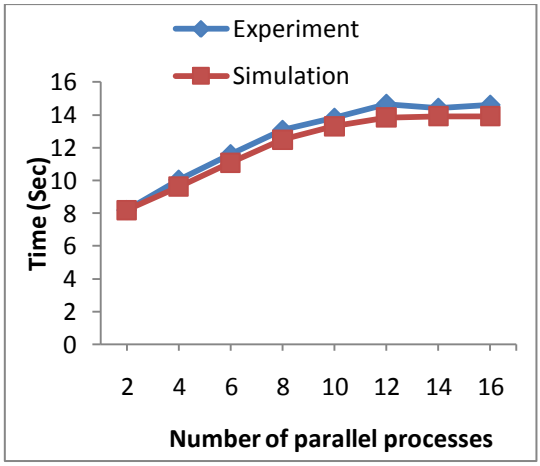


Figure 3-18 : Standard deviation for job turnaround time vs. number of maximum parallel processes

- Average waiting time: amount of time between submission of the job and the start of execution.(Figure 3-19 and Figure 3-20)

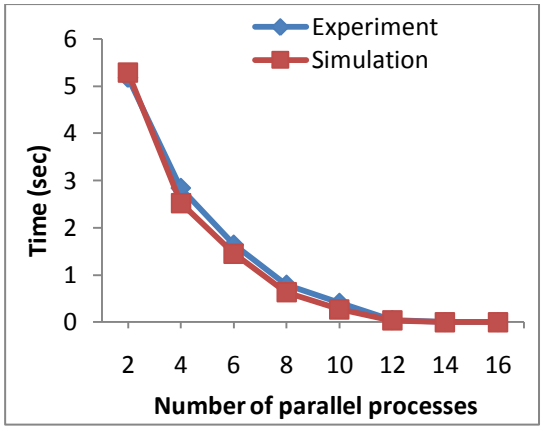


Figure 3-19: Average Job waiting time vs. number of maximum parallel processes

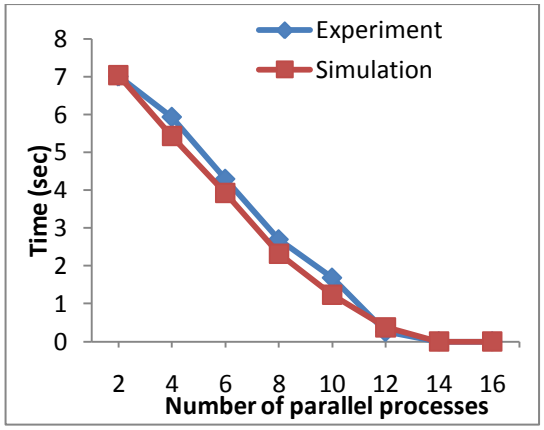


Figure 3-20: Standard deviation of job waiting time vs. number of maximum parallel processes

- Average Response time: amount of time it takes from when a request was submitted until the first response is produced. This is very close to the average waiting time in the queue before starting executing. It only differs with one time slice used in the Round robin scheduler.
- Average Job Execution Time: Time between starting executing the job and the time of completion. This time increases when the maximum number of allowed parallel processes on the CPU increases. (Figure 3-21 and Figure 3-22)

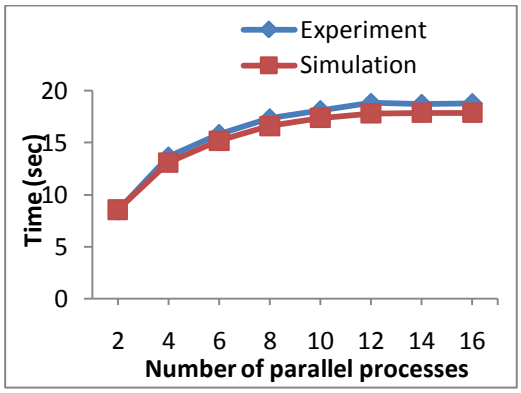


Figure 3-21: Average Job execution times vs. number of maximum parallel processes allowed on CPU

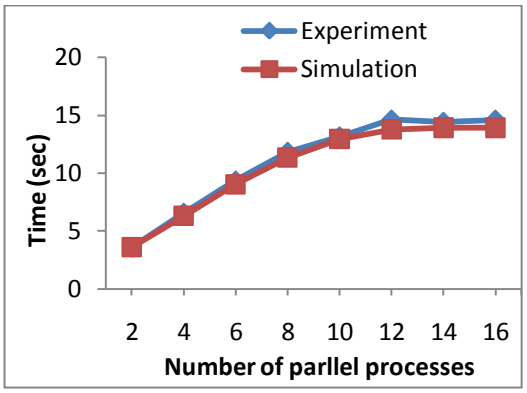


Figure 3-22: Standard deviation of job execution time vs. number of maximum parrallel processes allowed on CPU

- Fairness: The jobs submitted in the experiments all have same priority. In this case, the round robin scheduler ensures equal CPU time to each thread. With different job priorities there is a possibility of process starvation in WRR. Although the CPU model in MRSim supports different job priorities, other entities in the system submit

the jobs with the same default priority. Thus, no experiments are needed to validate the fairness in the scheduler of MRSim CPU entity.

3.4.1.2 MRSim Hard Drive validation:

Hard drive accurate simulation is the key to an accurate MapReduce simulator because MapReduce jobs usually process large sizes of data and the amount of intermediate data produced while executing are of large sizes too. Also mergers work recursively on data splits to merge them into one file. This means more than one read/write operation for the same intermediate data. This is why a Hadoop map reduce cluster is usually sensitive to hard disk performance. In production clusters, machines are provided with several hard disks to increase the overall I/O speed. To the best of our knowledge, DiskSim [112] is the most accurate open source available tool for disk simulation. However, using DiskSim require configuring hundreds of parameters. Although DiskSim offers tool to automatically extract the Hard Disk parameters, the new SATA Interface disks are not fully supported and not all parameters are available to the simulator.
As an alternative, different method is used by simulating the hard disk with few parameters: Capacity, Access time, and Read/Write speed. The CPU model of average speed with weighted round robin scheduler is also adopted for hard disk model. However, unlike the CPU which has stable processing speed, the average read/write speed of hard disks changes dynamically depending on the concurrent number of read/write processes using the disk. Without DiskSim-like tools, it cannot decide dynamically the current average read/write speed. To solve this problem MRSim adopted hybrid approach. Hard disk model uses a CPU-like model for the Hard Disk and then added a dynamic speed adjustment functionality to the model. Adjustment functionality data is collected from real experiments on the hard disks used in the cluster. This means, for each type of hard disk in the system, at least one experiment is needed to extract the parameters for adjustment functionality. Usually clusters are composed of tens or hundreds of homogeneous machines. So the number of experiments needed is small. Here the GUI Linux tool is used to measure the hard drive performance for one types of hard drive. As shown in Figure 3-23 the average access time and average read/write speed vary depending on the concurrent number of read/write processes on the hard drive.

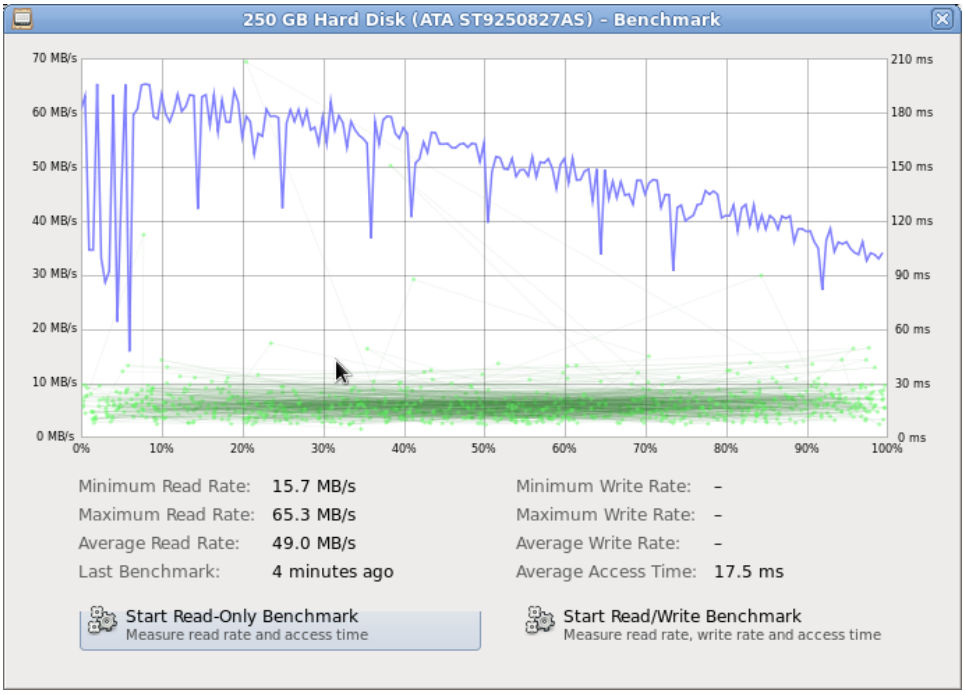


Figure 3-23: Read benchmark

3.4.1.3 MRSim Network Traffic Validation

MRSim uses GridSim [8] [113] [40] for network traffic simulation. Packet level or flow-level simulation can be used in GridSim. However, using packet-level simulation takes considerable time to complete the simulation. More accurate network simulators may be used instead of GridSim. Any candidate alternative should implement the NetEnd interface defined in MRSim. The network simulator ns-2 [11] is a good candidate to replace the GridSim network simulator in future development because it has been used for while by the research community and more research is being done on it to ensure its accuracy, such as [68]. Also, ns-2 has call back functionality which allows implementing a MRSim NetEnd interface.

3.4.1.4 MRSim Combiner Function Validation

The Java application was written to generate sample subsets of varying sizes from a set of keys of size g (g=100,000 in the experiment). Each subset is generated by random choice with return sampling. Then the number of unique keys in the subset sample is counted. The experiment was repeated 1000 times for each sample subset and repeated the whole process for 50 sample sets of sizes ranging from 10,000 to 490,000. **Error! Reference source not found.** and **Error! Reference source not found.** are used to predict the number of unique

keys in each sample. Figure 3-24 shows the result of using combine functions of various sets of data.

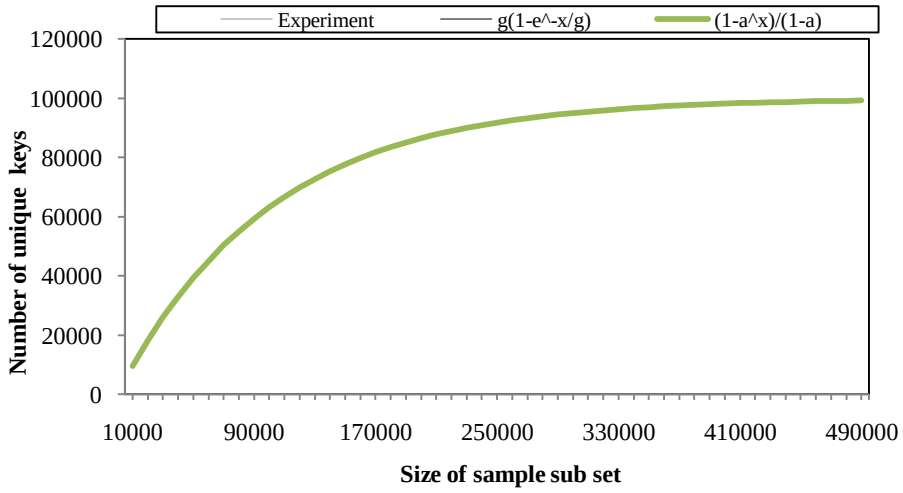


Figure 3-24: Number of unique keys in sample subset vs. size of sample subset

This is a very accurate result. With a confidence level of 95%, the confidence interval for data generated by **Error! Reference source not found.** is ±5.604 and by **Error! Reference source not found.** is ±5.606. This means in test samples of sizes of 10,000 and more, 95% of the results are within a range of less than 0.01% of the expected values. However, in different algorithms, the map task may generate entries with keys not equally distributed among the range of keys. **Error! Reference source not found.** and **Error! Reference source not found.** will not generate such accurate results. In the current experiments of algorithms used in this research, the keys are – with good confidence- equally distributed and thus equations 1 and 2 can be used to estimate the scalability of algorithm when the combiner function is enabled.

3.4.2 Local Cluster Experiment Validation

The following experimental results were collected in a single rack cluster, consisting of four participating nodes. Three of the nodes are Intel CPU Q6600, 3GB RAM and Fedora 12 OS, the fourth one Intel Core 2 Duo T7300, 4GB RAM and Fedora 12 OS. The experiments focused on job execution times, average task times in every job, intermediate spilled records in Map and Reduce tasks and the number of HDD read and writes for each job. Datasets used with different sizes are used. The algorithm used in the test is word count. Each job consists of 60 map tasks and one reduce task. Each job is tested three times with three configurations;

configuration one is the default configuration. Configuration two is by using combiner tasks. Configuration three is by using more virtual memory for each of MapReduce tasks. There are many more configurations that can be tested and is altered from job description file Figure 3-17 . The prefix "s-" indicates simulation result where other legends used without "s-" prefix indicate real cluster result.

3.4.2.1 Spilled Records (local hard disk writes)

- **Without combiner**

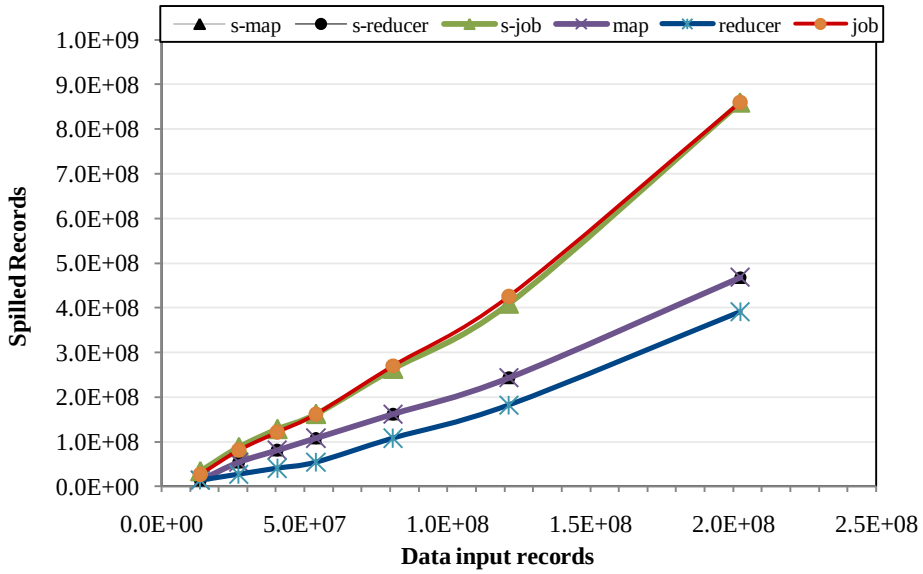


Figure 3-25: Intermediate spilled records to local file system vs. input records

- **With combiner**

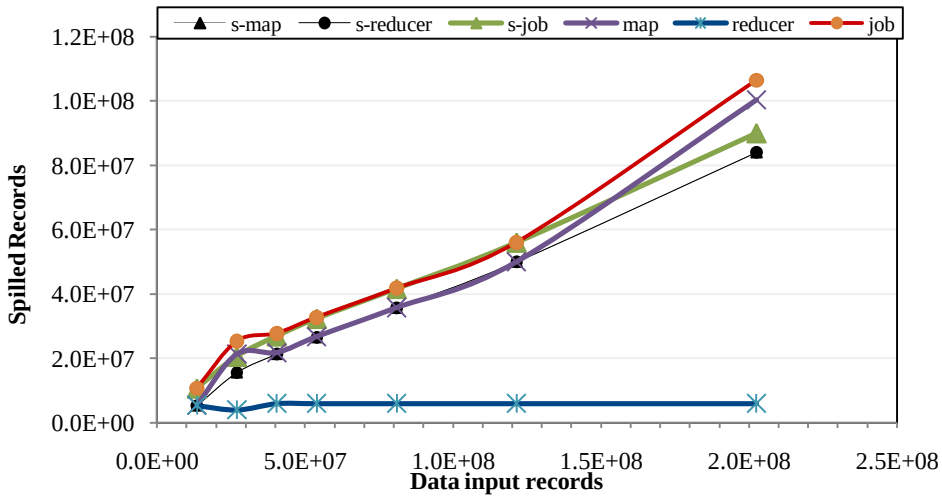


Figure 3-26: Intermediate spilled records to local file system vs. input records, using combiner function

- **More virtual machine per task**

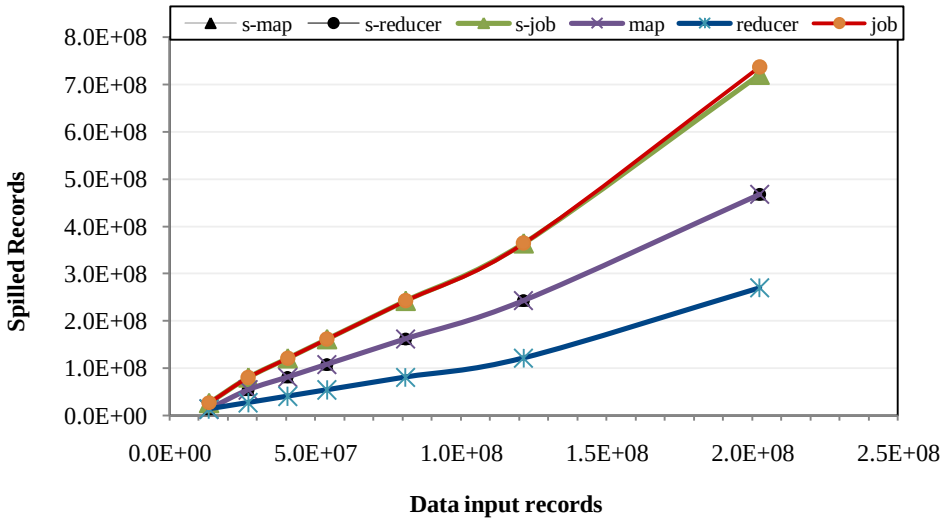


Figure 3-27: Intermediate spilled records to local file system vs. input records, using double virtual memory per task

3.4.2.2 Local Hard Disk reads and writes

- **Without combiner**

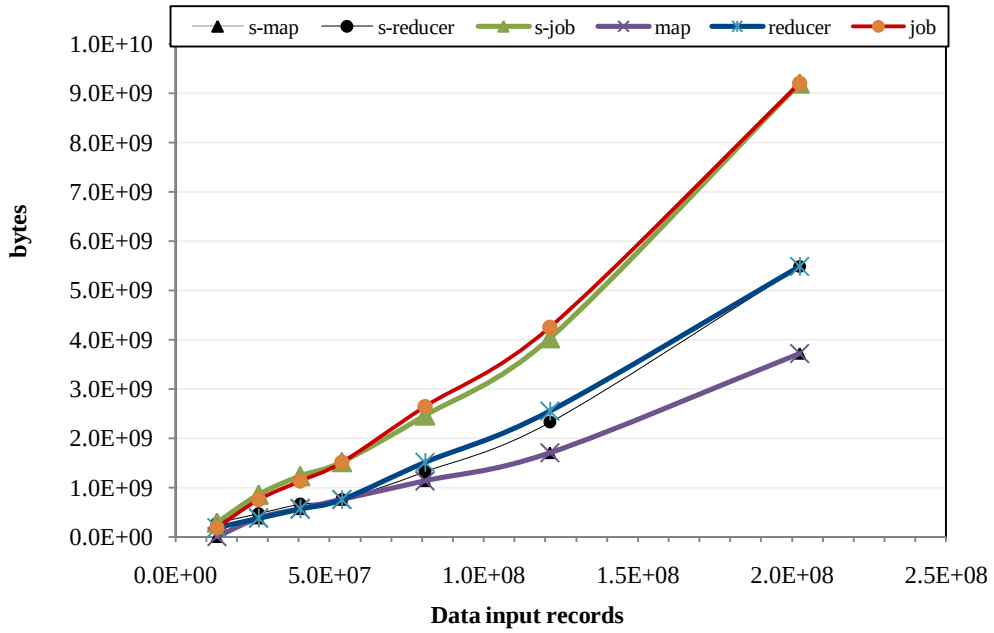


Figure 3-28: Local file system read bytes vs. input records

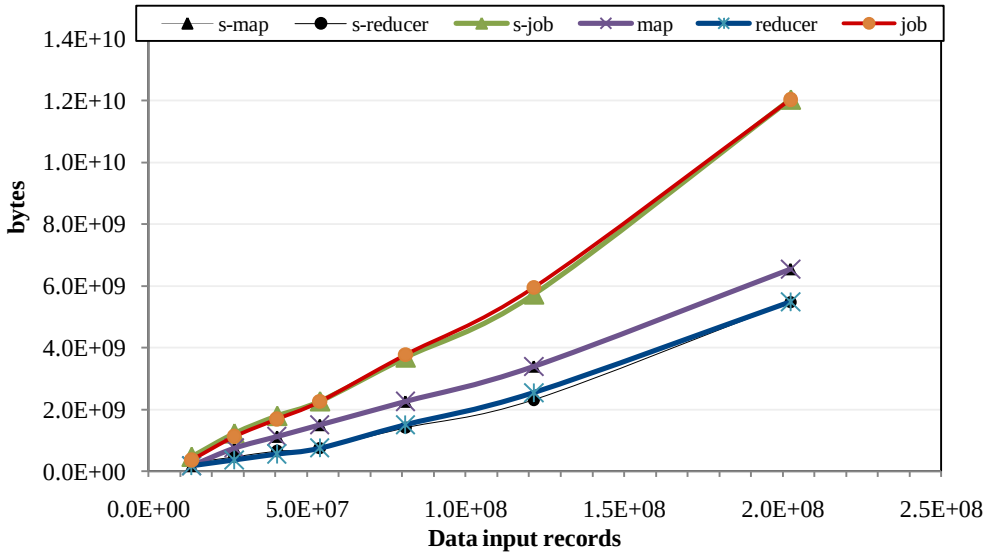


Figure 3-29: Local file system written bytes vs. input records

- **With combiner**

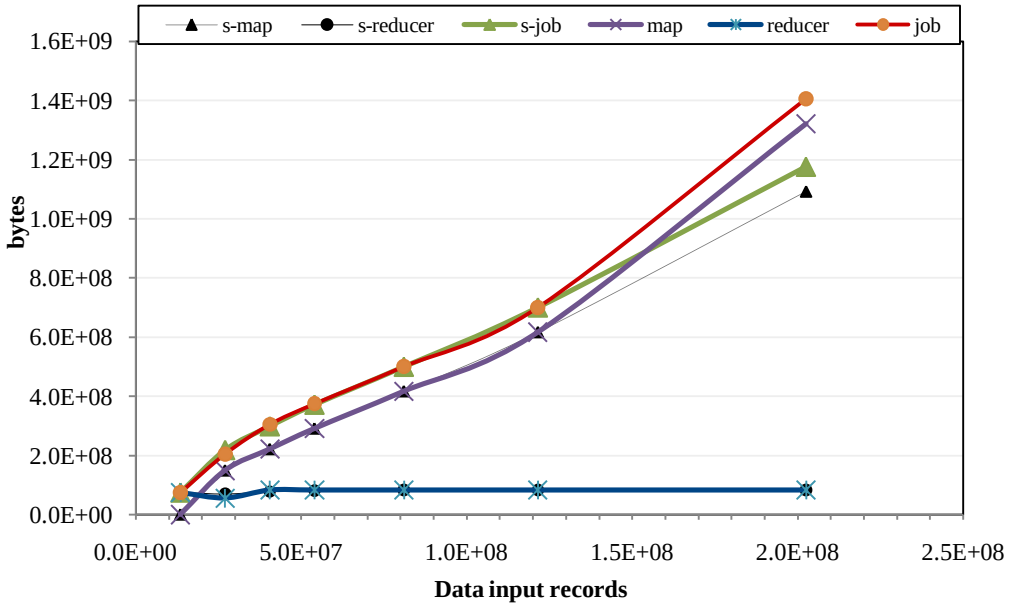


Figure 3-30: Local file system read bytes vs. input records, using combiner function

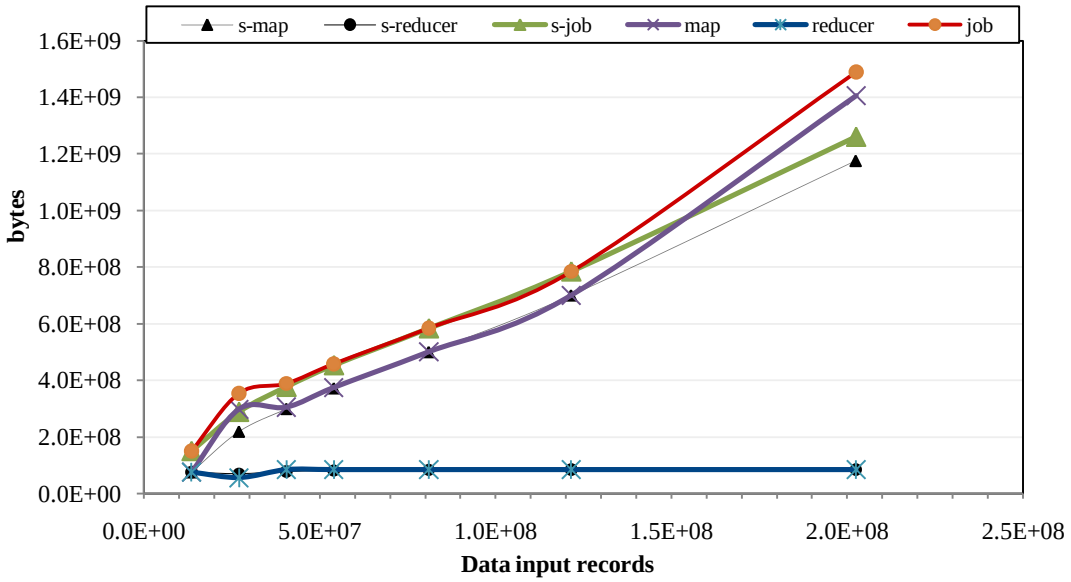


Figure 3-31: Local file system written bytes vs. input records, using combiner function

- **More virtual machine per task**

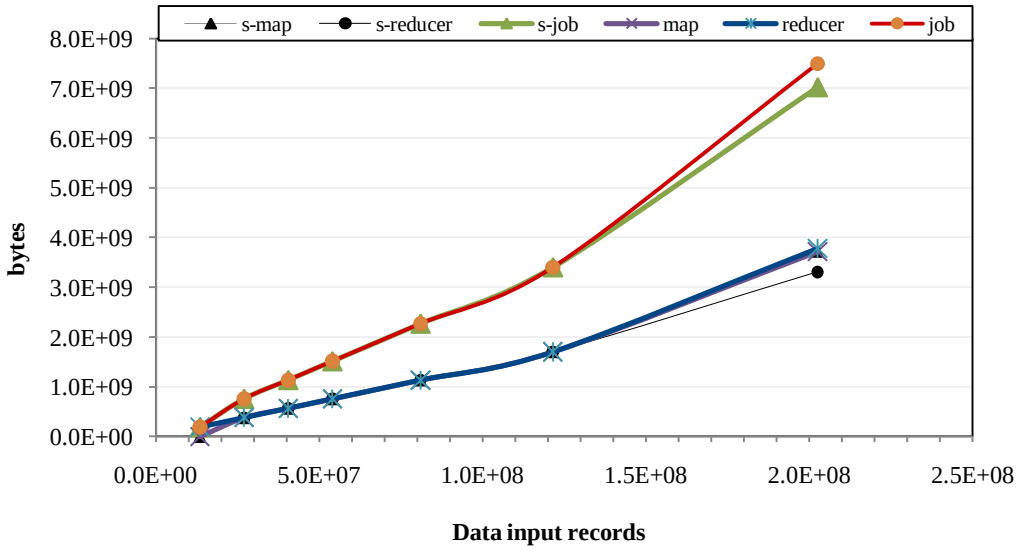


Figure 3-32: Local file system read bytes vs. input records, using double virtual memory per task

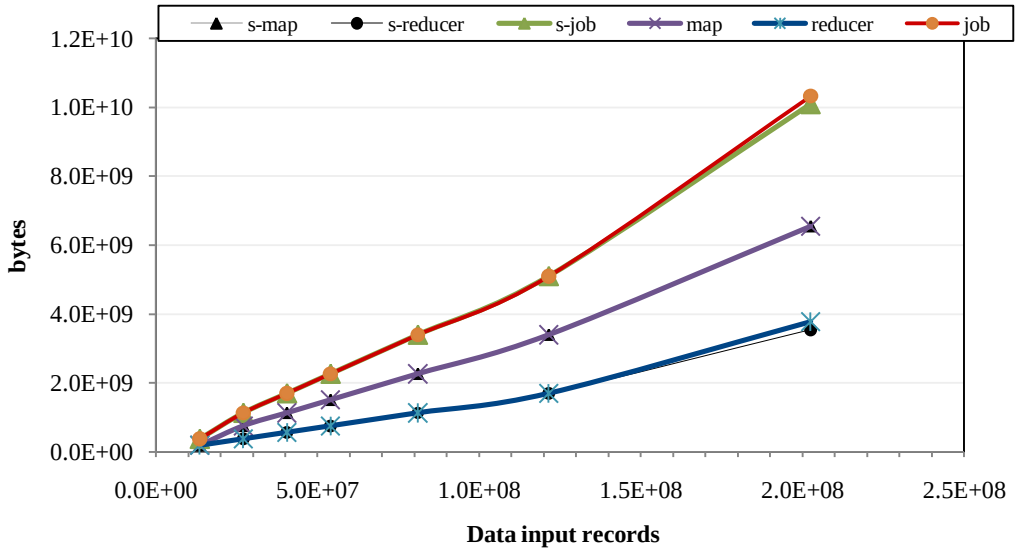


Figure 3-33: Local file system written bytes vs. input records, using double virtual memory per task

It is obvious that the sizes of intermediate data generated between the mappers and reduces is greatly affected by the job configurations. Using combiner in job submitted results in applying an aggregate mathematical operation of groups of records in-memory before writing the intermediate data to local hard disks. Such reduction is shown between Figure 3-25 and Figure 3-26 for example. On the other hand, using more virtual memory for processes tasks also reduces the sizes of intermediate data and I/O operations used for the job as shown between Figure 3-25 and Figure 3-27 for example. However, this reduction in sizes is not as much as reduction resulted from using combiners Figure 3-26. Using more virtual memory for MapReduce tasks reduces the I/O operations between CPUs and local hard drives because data are merged in memory and rather than on hard disks. Same argument applies for all tests of spilled records, local read sizes, and local write sizes. Figures of these categories usually have the same trends (ex. Figure 3-25,Figure 3-28, and Figure 3-33). However, there is no definite consistent pattern between those measurements and thus, MRSim monitor them separately.

3.4.2.3 Mappers, Reducers, and Job Execution Times

As shown in figures Figure 3-34, Figure 3-35, Figure 3-36 mappers consumes less time from the total job time. However, this is not a rule and is affected greatly by the algorithm definitions in job description file Figure 3-14. Figures show that using combiner greatly reduced times of jobs. Also, there are slightly better execution times when using more virtual

memory for MapReduce tasks. However, this is not general case and is tightly dependant on algorithm and job parameters in job description file.

- **Without combiner**

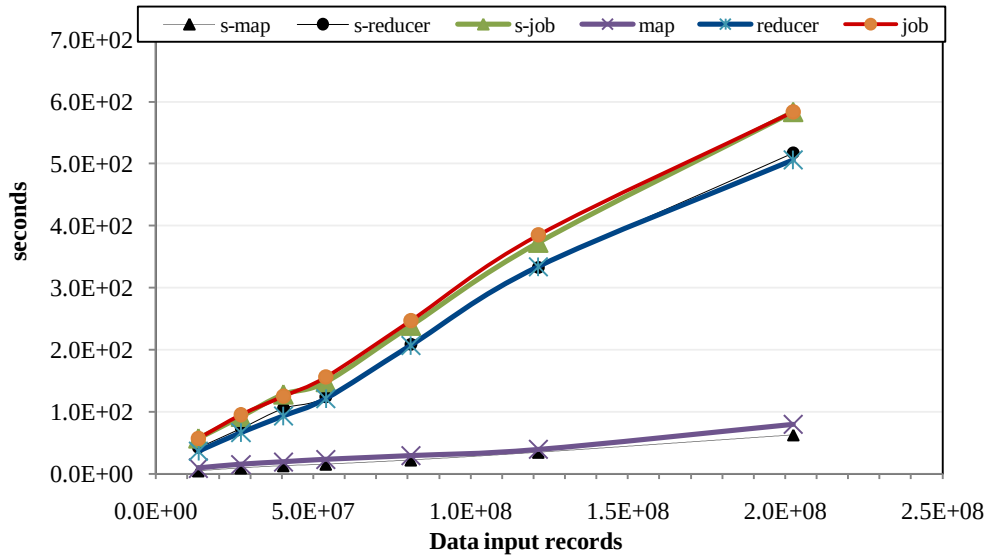


Figure 3-34: Execution times vs. input records

- **With combiner**

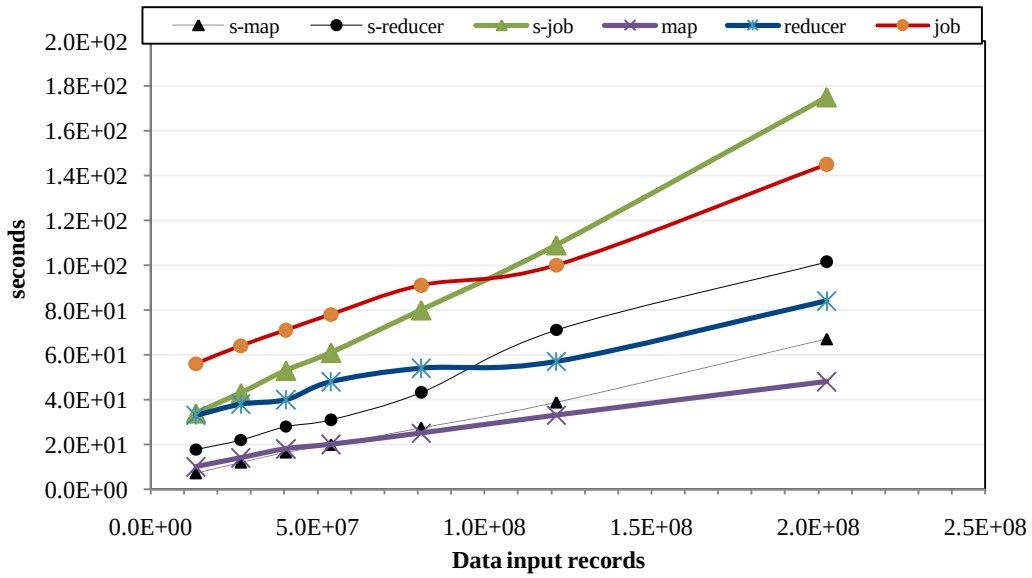


Figure 3-35: Execution times vs. input records, using combiner functions

- **More virtual machine per task**

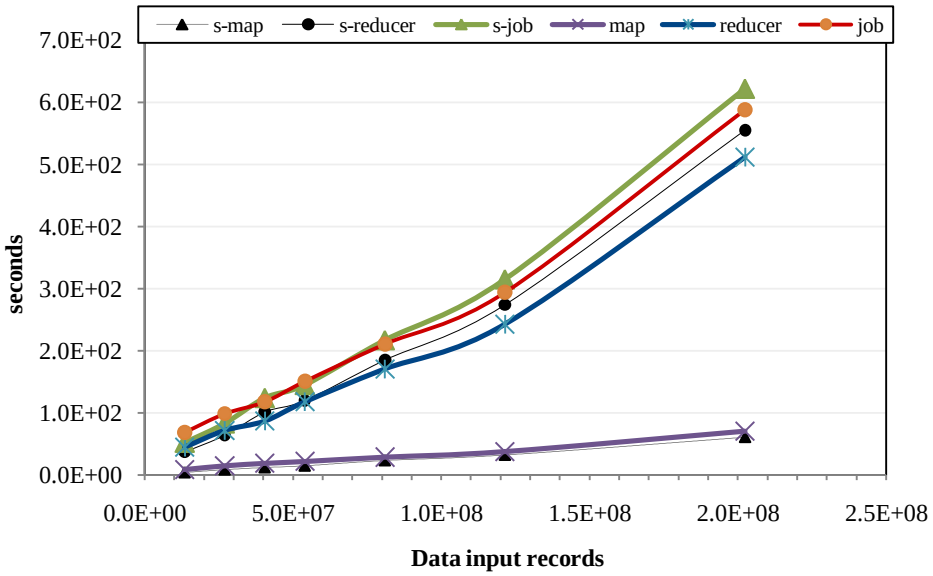


Figure 3-36: Execution times vs. input records, using double virtual memory per task

3.4.2.4 Data Shuffled Over the Network Between Mappers and Reducers

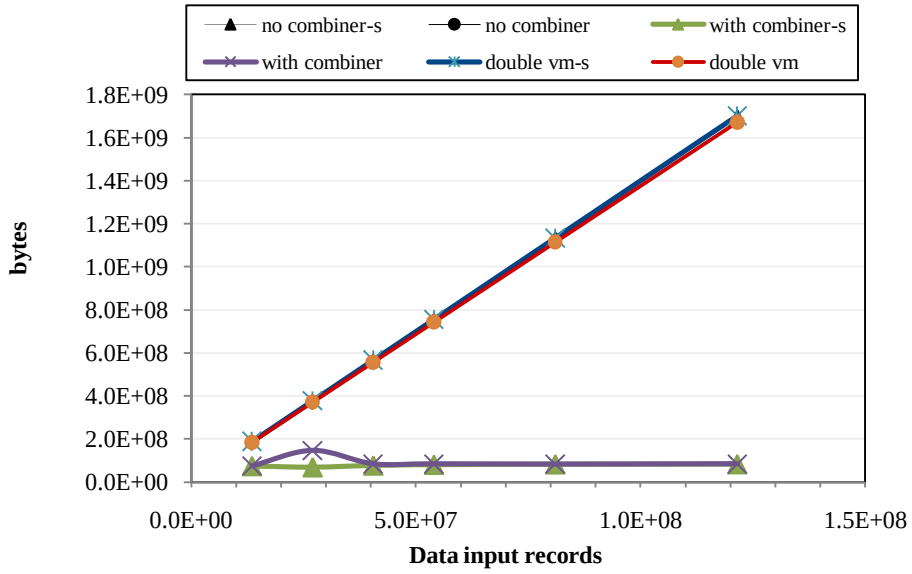


Figure 3-37: Data shuffled bytes vs. input records, using different job configurations

Without using the combiner function, the data shuffled in this algorithm is linear to the number of input records, because all records are transferred over the network to the reducers. When using the combiner, the data shuffled is decreased to a great extent and the value of records shuffled is close to the value of the number of unique keys in the map output keyset. In this example, no matter how big the number of input records is, the shuffled records are always close to the maximum number of unique keys in the map output keyset (100,000).

Figure 3-24 shows the equations that control this behaviour when using the combiner function in the job.

3.4.3 Sort Benchmark Validation

Malley and Murthy [1][114] used Apache Hadoop in a Yahoo cluster of a few thousands machines to compete in Jim Gray's Sort benchmark [115] . Jim's Gray's sort benchmark consists of a set of several related benchmarks, with different rules. All of the sort benchmarks measure the time to sort different numbers of 100 byte records. The TeraSort benchmark samples the input data of one Tera byte size and uses map/reduce to sort it. O.O. Malley and A.C. Murthy published their results and provided a comprehensive description of the cluster configuration and job configuration. This information was used to set up MRSim to simulate their terasort experiment. They used approximately 3800 nodes with these hardware features:

- Quad core Xeons @ 2.5GHz per node
- SATA disks per node
- 8G RAM per node (and 16GB for the petabyte sort)
- 1 gigabit Ethernet on each node. 8 gigabit Ethernet uplinks from each rack to the core.
- 40 nodes per rack
- Red Hat Enterprise Linux Server Release 5.1 (kernel 2.6.18)
- Sun Java JDK (1.6.0 05-b13 and 1.6.0 13-b03) (32 and 64 bit)

Malley & Murthy modified the Hadoop code to optimize it for best performance for this specific terasort job.

3.4.3.1 Simulator Configuration

Trying to simulate yahoo terasort experiments is a challenging task for any distributed system simulator. It takes days to simulate such experiments which run on at least 1406 machines grouped into racks of 40 machines per rack and linked by routers in a tree topology. When performing the experiments, it is noticeable that MRSim – in its full feature version – could not simulate the experiments. Usually simulation engines, such as SimJava, allocate one thread for each simulated process. In the terasort experiment, the number of processes needed to be simulated is several times bigger than the default number of processes allowed by the

operating system for each application. OS configuration was modified to launch the simulator with more memory heap size. Also, MRSim uses GridSim to simulate network traffic. Mehta [116] performed an evaluation of GridSim simulation scalability using NetBeans' profiler and showed that GridSim scaled up to 1380 resource and with memory usage of 413MB and the number of processes running was 18000. MRSim showed even less scalability because it not only uses the GridSim network simulator but it also uses its own simulation entities to simulate other components in the system.

As Malley & Murthy modified the Hadoop code to get optimum performance for this certain test, MRSim was modified to get the best performance value. First logging was turn off as the experiments is interested only in the total job time. Second, by checking the code of Yahoo TeraSort application and by studying the behaviour of it in small cluster, it is noticeable that map and reduce tasks have an input closely equal to the amount of data, the data keys are equally distributed among mappers, and reducers get equal amounts of Mappers outputs. Thus, there is no need to save different objects to hold the values for each data split generated by the map tasks. When saving the data one instance is enough. Another modification is simulating a fewer number of racks while *keeping simulating network traffic with other racks*. Also, the machines used by Malley & Murthy to run TeraSort are of homogeneous hardware features. This allows us to apply the second optimization by simulating part of the whole cluster while keeping simulating the network traffic with the other part of the cluster. This generated the same result as if the whole cluster was simulated but with much faster performance. The third modification to MRSim is the ability to simulate Hadoop in speculative mode as it is enabled in the previous terasort experiments. In speculative mode, several copies of the same map or reduce task are running in parallel on other idle slots in the cluster. Speculative mode provides more reliable job execution but consumes more of the cluster hardware. The source code of optimized MRSim version for TeraSort experiments is available on the project website [41].

Here there are job and cluster configurations for each experiment. These values are derived from job history logs for real experiments and provided by Malley & Murthy [114].

Table 3-3: Cluster configuration for TeraSort benchmark [114]

Job Configuration	Input Data in Bytes			
	500 Giga	**1 Tera**	**100 Tera**	**1 Peta**
"ioSortFactor"	800	100	100	100
"ioSortSpillPercent"	0.95	0.95	0.85	0.9
"mapReduceParallelCopies"	20	20	20	20
"ioSortMb"	800	300	400	1100
"useCompression"	TRUE	TRUE	FALSE	FALSE
"mapredJobShuffleInputBufferPercent"	0.95	0.95	0.8	0.8
"reduceTasksMax"	2	2	2	2
"ioSortRecordPercent"	0.2	0.4	0.4	0.2
"mapredJobShuffleMergePercent"	0.95	0.95	0.8	0.8
"mapredInmemMergeThreshold"	10000	10000	10000	10000
"mapredChildJavaOpts"	Xmx1024m	Xmx200m	Xmx200m	Xmx200m
"mapTasksMax"	6	4	4	2
Nodes	1406	1460	3452	3658
Maps	8000	8000	190000	80000
Reduces	2600	2700	10000	20000
Replication	1	1	2	2
Benchmark by Malley&Murthy (seconds)	59	62	10,380.0	58,500.0
MRSim(seconds)	83.74	111.3	4,509.24	55,506.9

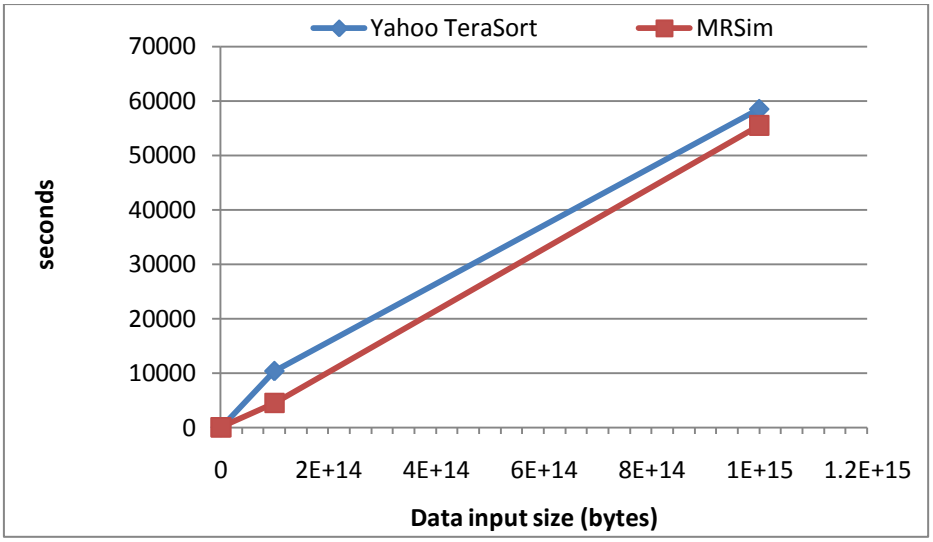


Figure 3-38: TeraSort experiments, time to complete the sort job vs. data input size

Figure 3-38 Shows results of simulating TeraSort benchmark on optimized MRSim simulator. There is difference in times especially in 1 Tera dataset. There is no clear answer for this difference. However, it was noticed that the number of machines used and provided by Malley & Murthy in their paper is different than the numbers of machines used in each experiment as logged into the job history logs provided by the experiment. This issue needs more clarification in order to get more accurate results by the simulator. However, this validation sticks with the number provided in the published paper and not by the technical files provided on the experiment website.

3.5 Summary

This chapter introduced MRSim, a discrete-event MapReduce simulator used to study the behaviour of clusters that run MapReduce middleware implementation. It discussed the design approach and defined layered model. It showed the flow control between the core system entities and how core entities are used by other components in the system. Two schedulers were designed to run the CPU and Hard Drive model. Network topology and algorithm description is fed to the system using JSON format. Several experiments carried out to evaluate both core entities and for overall performance. MRSim is used in next chapter to evaluate the scalability of MapReduce algorithms developed for mining frequent itemset in large datasets.

Chapter 4

MRAPriori: MapReduce Apriori-like Algorithm for Associative Rules Mining

4.1 Introduction

This chapter introduces MRApriori MapReduce Apriori-like Algorithm using MapReduce framework. It can be considered as hybrid approach between Eclat [19] and apriori with HT pruning [20]. It first discusses distributing apriori in MapReduce. The approach is similar to Dynamic counting algorithm [21] when dataset has horizontal format. Then it discusses using vertical format [19] combined with applying scalable parallel intersections for group of tid sets at once using MapReduce framework. This new method used to build MRApriori new associated rule miner using both vertical and horizontal dataset. Two implementations of MRApriori are discussed. At last, experiment results and MRSim simulation shows and analyzed in addition to study of scalability of the new algorithm using MRSim [35] simulator introduced in chapter 6.

4.2 Paralleling Apriori -like Algorithms that Uses Horizontal Data Representation

Counting in Apriori algorithm is used to find all frequent items that occurred in the dataset with number of times of value over the minimum support level. Distributed and parallel counting in apriori is easy to implement in Map-Reduce framework as data are divided into

chunks and are distributed to several nodes with parallel counting processes counting on each node. The results of counting have much smaller size and are communicated into another central node to calculate the overall sum. Counting is an aggregate operation. Thus, using combiners that calculate intermediate sums locally before sending them over the network greatly reduces the data sizes transferred over the network. Using hash tables allows checking all frequent items of certain size which are included in one line. Frequent items are saved in hash table and are distributed to the all working processes before scanning the data. Two main constraints of the previous implementation are it demand repeating scanning whole input dataset until finding all frequent items and hash tables that hold references to so far discovered frequent items may grow to sizes does not fit the available memory. Figure 4-1 shows how to implement distributing counting in apriori algorithm using Map Reduce framework.

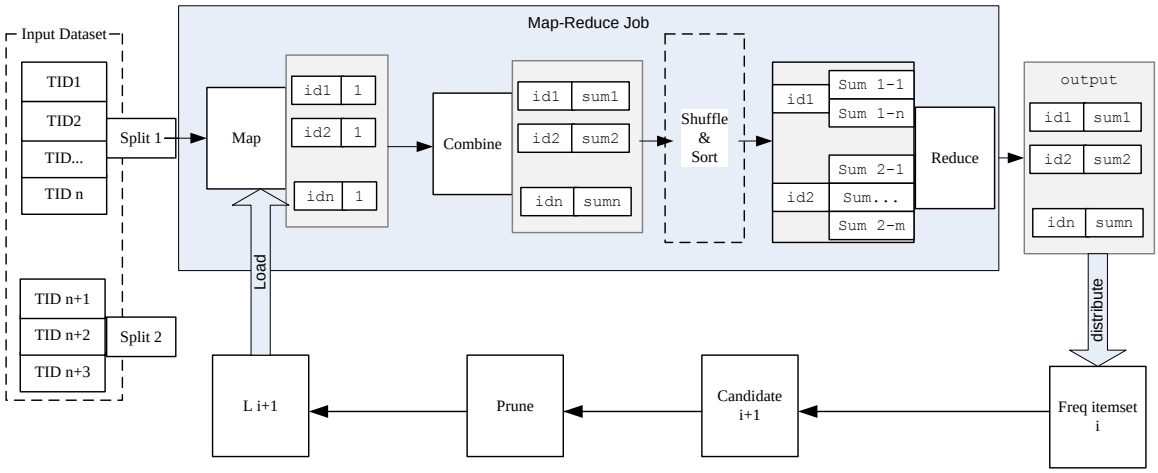


Figure 4-1: Distributed counting in Apriori using MapReduce framework

4.3 Paralleling Algorithms with Vertical Data Representation

Other algorithms showed faster performance than scanning the dataset each time. Other algorithms such as [19][18] avoid iterative scanning of input dataset. It is using the vertical representation of data and finds frequent itemsets by intersection between all set of lines attached to the frequent itemsets of lower size. However, all data is held in virtual memory. Distributing the algorithms that uses vertical data format and set intersection faces some challenges. If the current number sets of frequent item sets is N. Then, on average, each set has to be distributed to (N-1)/2 intersection processes to generate frequent items of higher

degree. The more parallel processes there are the more data has to be transferred to the other processing nodes.

Another restriction in paralleling algorithms that uses vertical datasets is that it requires random access to the data. Random access to data would consume most of the application time. Relational database systems can be used to improve the access time speed. However is add restriction to the scalability as more data to be used in distributed way means that relational database becomes the bottle neck of the system. A good framework for such data storage with random access is using Google Bigtable [5] distributed data structure introduced by Google and has good open-source implementation by Apache Hadoop Project called HBase [3] [117]. The advantage of using HBase also includes seamless effort to parallel distribution of the data using parallel working nodes. However, more effort is needed to coordinate the intersections over the distributed environment.

Here comes MRAproiri which benefits from the simplicity and the abstraction introduced by MapReduce framework to define simple algorithm that is fully distributed and benefits from a new data representation to avoid repeating scanning of original dataset and to avoid using more complicated data structures with more fine scheduling to coordinate processes in distributed environments. MRApriori can be seen as an algorithm that uses vertical data representation and uses fast distributed batch set intersection for finding frequent itemsets.

4.4 MRApriori

Using MapReduce for frequent itemsets counting in apriori (Figure 4-1) showed good scalability for algorithm. However, repeated scanning of dataset is still needed. MRApriori eliminate the need to iterative scanning of the data to find all frequent items. Instead, MRAPriori repeats scanning other intermediate data that usually keep shrinking per iteration. Number of iterations is same as number of iterations in Apriori. But usually, MRApriori scan less data rather than scanning whole data as in Apriori. Thus, the main differences between Apriori and MRApriori are:

- MRApriori do only one scan for the data in original format.
- MRApriori uses new data structure to represent the dataset. It is hybrid of both vertical and line representations.

- MRApriori uses batch set intersection using MapReduce framework where Apriori uses counting.
- MRApriori uses batch rule extracting based on MapReduce framework.

MRApriori consists of three steps, data initialization, frequent items discovery, and rule extraction for frequent items. Those steps in addition to data transformations are explained here.

4.4.1 Data Initialization

MRApriori uses integer values to represents the items in dataset. This makes the algorithm faster and takes less memory sizes. Mapping items into integer values can be delayed and merged to the step of finding frequent item sets of size one. Dataset consists of transactions or records. Each record contains several items. Items in each transaction may be spars. Table 4-1 shows example of input dataset.

Table 4-1: Initial dataset

Items
I1,I2,I3
I2,I4
I2,I5
I1,I2,I4
I1,I5
I2,I5
I1,I5
I1,I2,I5,I3
I1,I2,I5

Table 4-2: add unique ID for each transaction

TID	Items
1	I1,I2,I3
2	I2,I4
3	I2,I5
4	I1,I2,I4
5	I1,I5
6	I2,I5
7	I1,I5
8	I1,I2,I5,I3
9	I1,I2,I5

Table 4-3: Spars dataset

	Column Id				
TID	1	2	3	4	5
1	1	1	1		
2		1		1	
3		1			1
4	1	1		1	
5	1				1
6		1			1
7	1				1
8	1	1	1		1
9	1	1			1

Table 4-4: Map each item to its lowest tid occurrence in the dataset

TID	Item Ids
1	(1)1,(2)1,(3)1
2	(2)1,(4)2
3	(2)1,(5)3
4	(1)1,(2)1,(4)2
5	(1)1,(5)3
6	(2)1,(5)3
7	(1)1,(5)3
8	(1)1,(2)1,(5)3,(3)1
9	(1)1,(2)1,(5)3

The minimum initialization done for data is to add unique integer value for each transaction. Line numbers were used as transaction id (TID). Next, items are mapped to its integer representation (item ids) where each Item is replaced with integer values of two parts; Column Ids, and row Id.

ColumnIds: Map the attributes values to integer ids as shown in Table 4-3.

RowId: the lowest value of TID at which the item first occurred in the dataset.

Mapping items to "(ColumnIds)RowId" format can be done in one scan of data set. In standard alone implementation this can be done by using hash Tables that link each item with the current RowId. Those tables are updated each time new transaction is read. In large datasets, one MapReduce job can do the mapping using simple "map" and "reduce" functions (Figure 4-2).

```
Map():

For each item in the transaction:

        throws the entry <item, TID>

Reduce():

For each group of entries that have same item as key:

        Choose the lowest TID for the item.

        Throw <item, lowest TID number>
```

Figure 4-2: Initialize data using map and reduce methods

The previous reduce function is an aggregate operation of finding the minimum value. Thus reduce function can be used as combiner and this greatly reduces the amount of data to be communicated between the nodes, and will do the data conversion very fast.

Lazy initialization can be applied to the dataset. In this case, mapping the items into the format of "(ColumnIds) RowId" is moved to the process of finding the frequent item sets of size 1. In this way, the previous map or reduce functions are modified to be as shown in Figure 4-3

```
Map( transaction)

For each item (string value) in transaction:

        Throw entry <Item, TID>

Reduce (Group of Entries of same key):

        counter = number of TIDs

        TID_min = lowest value of group of TIDs.

        If counter > min support level

                Throw <Item, TID_min >
```

Figure 4-3: Lazy Initialization using map and reduce functions

MRApriori uses two data structures formats to represent intermediate data used in the algorithm; *line space* format and *item space* format. Example of line space format is the dataset initialized in Table 4-4, where dataset is represented in collection of lines. Each line has the format of:

Line , (columnIds_0)rowId_0,…., (columnIds_n)rowId_n

Line , list of items ids

This is *horizontal* representation of data. Other used representation is the *vertical* or "*item space*" format. Item space can be seen as map where keys are items ids and values are set of occurrence lines to corresponding items. Table 4-5 shows the result of transforming data of Table 4-4 from horizontal format (line space) to vertical format (item space).

ItemId: set of occurrence lines

Table 4-5: Initial data representation in item space

Item	Lines
(1)1	1, 4, 5, 8, 9
(2)1	1, 2, 3, 4, 6, 8, 9
(3)1	1, 8
(4)2	2, 4
(5)3	3, 5, 6, 7, 8, 9

As shown later, this simple data format allows ruleitems of higher degrees to be represented the same way.

4.4.2 Frequent Items Discovery and Rule Pruning

Frequent ruleitem discovery phase in MRApriori works by applying the support condition while repeating the transformation of the input data between the Line space and the Frequent Item space until discovering all frequent ruleitems (Figure 4-6). Data transformation from a Line space to a frequent space is performed using the MapReduce methods "ToFrequent.Mapper" and "ToFrequent.Reducer". The input for the ToFrequent.Mapper method is <line, list of ItemId>, and the output is <ItemId, Line>, which then gets inputted to the "ToFrequent.Reducer" and this method outputs <ItemId, set of lines>.

(line space)<Line Number, List of ItemIds> → ToFrequent.Mapper → <ItemId, Line> → ToFrequent.Reducer → <ItemId, set of lines> (item space)

On the other hand, transforming the data from a FrequentItem space into a Line space is performed using the methods "ToLine.Mapper" and "ToLine.Reducer". The "ToLine.Mapper" gets <ItemId, set of lines> as an input and produce <Line Number: ItemId> as an output, which is in turn gets inputted to the "ToLine.Reducer" and this method collect the ItemIds entries for certain line and outputs<line, list of ItemId> in line space.

(item space) <ItemId, set of lines> => ToLineMapper => <ItemId, Line> => ToLine.Reducer => <line, List of ItemIds> (line space)

If dataset has fixed number of attributes then the maximum number of iterations to find all frequent ruleitems equals the number of attributes (columns) in the training data set. If the data has sparse attributes then the maximum number of iterations is equal to the maximum number of attributes occurred in one line in the dataset. However, the actual number of iterations could be much smaller as the number of items in both line space and item space usually keeps shrinking by iterations. In each iteration, more items are dropped because of

applying support condition and more lines are dropped because they do not have sufficient number of items to generate next ids of frequent items of higher size.

Let's apply the previous frequent items discovery procedures to Table 4-4 . Assume that the MinSupp is set to 3/9 meaning that any keyword that occurs at least three times in the table is considered a frequent ruleitem. To discover the frequent ruleitems of size "1" (Single attribute values with frequencies above the MinSupp threshold), firstly the proposed algorithm transfers the data into Item space .

(Line 1) <(1)1, (2)1 ,(3)1> ToFrequentItem.Mapper <(1)1 , 1 >, <(2)1, 1>, <(3)1, 1>.

(line 2) <(2)1, (4)2> => ToFrequentItem.Mapper =><(2)1, 2>,< (4)2 ,2>.

... etc.

Output results from the Mapper get sorted and introduced to the Reducer grouped by the key value. For instance and for attribute values (keywords) "I1" and "I3", the data offered to the Reducer are as follows:

<(1)1, 1 >,<(1)1, 4 >, <(1)1, 5 >,<(1)1, 8 >,<(1)1,9> ToItem.Reducer < (1)1 ,[1, 4, 5, 8, 9]>

<(3)1,1>, <(3)1,8> ToItem.Reducer <(3)1, [1,8]>

For these particular attribute values, it is obvious that (1)1 is frequent ruleitems with support value 5/9 where (3)1 is not frequent ruleitem because it has support value of 2/9 < 3/9. Thus, item (3)1 is dropped from item space of size 1 as it did not survive the support threshold.

Once the frequent ruleitems of size 1 are determined, then only their occurrences are transformed into the Line space to data format using the MapReduce methods ToLineItem.Mapper and ToLineItem.Reducer. So for ruleitems <"a", r> and <"b", r> which are frequent, their Line space representations are as follow:

<(1)1, [1, 4, 5, 8, 9] => ToLine.Mapper => <1, (1)1 >,<4, (1)1>, <5, (1)1 >,<8, (1)1 >,<9, (1)1 >

<1, (1)1> , < 1, (2)1> => ToLine.Reducer => <1, [(1)1,(2)1]>

The sample outputs are sorted and grouped by the line number and then offered to the ToLine.Reducer which only accumulates the ItemIds and output them to line space. The resulting lines (Table 4-6) would be similar to the previous lines set of Table 4-4 excluding any attribute value which was discarded during the frequent ruleitems generation. If no ItemIds thrown with certain line, or if the number of remaining ItemIds in the line is less than iteration value then this line is dropped from the line space.

Table 4-6: Map each item to its lowest tid occurrence in the dataset

TID	Item Ids
1	(1)1,(2)1
3	(2)1,(5)3
4	(1)1,(2)1
5	(1)1,(5)3
6	(2)1,(5)3
7	(1)1,(5)3
8	(1)1,(2)1,(5)3
9	(1)1,(2)1,(5)3

In next iteration, the proposed algorithm simply finds frequent ruleitems of size N by appending frequent ruleitems of size N-1. Particularly, and for each two disjoint ItemIds a single line within the Line space, the algorithm checks the possibility of joining them to one ItemId then emits the new item to the Reducer with (line) values similar to the previous iteration. For example,

<1, [(1)1,(2)1]> => ToFrequentItem.Mapper => < (1,2)11, 1>

The Reducer groups all items of the same key and for each key generates one ruleitem, such as

< (1,2)11, 1> , < (1,2)11, 4> , < (1,2)11, 8> , < (1,2)11, 9> => ToItem.Reducer => < (1,2)1, [1,4,8,9]>

"(1, 2)1" is frequent item id where "(1, 2)" are columns attributes and "1" is the line number of first occurrence of this item. (1,2)1 is mapped in the original data set to (I1, I2) whose first appearance in the data set is at line 1 (Table 4-1).

The algorithm then repeats the data transformation until all frequent ruleitems are discovered. If the ruleitem survives the MinSupp threshold it will be kept, otherwise it will be discarded

```
1   Input: Training data (D), minSupp and minConf thresholds
2   Output: Set of Frequent ruleitems R
3   // Pre-processing phase
4   If D contains real/integer attributes then
5      Discredite it
6   //Initialization:
7      Map D to integer values to Line Space format LS0
8   //Frequent Items Discovery
9   Map D from Line space LS_0 to Item Space IS_1 to get set S_1 of frequent 1-ruleitems
10  IS_1 = ToItemSpac e(LS_0)
11
12  R ← IS_1
13  max Iteration = max NumberOfAt tributes
14  i ← 2
15  While (i < max Iteration _ and _ size (IS_{i-1}) > 1)
16  {
17        LS_{i-1} = ToLineSpac e(IS_{i-1})
18        IS_i = ToItemSpac e(LS_{i-1})
19        R ← R ∪ IS_i
20           i ← i + 1
21  }
22
```

Figure 4-4: Pseudo code for Initialization and Frequent Item Discovery steps

4.4.3 Generate Strong Association Rules Form Frequent Itemsets

Now that MRApriori has collected the set of all frequent items of all sizes that survived the support threshold. Then it follows the apriori definitions to extract strong associated rules from frequent item set. MRApriori also uses Map Reduce framework to extract significant rules form all frequent item sets.

This is done in one Map Reduce job step using the two map and reduce functions.

```
1  //generate strong rules form frequent itemsets
2  Input set of all frequent items fi
3  Output R: set of strong associated rules
4  Map Function()
5  for each frequent item fi, of support supp_i:
6          for each left-hand a possible subset from fi:
7                  right-hand = get the complement right hand part from fi and left-hand.
8                  map (left-hand -> right-hand: supp_i
9          map fi -> []: supp_i   //(fi is left-hand part, [] empty set as right-hand part)
10 //group entries of same left-hand keys :
11 <left-hand, [right_hand 1: supp_1, …  right_hand_n: supp_n ]>
12 Reduce Function():
13 supp_ = support of  entry that has no right-hand _side []
14 for other entry i which has left-hand side:
15         calculate confidence conf = supp_i/ supp_
16         if conf >= confidence threshold:
17         R = R union (left-hand -> right-hand)
```

Figure 4-5: Pseudo code for extracting associative rules in MRApriori

If all frequent item sets can fit in computer memory and if the processing time is not that big then Hash table data structure can be used to hold the data thrown from the map function. In this case, the key will be the left-part and the value will be set of (right-part: supp) entries for frequent item fi. In the distributed implementation of this step, data are thrown to distributed file system and the Map-Reduce middleware is responsible to sort the entries and to fetch them grouped to the reduce functions.

The overall workflow of MRAPriori algorithm can be shown in Figure 4-6

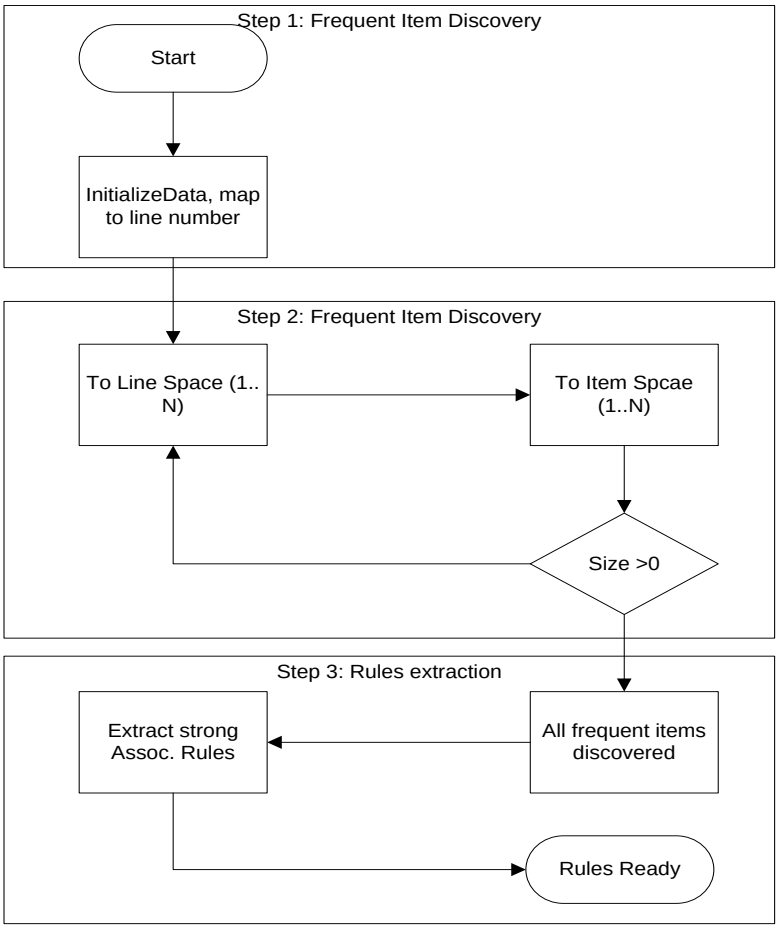


Figure 4-6 Workflow of MRApriori algorithm

4.4.4 Algorithm Features

- All elements, either in line space or in frequent item space, are saved in one virtual collection that has same data structure. This produces simpler abstracted data that is easy to be serialized and to be distributed among the cluster nodes. Also, this helps to

develop more abstracted algorithms such as MRApriori which does not impose restrictions on how to save and coordinate the distribution the data. Data chunks can have arbitrary sizes with no effect on MRApriori accuracy. This allows the underlined middleware (Hadoop in our implementation) to split the data dynamically to achieve load balancing execution with no accuracy consequences. This is an advantage over other algorithms that uses bagging [22] and boosting [23] are affected very much with the sizes of the splits in parallel implementations.

- All candidate frequent items of all degrees are represented in the same way. In the special cases where number of attributes is less than hundreds of attributes, MRApriori can use binary format as in Table 4-3 to hold the values of dataset. Thus one integer number is sufficient to represent the ColumnIds and another integer is sufficient to represent RowId of the item. This is used heavily in MCAR algorithm [15]. In cases of all twenty datasets used in experiments from UCI [103], one integer number of 32 bits memory size was sufficient to represent any frequent item of any degree. For example, the first transaction data in "tic-tac" datasets taken from UCI [103] is "b,o,b,b,o,x,x,o,x,negative" (arff format [42]) there are ten attributes. Thus ColumnIds are represented in integer value with at least 10 bits computer memory size. To represent the ColumnIds of candidate frequent item of size 2 of (attribute 0 = b and attribute 5= x) use $100001_{binary} = 33_{decimal}$ .For other candidate frequent item that represent the highest possible degree reached in this line use $1111111111_{binary} = 4095_{decimal}$. This reduces very much the amount of used memory and increases the algorithm speed as most of operations done on ItemIds in the algorithm are usually reduced to one direct arithmetic operation. For example, merging two ItemIds is done using either union or add operator on two integer numbers ($I_{merg} = I_1 + I_2$) rather than using Java Set data structure to do the union.
- In MRApriori, all data are saved on file system. Processing the data is done in stream I/O reading. This is much faster than accessing the datasets in random access way.

4.5 Scalable Distributed Set Intersection

In algorithms that uses vertical format as in [19], set intersection is used heavily to discover frequent itemsets of higher degree. Two main constraints may arise in this case: first one is data may not fit all in computer memory, the second one is if the number of current frequent

item set is big then disjointing the set and doing the intersection tends to consume a lot of time. Prune process helps decrease the amount of sets to be intersected. However, in processing big datasets, the number of sets to be interested still big even after the pruning process.

MRApriori can be seen in a way similar to [19] algorithms that uses vertical data representation. But MRAproiri is doing the set intersections in parallel in one step for all sets. Here how batch set intersection is done in MRApriori.

Table 4-7: Batch set intersections

Vertical Format (a)	Map to Horizontal Format (b)	Generate intersections per line (c)	Map to Vertical Format (d)
I1 { 1, 2,4,5 } I2 {1,2,5,7 } I3 {2,3,4} I4 {4,5,6,7}	1 {I1,I2} 2{I1, I2,I3} 3{I3} 4{I1, I3, I4} 5{I1, I2, I4} 6{I4} 7{I2, I4}	1 (I1,I2) 2 (I1,I2), (I1,I3), (I2,I3) 3 4 (I1,I3) , (I1, I4) , (I3,I4) 5 (I1,I2), (I1,I4), (I2,I4) 6 7 (I2,I4)	(I1,I2) {1,2,5} (I1,I3) {2,4,} (I1, I4) {4,5} {I2,I3) {2} (I2,I4) {5, 7} (I3,I4) {4}

Column (a) in Table 4-7 shows set of four frequent items with their lines occurrences. There are few steps to generate intersections to all the disjoint of sets. Step one is to map the data line space as in Column (b) in Table 4-7. Step two is to generate disjoints of all items in each line as shown in Column (c). In this column, lines three and six do not produce any disjoint. Step four, is to map back the new generated disjoints to vertical space with the corresponding line (Column (c) in Table 4-7). Doing intersections in this way fit naturally to MapReduce framework. Exact details were explained previously in 4.4.2.

To compare the performance of two methods, another in-memory implementation of this intersection method is built. The following Figure 4-7 shows the times it take to do all the intersection using this new method versus the traditional methods for different number of items.

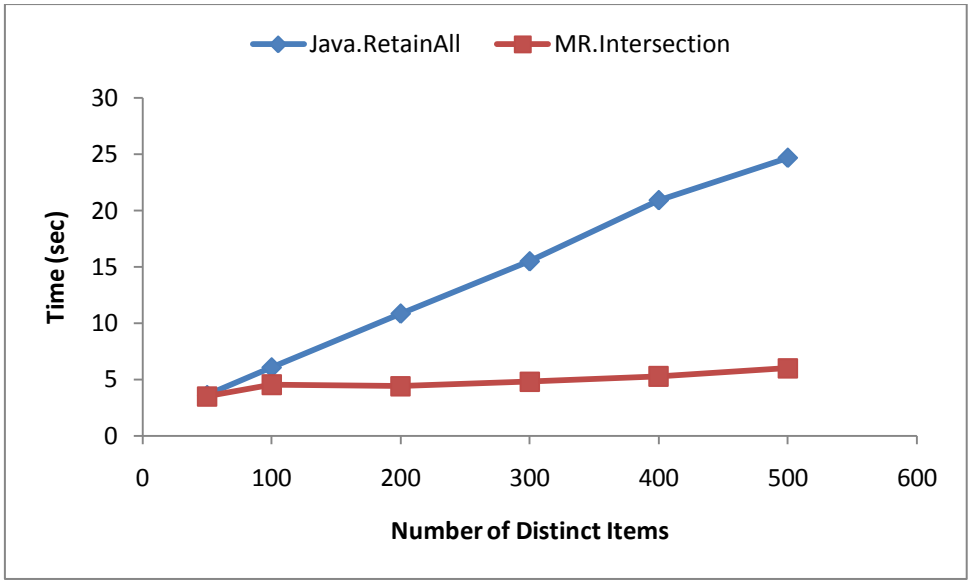


Figure 4-7: Set intersection times using Java.retainAll vs. MR.Intersection methods

Implementation of second method is done in Java and source is available at [44]. This method is compared with method "Set.retainall()" in Java Collections framework [118]. The dataset chosen was a set of 10,000 lines with up to 50 random items occurs in each line. The experiment was repeated for for different distinct items in set from 50 to 500 items. Data size is fixed in all experiments. The more distinct items value means more disjoints to be intersected and at the same time means fewer items in the intersected sets. Figure 4-7 shows that MR-Intersection times- after certain range- are faster than traditional method after certain number of items. The difference between two methods increases as the number of items increases. This means the new set intersection method is much efficient in big datasets where huge disjoint entries can occurs. MRAPriori is designed to target such datasets.

4.6 Implementation

Two implementations in the algorithm, one is included in Weka machine learning software [42]. Other implementation is done in Apache Hadoop [3].

4.6.1 WEKA Software

One implementation of the algorithm is embedded in Weka software [42]. The Weka workbench is a collection of state-of-the-art machine learning algorithms and data pre-processing tools. Weka was developed at the University of Waikato in New Zealand. The system is written in Java and distributed under the terms of the GNU General Public License.

It provides a uniform interface to many different learning algorithms, along with methods for pre- and post-processing and for evaluating the result of learning schemes on any given dataset. Weka provide hundreds of algorithms in filters, clusters, association, classification, regression, neural networks, (add more), analysis and visualization tools, evaluation, etc. A brief explanation about the main GUI interface used in Weka the Weka explorer. Algorithms used in Weka are grouped in six tabs as in Figure 4-8. Short description of the tabs

- Pre-process: Choose the dataset, load it from file system, URL or data based, convert it to different formats choose attributes to be processed and apply dozens of filter algorithms of the data.
- Classify: Train learning schemes that perform classification or regression and evaluate them.
- Cluster: Learn clusters for the dataset.
- Associate: Learn association rules for the data a.
- Select attributes: Select the most relevant aspects in the dataset.
- Visualize: View different two-dimensional plots of the data and interact with them.

4.6.2 MR-Apriori in Weka

Embedding the MRApriori in Weka software will benefit from the extensive tools available in the software to analysis the data and to pre process it before applying the new learning scheme to it. Weka software also has uniform interface to other algorithms. This allows comparing the new algorithm with already existed algorithms.

Weka's class hierarchy was extended with the new associate algorithm MRApriori. So the MRApriori algorithm is available to the user in the Associate tab in the GUI Explorer interface. The base class is MRApriori is included in the "weka.association" package. More methods are overridden to provide generic information about the class such as documentation, its version, its authors and related papers.

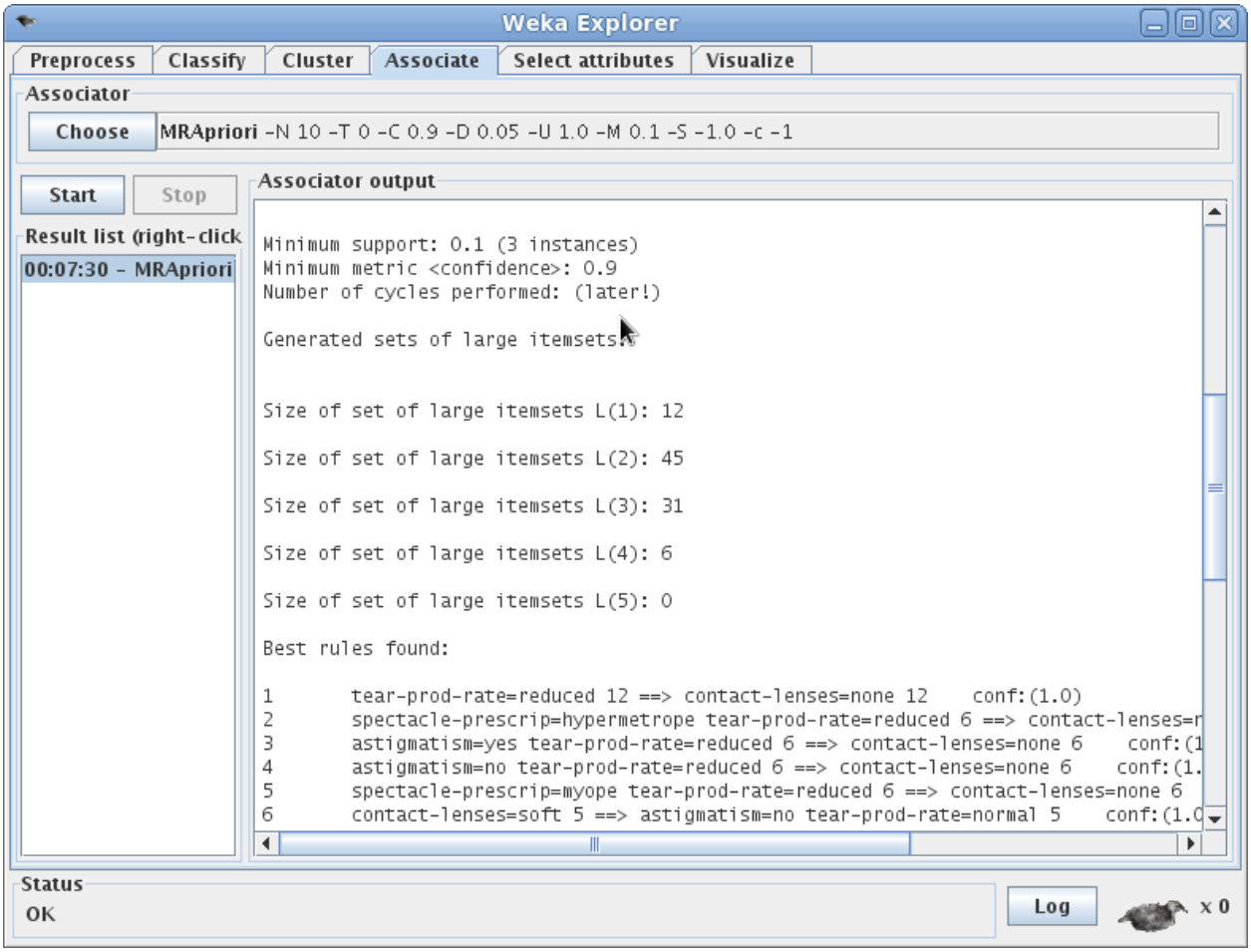


Figure 4-8: Associate panel in WEKA software

Use the Associate panels (Figure 4-8) to invoke methods for finding association rules. MRApriori is added to the other few existing methods. Figure 4-9 shows the output from the MRApriori program for association rules on the nominal version of the "contact" dataset from UCI [103].

Best rules found:

1 tear-prod-rate=reduced 12 ==> contact-lenses=none 12 conf:(1.0)

2 spectacle-prescrip=hypermetrope tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1.0)

3 astigmatism=yes tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1.0)

4 astigmatism=no tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1.0)

5	spectacle-prescrip=myope tear-prod-rate=reduced 6 ==> contact-lenses=none 6 conf:(1.0)
6	contact-lenses=soft 5 ==> astigmatism=no tear-prod-rate=normal 5 conf:(1.0)
7	tear-prod-rate=normal contact-lenses=soft 5 ==> astigmatism=no 5 conf:(1.0)
8	astigmatism=no contact-lenses=soft 5 ==> tear-prod-rate=normal 5 conf:(1.0)
9	contact-lenses=soft 5 ==> astigmatism=no 5 conf:(1.0)
10	contact-lenses=soft 5 ==> tear-prod-rate=normal 5 conf:(1.0)

Figure 4-9: Example of MRApriori results

Despite the simplicity of the data, several rules are found. The number before the arrow is the number of instances for which the antecedent is true; that after the arrow is the number of instances in which the consequent is true also; and the confidence (in parentheses) is the ratio between the two. Ten rules are found by default: User can ask for more by using the object editor (Figure 4-10) to change "numRules" .

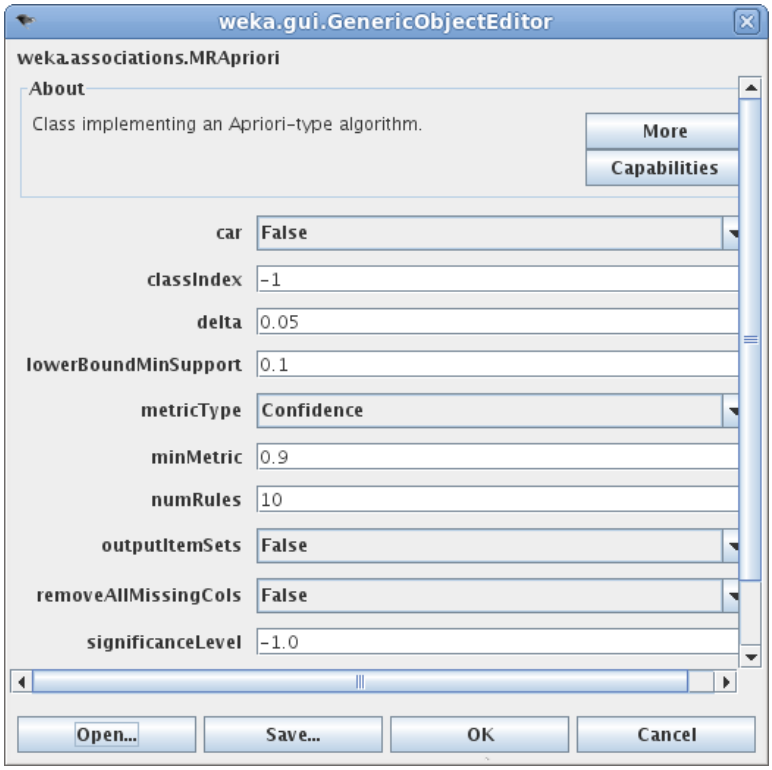


Figure 4-10: Object editor of MRApriori

This is a brief description about Object editor parameters for MRApriori miner in Figure 4-10

- Class Index: index of which attribute is used as label.
- Minimum Confidence: confidence threshold for surviving rules to be considered in the classifier model.
- Is Sparse: true if instances in the training dataset have sparse attributes.
- Minimum Support: support threshold for surviving rules to be considered in training model.
- Number of Rules: build the classifier with support level that generates only this number of rules. Classifier model will start from high support levels then decrease the support in steps till generating the needed number of rules
- Verbose: Used for debugging and algorithm demonstration. If true, then the intermediate data in Item space and Line Space are printed to the output for all iterations. Also will print all surviving rules before pruning.
- Delta: is the difference between every two support steps used in descending way until finding the required number of rules.

4.6.3 Map-Reduce Implementation

Map Reduce implementation is done using Apache Hadoop version 20.1 [3]. The code is documented and available at Google code repository under project name "dataminingGrid" [44].

4.7 Experiments

Twenty datasets were used from UCI [103] were used. Unfortunately, in contrary to the classification algorithms, Weka does not provide a common evaluation module to measure the performance of associate algorithm. Still few measurements can be done on MRApriori.

4.7.1 Number of frequent Item Sets

MRAPriori generates almost identical rules as Apriori algorithm [45]. The algorithm was run on twenty different datasets. The minimum support and minimum confidence levels were adjusted to values of 50% and 80 % respectively.

Results of number of frequent itemsets found in each run is identical as shown in Table 4-8

Table 4-8: Number of associated rules generated for minSupp = 50 % and min Confidence = 80%

Dataset	Size	Classes	Apriori	MRApriori
Austrad	690	2	2081	2081
Balance	625	3	0	0
Breast	699	2	282	282
Cleved	303	2	760	760
Contact	24	3	38	38
Diabetes	768	2	83	83
German	1000	2	1172	1172
Glass	214	7	192	192
Heart	294	2	94	94
Iris	150	3	55	55
Labord	57	2	69	69
Led7	3200	10	145	145
Lymph	148	4	251	251
Mushroom	8124	2	152	152
Pimad	768	2	83	83
Primary-tumor	339	23	1262	1262
Tic-tac	958	2	42	42
Vote	435	2	1083	1083
Wined	178	3	5747	5747
Zoo	101	7	856	856

4.7.2 Times for find frequent Item Sets in Standalone Implementation

This experiment compares the original Apriori [45] with the new MRApriori algorithm. Source code is for Apriori [45] is obtained from Weka software [42]. The second algorithm is implemented as Weka plug-in and source code is available on [44]. Minimum support used is 10% and minimum confidence used is 90%.

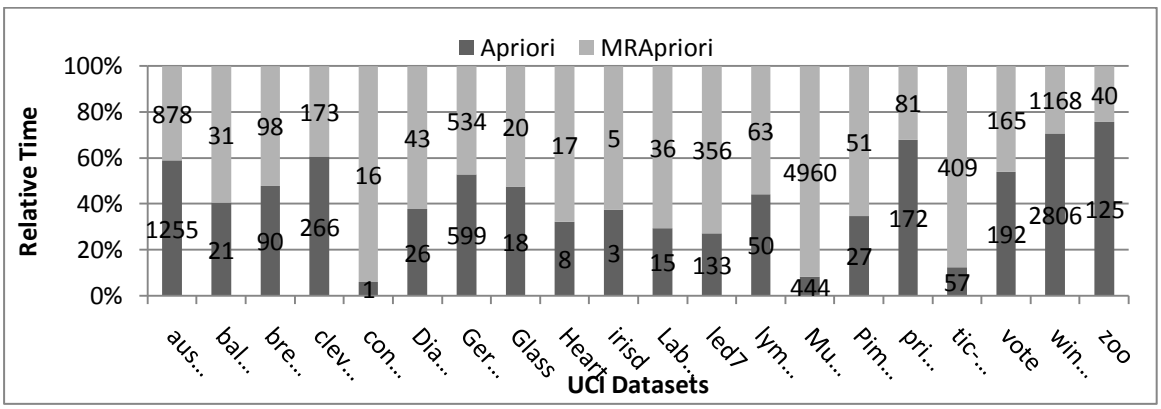


Figure 4-11: Times between Apriori and MRApriori

Figure 4-11 shows the difference of times using 100% stacked columns. WEKA implementation of Apriori is generally takes less time than MRApriori. This might be because, for small datasets the batch set intersection in MRApriori is taking more time comparing with straightforward set intersection. This is shown in Figure 4-7. Also the code of Apriori is taken from WEKA software which is well tested and optimized and thus expected to avoid bottlenecks that affect the application performance.

4.8 MRApriori Performance in Hadoop Cluster

A cluster of three machines was used to test the Hadoop implementation of MRApriori. Also, to further evaluate the effectiveness of MRApriori in large scale MapReduce environments. MRSim, a MapReduce Hadoop simulator introduced in chapter 6 is used to simulate the algorithm in Hadoop cluster environment.

4.8.1 Cluster Configuration

The Hadoop cluster for this set of experiments consist of a total of 12 physical cores across 3 computer nodes as shown in Table 4-9.

Table 4-9 Hdoop cluster configuration

Hardware environment			
	CPU	Number of Cores	RAM
Nodes 1,2 and 3	Intel Quad Core 6600	4	4GB
Software environment			
OS	Fedora13		
Hadoop	Hadoop 0.20.1		
Java	JDK 1.6		

The performance of MRApriori has been evaluated from the aspects of efficiency. Regarding the accuracy, MRApriori generates same results no matter how the input data are splitted.

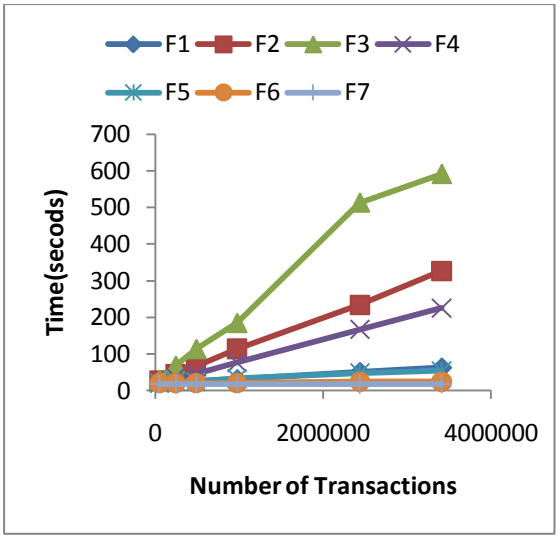


Figure 4-12: To Item space support = 3%

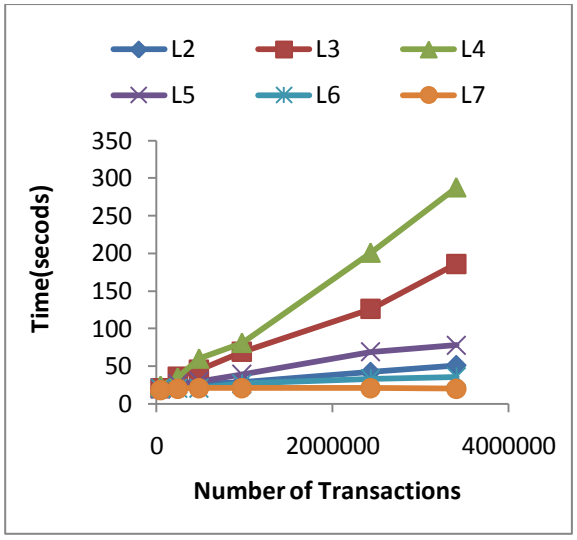


Figure 4-13 To Line space support = 3%

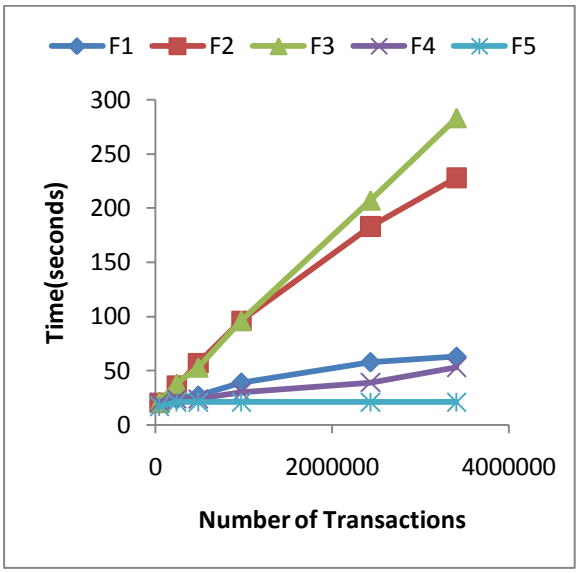


Figure 4-14: To Item space support = 20%

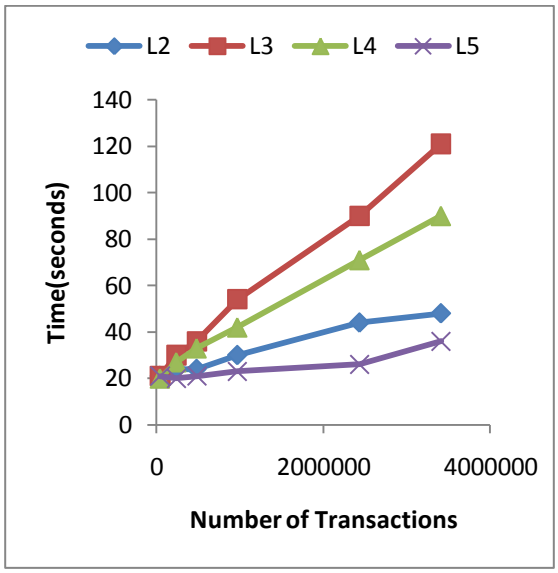


Figure 4-15: Line space support = 20%

F1 to Fn shows the times needed to find frequent items of sizes 1 to n. L2 to Ln shows the times needed to transform data from previous item space to next line space of size 1 to n respectively. Dataset is generated from Mushroom dataset UCI [103] with 11 nominal attributes. In all figures, axis X represents the number of transactions introduced to MRApriori miner, axis Y represents time in seconds to complete the task. Figure 4-12 and Figure 4-14 shows the times it takes to transform the data to item space for support levels equal to 3% and 20% respectfully. Figure 4-13 and Figure 4-15 shows the times it takes to transform the data to line space for support levels equal to 3% and 20% respectfully.

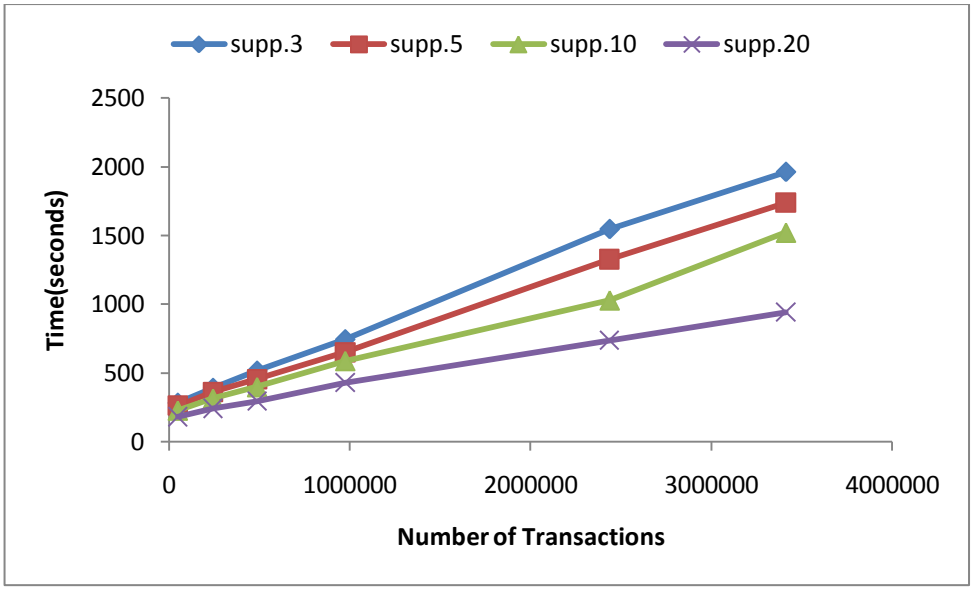


Figure 4-16: Total Time for MRApriori using several support thresholds

Figure 4-16 shows the total time for all iterations for different support levels. It is obvious from all figures that there is initial overhead time is consumed for initializing the job on Hadoop cluster. No matter how small the data size is, at least 15 second is needed to submit the job to the cluster. However, apart from initial overhead, the execution times tend to be linear to the number of records processed. Another remark is that, in both support levels, finding frequent itemsets of sizes two and three consumes most of the algorithm time as shown in Figure 4-12, Figure 4-13,Figure 4-14, and Figure 4-15. This is because line spaces and item spaces of degree two and three has more items than others. Sometimes, when the remaining data in current space is small (spaces of degrees more than four) it is more efficient to carry on finding itemsets of higher sizes in one machine using MRApriori in WEKA implementation to get rid of the overhead needed to initiate new java processes on Hadoop cluster.

4.8.2 Scalability & Simulation Results

To further evaluate the scalability of the MRApriori algorithm, MRSim have also implemented and employed to simulate number of Hadoop environments using a varying number of nodes up to 50.

Table 4-10: Configuration of MRSim for scalability evaluation

Simulation environment	
Number of simulated nodes:	20
Input transactions:	D1=3,412,080- D2=13,648,320 records
CPU processing speed:	100 MIPS
Hard drive reading speed:	80MB/s
Hard drive writing speed:	40MB/s
Network bandwidth:	1Gbps
Total number of Map instances:	6 mappers per node

MRSim [35] is general purpose MapReduce simulator that aims to simulate the behaviour of different algorithms on Hadoop cluster. It is described in details in chapter 6. Each simulated Hadoop node is with 6 mappers, and 2 reducers. Same input dataset were used as an input to the algorithm. Two input datasets were used in the simulation tests; D1 and D2 is generated from Mushroom dataset [103] with number of transactions equals to 3,412,080 and 13,648,320 respectively. Table 4-10 shows the configurations of the simulated Hadoop environments.

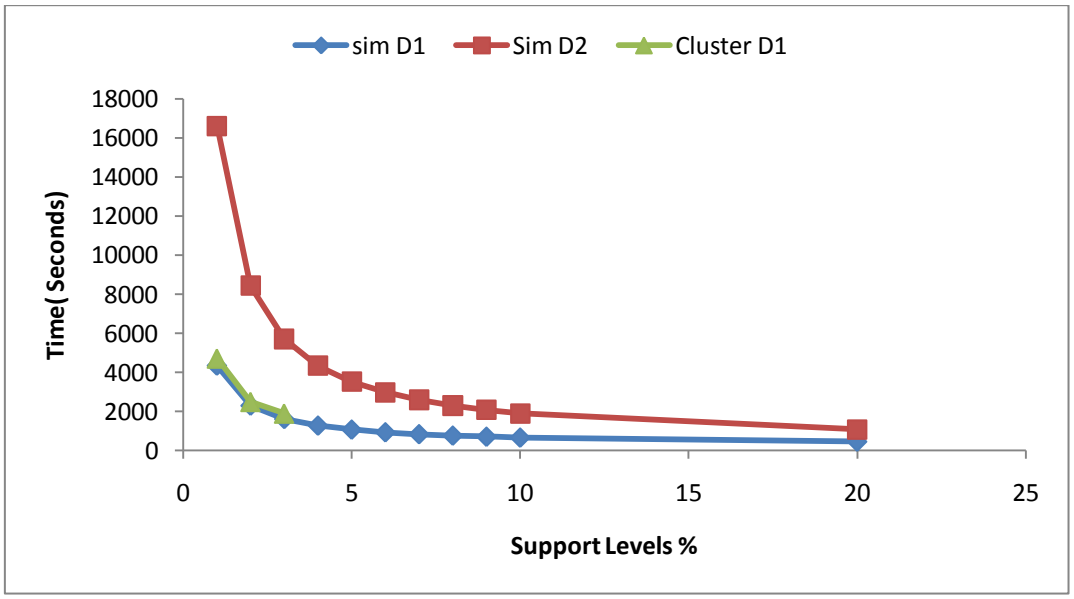


Figure 4-17: Number of nodes vs. execution times

Figure 4-17 shows results of MRSim combined with three points representing runs from real experiments on using D1. The real experiment times is slightly bigger than simulated times.

Also Figure 4-17 shows that the processing time of MRApriori decreases as the number of nodes increases (Cluster D1). It is also worth noting that there is no significant reduction in processing time of MRApriori beyond certain number of nodes for certain data sizes. This is primarily due to the fact MRApriori is using several Hadoop jobs to complete the work. Each job requires around 15 seconds initialization time. Thus, MRApriori in Hadoop scales better when using huge datasets that require times much greater than job initialization times. This is shown clearly in Figure 4-16.

4.9 Summary

This chapter introduced a new association rule miner that uses hybrid approach between algorithms that uses horizontal representation for datasets and other algorithms that use vertical representation. Two implementations of MRApriori were addressed. This chapter concluded on presenting the evaluation and scalability results.

Chapter 5

MRMCAR: MapReduce Multi-Label Classifier based on Associative Classification

5.1 Introduction:

This chapter introduces MRMCAR (**M**ap-**R**educe **M**ulti-label **C**lassifier based on **A**ssociation **R**ules). It talks about data representation, steps of the algorithm, prediction, and incremental learning in the new algorithm.

5.2 The Proposed MapReduce-MCAR (MRMCAR) Algorithm

The proposed AC MapReduce algorithm can be seen as generalized version of MCAR algorithm [15] that is distributable on MapReduce framework. It consists of four main steps, where each step may demand one or more MapReduce jobs:

Step One (Initializing): Representing the input data set in a suitable format for the MapReduce framework, i.e. ItemId= (ColumnId) RowId.

Step Two (Rule Discovery): This step includes discovering frequent ruleitems, rule extraction, and rule pruning. More details are given in 5.2.2.

Step Three (Constructing the classification model): This step involves selecting high confidence and representative rules from the set of candidate rules extracted in Step (2) to represent the classification model, which is laterally employed to predict the class labels of unseen data cases. Also, MRMCAR generates rules of multiple labels. More details are given in 5.2.3.

Step Four (Predicting test cases): In this step, MRMCAR algorithm utilizes a single rule prediction mechanism, and prediction using multiple labels. More details are given in 5.6.

The algorithm deals with categorical and continuous attributes in which continuous attributes are treated using the Multi-interval discretisation technique [96]. Details on the MRMCAR algorithm which involve data initialization, frequent ruleitems discovery, rule generation, classifier builder, and prediction are given in the next subsections.

5.2.1 Initialization

MRMCAR maps each transaction in the dataset to a unique integer value. This value is the number of lines where the transaction occurs in the dataset. This unique value will be noted as RowId. It will be part of the ID of corresponding rules or frequent items that first appeared in dataset at this line. Every frequent item id (ItemId) consists of two parts; column ids, and RowId

ItemId = (column ids) Row Id

Column Ids: are the ids of attributes in the original data set which compose this frequent item.

RowId: The line number (transaction id) of the first occurrence of this item in the original data set.

Once the original data is represented in ItemId format, then all intermediate data generated in the algorithm will keep the same representation. This makes the iterative process of finding frequent itemsets simpler throughout the algorithm. One more benefit of such a data representation is to reduce the amount of data to be communicated between the nodes running the algorithms in the distributed implementation.

Here an example of how to initialize dataset is shown in Table 5-1. The initial data representation will look as shown in Table 4-2.

Table 5-1 Example dataset

TID	Attributes			Class
0	A	B	C	M
1	C	B	C	M
2	C	D	C	P
3	C	D	C	R
4	A	B	A	P
5	A	D	A	R
6	C	D	A	R
7	C	B	D	R
8	A	B	A	R

Table 5-2 Initial data in line space

Line:Label	Attributes		
0:0	(0)0	(1)0	(2)0
1:0	(0)1	(1)0	(2)0
2:2	(0)1	(1)2	(2)0
3:3	(0)1	(1)2	(2)0
4:2	(0)0	(1)0	(2)4
5:3	(0)0	(1)2	(2)4
6:3	(0)1	(1)2	(2)4
7:3	(0)1	(1)0	(2)7
8:3	(0)0	(1)0	(2)4

Table 5-3 Initial data in Item Space

Attribute	Line:Label
(0)0	0:0, 4:2, 8:3
(0)1	1:0, 2:2, 3:3, 6:3, 7:3
(1)0	0:0, 1:0, 4:2, 7:3, 8:3
(1)2	2:2, 3:3, 5:3, 6:6
(2)0	0:0, 1:0, 2:2, 3:3
(2)4	4:2, 5:3, 6:3, 8:3
(2)7	7:3

MRMCAR uses two data structure formats to represent intermediate data used in the algorithm; *line space* format and *item space* format. An example of line space format is the dataset initialized in Table 5-1, where dataset is represented in collection of lines. Each line has the format of:

Line:label, (columnIds_0)rowId_0,…., (columnIds_n)rowId_n

Line:label, list of items ids

This is a *horizontal* representation of data. The other representation used is the *vertical* representation or "*item space*" format. Frequent Item is data structure which maps the labels with corresponding lines for this ItemId. ItemId: is set of occurrence lines with their labels. As shown later, this simple data format allows ruleitems of higher degrees to be represented the same way.

5.2.2 Frequent Ruleitem Discovery

The key to success of the proposed AC algorithm is data representation and data transformation in which the algorithm keeps changing the data format between two spaces (Line space and Frequent Item space).

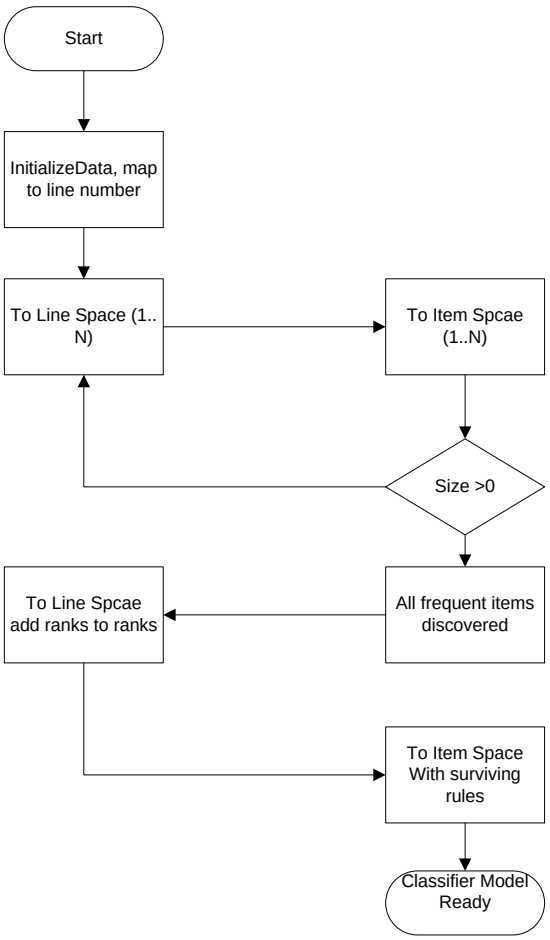


Figure 5-1: Data workflow in MRMCAR

Frequent ruleitem discovery in MRMCAR works by repeating the transformation of the input data between the Line space and the Frequent Itemset space until all frequent ruleitems are discovered. Data transformation from a Line space to a frequent space is performed using the MapReduce methods "ToFrequent.Mapper" and "ToFrequent.Reducer". The input for the "ToFrequent.Mapper" method is <line: label, list of ItemId>, and the output is <ItemId, (Line: label)>, which then gets inputted to the "ToFrequent.Reducer" and this method outputs <ItemId, FrequentItem>.

(line space)<Line Number: Label, List of ItemIds> => ToFrequent.Mapper => <ItemId, (Line: label)> =>

ToFrequent.Reducer => <ItemId, FrequentItem>(Frequent ruleitem space)

On the other hand, transforming the data from a FrequentItem space into a Line space is performed using the methods "ToLine.Mapper" and "ToLine.Reducer". The "ToLine.Mapper" gets <ItemId, FrequentItem> as an input and produces <Line Number:Label, ItemId> as an output, which is in turn gets inputted for the "ToLine.Reducer" and this method collects the ItemIds entries for a certain line and outputs<line: label, list of ItemId> (Line space).

(items space) <ItemId, FrequentItem> => ToLineMapper => <ItemId, (Line: label)> =>

ToLine.Reducer => <line: label, List of ItemIds> (line space)

The maximum number of iterations to find all frequent ruleitems equals the number of attributes (columns) in the training data set excluding the class attribute (For Table 5-1, the maximum number of iterations is three). The actual number of iterations could be smaller if there are no FrequentItems discovered at certain steps.

Let's apply the previous ruleitem discovery procedures to Table 5-1 . Assume that the last value of each row in Table 5-1 denotes the class label, and MinSupp is set to 2/9, meaning that any keyword that occurs at least two times in the table is considered a frequent ruleitem. To discover the frequent ruleitems of size "1" (Single attribute values with frequencies above the MinSupp threshold), firstly the proposed algorithm transfers the data into Line space as shown in Table 5-2. The proposed algorithm then applies the "ToFrequent.Mapper" and "ToFrequent.Reducer" methods to map the input data to entries in the frequent space. In this way and for each item in the Line space the "ToFrequent.Mapper" method is invoked to emit list of <ItemId, (Line,Label)>

(line 0) <0:0, (0)0, (1)0, (2)0> ToFrequentItem.Mapper <(0)0 ,(0:0)>, <(1)0, (0:0)>, <(2)0, (0:0)>.

(line 1) <1:0, (0)1, (1)0 , (2)0> => ToFrequentItem.Mapper =><(0)1,(1:0)>,< (1)0 ,(1:0)>,< (2)0 ,(1:0)>

... etc.

Then, the output results from the Mapper are sorted and introduced to the Reducer grouped by the key value. For instance, for attribute values (keywords) “a” and “c”, the data offered to the Reducer are as follows:

```
<(0)0, 0:0 >,<(0)0, 4:2 >, <(0)0, 5:3 >,<(0)0, 8:3 > ToFrequentItem.reduce < (0)0 ,[ 0:0, 4:2, 5:3, 8:3]>
..... ToFrequentItem.reduce< (0)1 ,[ 1:0, 2:2, 3:3, 6:3, 7:3]>
```

For these particular attribute values, it is obvious that (0)0 and (0)1 are frequent ruleitems with support values 2/9, and 3/9, respectively. It should be noted that in the rule discovery step while determining the frequent ruleitems, MRMCAR considers the attribute value occurrence with its largest class label, and for this reason (0)0 and (0)1 are marked as frequent with class label 3 since they appear it in the training data set with it more than the rest of the class labels (label 3 corresponds to R in original data set). This is the preliminary label choice attached to this ruleitem. However, the final label for this rule item is decided in a later step based on which label among the possible labels attached to the ruleitem actually covers more instances. Thus, MRMCAR, at this step, considers only single label rules and chooses the largest frequency class associated with an attribute value. In case an attribute value is associated with multiple class labels with similar frequency, the choice is random. Now the frequent item set of size 1 (one attribute ruleitems) was collected. For frequent ruleitems that survived the minimum support threshold, the confidence of the item is calculated at once and if it passes the minimum confidence condition, it will be marked as a candidate rule.

List of frequent ruleitems:

```
(0)0    { sup=2 , conf=0.500,          0:[0]    2:[4]    3:[5, 8]}
(0)1    { sup=3 , conf=0.600,          0:[1]    2:[2]    3:[3, 6, 7]}
(1)0    { sup=2 , conf=0.400, 0:[0, 1] 2:[4]    3:[7, 8]}
(1)2    { sup=3 , conf=0.750,  2:[2]   3:[3, 5, 6]}
(2)0    { sup=2 , conf=0.500,  0:[0, 1] 2:[2]    3:[3]}
(2)4    { sup=3 , conf=0.750,  2:[4]   3:[5, 6, 8]}
```

As shown previous list of frequent ruleitems, in each ruleitem, lines of the same label value are grouped together. Once the frequent ruleitems of size 1 are determined, then only their

occurrences are transformed into the Line space to data format using the MapReduce methods "ToLineItem.Mappe"r and "ToLineItem.Reducer". So for ruleitems <"a", r> and <"b", r> which are frequent, their Line space representations are as follows:

(0)0 { sup=2 , conf=0.500, 0:[0] 2:[4] 3:[5, 8]} => ToLineMapper =>

<0:0, (0)0>, <4:2, (0)0>,<5:3, (0)0>,<8:3, (0)0>

(0)1 { sup=3 , conf=0.600, 0:[1] 2:[2] 3:[3, 6, 7]} =>ToLineMapper =>

<1:0, (0)1>, <2:2, (0)1>,<3:3, (0)1>,<6:3, (0)1>,<7:3, (0)1>

The sample outputs are sorted and grouped by the line number and then offered to the "ToLine.Reducer" which will only accumulate the ItemIds and output them to line space. So the lines would be similar to the previous lines set of Table 5-2 excluding any attribute value which was discarded during the generation of frequent ruleitems. If no ItemIds were thrown with a certain line, then this line is dropped from the line space.

In the next iteration, the proposed algorithm simply finds frequent ruleitems of size N by appending frequent ruleitems of size N-1. Particularly, and for each two disjoint ItemIds in a single line within the Line space, the algorithm checks the possibility of joining them to one ItemId. More details about this is shown in 5.2.2.1. The new ItemId of higher degree is then emitted to the Reducer with (line:class) values similar to the previous iteration. For example,

0: 0, (0)0, (1)0, (2)0 => ToFrequentItem.Mapper =>

<(0,1)00, (0:0)>, <(0,2)00, (0:0)>, <(1,2)00, (0:0)>

The Reducer groups all items of the same key and for each key generates one ruleitem, such as

(0, 1)2 {sup=2 , conf=0.667, 2:[2] 3:[3, 6]}

(0,1)2 is ruleitem id: "(0,1)" are columns attributes and "2" is the line number of the first occurrence of this item. ItemId (0,1)2 is mapped in the original data set to attribute 1= C, attribute 2= D whose first appearance in the data set is at line 3.

If the ruleitem survives the MinSupp threshold it will be kept, otherwise it will be discarded. The algorithm then repeats the data transformation until all frequent ruleitems are discovered. The previous steps can be summarised in the following pseudo code in Figure 5-2.

```
1   Input: Training data (D), minsupp and minconf thresholds
2   Output: Set of Frequent ruleitems R
3
4   Preprocessing phase
5   If D contains real/integer attributes then
6      Discredite it
7
8
9   Initialization:
10     Map D to integer values to Line Space format LS0
11  Frequent Items Discovery
12
13  Map D from Line space LS_0 to Item Space IS_1 to get set S_1 of frequent 1-ruleitems
14
15  IS_1 = ToItemSpac e(LS_0)
16  S_1 = CandidateR ule(IS_1)
17
18  R ← S_1
19  max Iteration = numberOfAt tributes
20  i ← 2
21
22  While (i < max Iteration _ and _ size (IS_{i-1}) > 1)
23  {
24
25        LS_{i-1} = ToLineSpac e(IS_{i-1})
26
27        IS_i = ToItemSpac e(LS_{i-1})
28
29        S_i = getCandida teRule (IS_i)
30
31        R ← R ∪ S_i
      i ← i + 1
  }
```

Figure 5-2: MRMCAR pseudo code for rules discovery step

As explained before, MRMCAR has some salient features.

- MRMCAR scans the original data set only once. However, it scans the ever changing intermediate data for several times during the rule discovery. Line spaces and frequent item spaces generated in an iteration tend to shrinks while iteration advances to higher degrees as will be shown later in incremental learning section 5.7.3.
- MRMCAR works in a file stream I/O way. It avoid the costly slower I/O random access file operations. “ToLineSpace” and “ToItemSpace” are two procedures to transform data formats between line and item spaces. Those procedures – in addition to the data formats – fit naturally in the MapReduce framework. Also, the algorithm

can be described in a simple abstracted notation usually used in functional programming languages as Lisp.

5.2.2.1 Generating and Pruning IDs for higher frequent items

In apriori-like algorithms, frequent itemset pruning is done by generating candidate ruleitems of higher degrees, then, for each candidate frequent item, check if all subsets generated in this ruleitem are included in the previous set of ruleitems. In MRMCAR, generating candidate ruleitems is done in the line space from existing ItemIds. For example ,the following line in a two degree line space:

```
Line:label  (1,2)row0 , (2,3)row1,  (1,3)row3 => (1,2,3)row0
    where row0 has the lowest integer value among the values of (row0, row1, row)
```

Means that it is sure that the candidate ruleitem occurred at least once in the data (at least at the current line in line space). Thus MRMCAR greatly reduces the candidate ruleitems by generating them from their actual occurrences in line space, and not from all possible disjoints of items in frequent item space as apriori-like algorithms do. However, there is still a need for in-line check before generating candidate ItemIds. This functionality is similar to hash based technique introduced in [20].

```
1    Input L= list of ItemIds in the current line,
2    Output out= list of ItemIds of higher degree
3    Sz= number of attributes for each item in L
4    DJ= disjoint of L
5    M= hash table that maps each resulting Item to its counter
6    For each I1, I2 form DJ: begin
7
     I_mrg = I_1 ∪ I_2
8        If( Imrg.size == sz+1)begin
9            Increase M(Imrg) by 1
10           End
11   End
12   For each entry in M map: begin
13   If  M(Imrg) == factorial(sz+1)/2: begin
14           Output.add(Imrg)
15   end
```

Figure 5-3: Generating next Candidate ruleitems IDs

In MRMCAR, hashing used in the ToItemSpace procedure and explained in pseudo-code in Figure 5-3. Generating frequent ids of the next iteration requires - for each line in line space - double scanning the line to generate disjoints of ids in line, and then counting the number of occurrences for items resulting from merging the disjoint items. If the expected number of attributes of generated Id is K, then its attached counter in the hash table should have the value equal to $\frac{K!}{2}$ as mentioned in line 13 of pseudo code.

5.2.3 Rule Pruning and Classifier Building

The proposed algorithm extracts a set of rules at each iteration starting from rules of length one (the antecedent contains a single attribute value) until iteration N where N corresponds to the number of attributes in the training data set excluding the class attribute. The proposed algorithm invokes a pruning procedure to significantly cut down the number of redundant and misleading rules and to select the fittest rules to form the classifier.

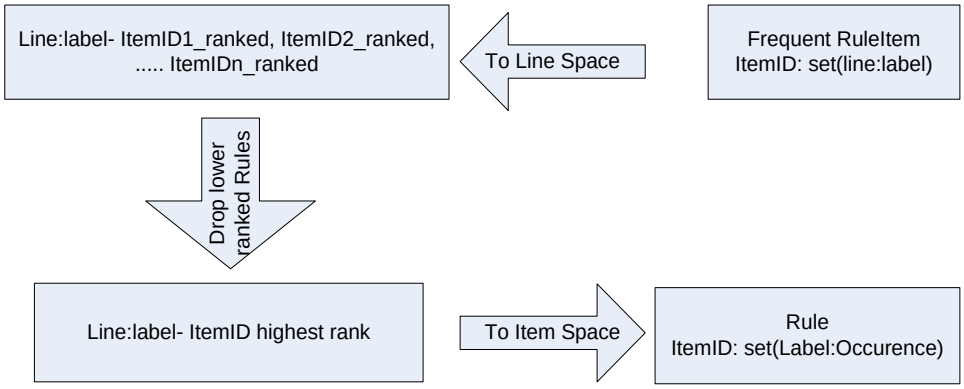


Figure 5-4: Steps of rule pruning

MRMCAR does the rule pruning in three steps as shown in Figure 5-4. First: it transforms the rules to line space format. At the same time, a rank value is attached to each rule. The rank value depends on several criteria; rule confidence, rule support, and number of attributes in the rule's antecedent. The algorithm behaviour is studied for each combination of ranking criteria and results are shown in next chapter at section 6.5.5 . Second: for each line remaining in the line space, keep only the rule which has the highest rank value which correctly predicts the instance label. Another possibility is to choose the rule with highest rank value no matter what the predicted label may be. Other rules that could not cover any instance in the line space are dropped and will not appear in the classifier. The third step in

the rule pruning phase is to re-transform the resulting line space to frequent item space. The same methods as used in rule discovery are used in this step. Using rank value in each rule ensures the resulting frequent items are sorted as well.

As mentioned before, the sorting of rules is done after the pruning process. In MCAR [15] and CBA [13], sorting all possible rules is done at once before the process of rule pruning. Thus, in MRMCAR sorting is done more efficiently, because it is sorting a smaller set of rules, which are the remaining rules that cover at least one instance. The classifier is the set of resulting significant sorted rules. One more rule of lowest rank value may be added to the classifier. It is the default rule which covers all lines that were not covered by any significant surviving rule. The default rule has the label which has the majority of occurrences in the remaining uncovered lines. The following pseudo-code in Figure 5-5 summarises the steps of rule ranking and classifier building.

```
1   Input: set of generated rules (R)
2   Output: The classifier (Cl)
3   MR' = rank(R);
4   Transform R from frequent item space to line space; add rank value for each rule.
5   For each line in line space, choose the first highest rule as follows:
6   begin
7       if exact label matching is required :
8           choose mr :the first highest ranked rule that has label equal to the label of the line
9       else
10          choose mr :the first highest ranked rule no matter what its label is
11      mr is marked as significant rule that covered at least one instance in the dataset
12      discard all remaining rule ids in line
13  end
14  Transform the rules from line space to item space
15  For each rule in item space :
16      Collect the occurrences of different labels to form multi-label rule
17      Choose label of majority of occurrences as main label for single label prediction
18  Sort the resulting rules based on rank to form the classifier model
19  if in covered lines > 0 and default label required :
20      add default rule with class equals to the majority class in remaining lines
```

Figure 5-5: Rule pruning and building classifier

The CBA [13] and MCAR [15] pruning procedure requires the similarity of the class labels of both the selected rule and the training case. This is similar to MRMCAR with the

configuration of "exact label matching" as in line 8 in the previous pseudo-code. However, in "any label" configuration (line 10 in pseudo-code) MRMCAR only considers the matching between the rule body and the training case. This indeed reduces overfitting since most current AC algorithms including CBA and MCAR mark a rule as a classifier rule if it matches the training case and has the same class as the training case. This may result in a more accurate prediction on the training data set but not necessarily on new unseen test cases. The class matching of the candidate rule and the training case does not necessarily give an additional sign of rule goodness besides the matching condition between this rule body and the training case attribute values. In other words, the performance of the classification model is not yet generalised since it has not been tested on an independent test cases to measure its predictive power. We argue that a similarity test between the candidate rule class and the training case class may not heavily affect the predictive power of the resulting classification models during the prediction step. Later in chapter 5, the main results obtained with reference to classification accuracy on different UCI data sets [103] were shown for both rule pruning procedures (The one that looks at the class and the one that marks the applicable rule without checking the class).

5.3 MRMCAR for Multi-Label Classification

In the final step in rule pruning, the frequent rule items are represented in line space in this format: Line number: Class Lable, ruleId

RuleId is the surviving ruleId based on ranking criteria; it covers the line instance and at the same time it eliminates all other RuleIds that mapped to that line. As a result, some the rules may not be able to cover any of the lines they are mapped to, because there are always other rules with a higher rank value in the same line. The final step in rule pruning is to transfer the rule to the Frequent Item space. The result is the subset of the surviving rules that for sure cover at least one line in the original dataset. However, ruleId may be mapped in line space to several lines of different class labels. When transforming to item space all these lines are grouped according to each class label. The resulting rule then has the information of all labels with their occurrences. One final rule in the item space has a similar format to the following rule:

RuleId: Label 1: Occurrences 1, Label 2 Occurrences 2 … Label n: Occurrences n

Thus for each rule, the distribution of occurrences for each label is known. It is straightforward to generate probability values to each label for this rule. The result would be a rule predicting multiple labels like this: if the attributes of an instance match these conditions, then the class label would be: Label one for 80% percent probability, Label 2 for 15% probability or label 3 with 5% probability. This information could be helpful for certain classification applications where it is important to know the other options that the label may take for a certain instance.

For single label prediction, the MRMCAR chooses the label with the highest number of occurrences. This label might not have the higher confidence when discovering the possible rule items in the rule discovery step. These occurrences are the actual occurrences after applying the rank criteria on the data set. Thus MRMCAR can provide high accuracy results if tested on the same training dataset. There is an over-fitting factor here. However, experiments in the next chapter, section 5.5 showed that by using other evaluating methods that eliminate the over-fitting factor such as a 10-fold cross validation test, MRMCAR still maintain good accuracy for single label prediction. Usually, the labels of higher confidence calculated in the frequent item discovery step, keep the same higher confidence and higher occurrences in the rule pruning step.

5.4 Confidence Batch Classifier

By "Confidence Batch Classifier" means that MRMCAR is able to build at once groups of classifiers of the same support with different confidences levels. In the rule discovery step, the candidate frequent items will be calculated based on the support threshold; if they survived the support threshold then they will be kept for the next iteration even though they might not survive the confidence threshold. Candidate rules are picked before the rule pruning step, from the surviving frequent items if they passed the confidence threshold. Having spent a lot of calculations to get to this step, it is very practical to pick a different rule set for different confidence values. This adds no calculation cost to the algorithm at the rule discovery step which takes most of calculations in the MRMCAR algorithm and in other algorithms such as CBA [13][119], MCAR, and [26]. This allows building - at one go - several classifiers for several confidence values for one support value. In the experiments,

this feature is used in MRMCAR for building classifiers for 20 confidence values ranging from 0% to 100% thresholds for each support threshold. The increasing delta used is 5% each time.

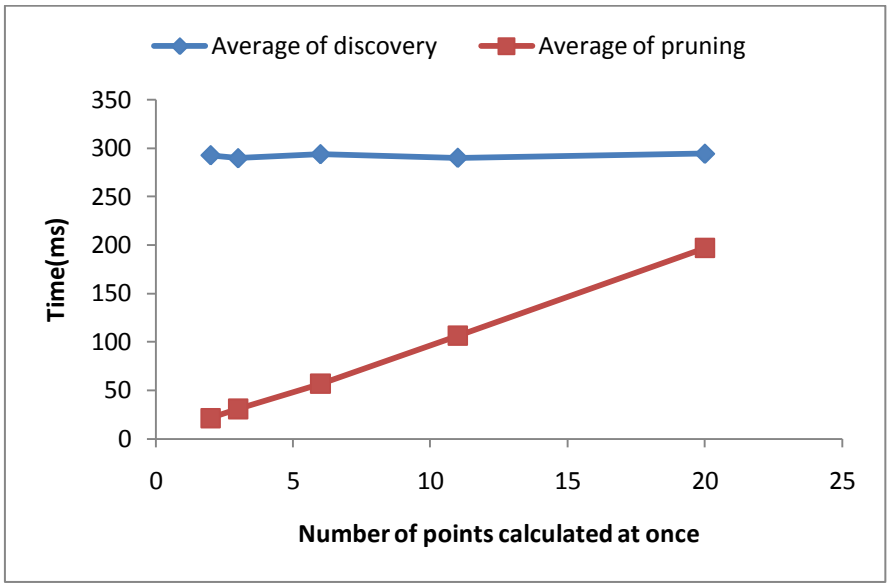


Figure 5-6: Average processing time vs. number of confidences calculated at once

As an example, Figure 5-6 shows times needed for running MRMCAR on a discredited "Wine" dataset taken from UCI Figure 5-6 . Each test is run for 100 times and average values are taken. At the same support level, MRMCAR extracted two sets of rules for two confidence levels in 292.8 milliseconds. Also, it extracted 20 sets of rules based on 20 confidences levels with only 1.5 milliseconds additional time to get 294.31 ms average time. This is a great saving in time if MRMCAR is to be used with different levels of confidences per support level. The other line in Figure 5-6 shows that the average time taken to prune the extracted rules and to build the classifier is linear to the number of sets calculated at one go. Figure 5-7 shows the total time used for doing all steps in two cases; with and without batch calculation.

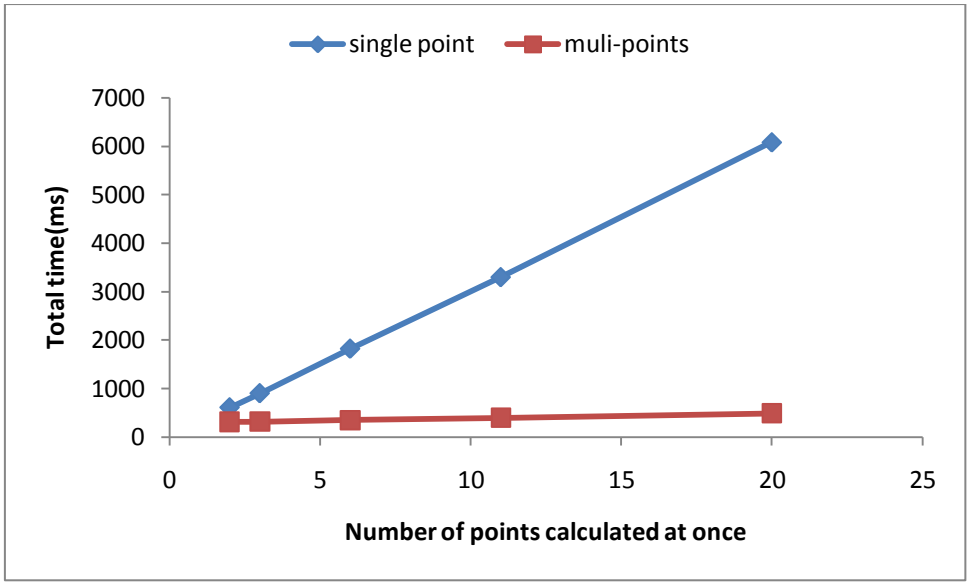


Figure 5-7: Total processing times in ms vs. number of confidences calculated at one go

A later result of evaluating MRMCAR (section 6.5.4) shows that the best accuracy that MRMCAR may achieve is not fixed in one range of confidence levels. Thus it is useful to build the classifier for several levels of confidences at once as it only adds a minor overhead to the calculations.

5.5 Rule Ranking and Sorting Criteria

MRMCAR is flexible when choosing between different rule ranking criteria in the rule pruning step. The rank value is used to predict the label of a training instance among all rules that target that instance. When transforming frequent ruleitems that survived the support and confidence thresholds to the line space, the resulting data has the following format:

Line: Label - RuleId1, RuleId2…RuleIdn

One or more ruleitems can cover the line. The user has the ability to define the way that MRMCAR chooses which rule to keep. MRMCAR will choose the ruleId based on rule ranks and based on label matching.

Rules differ from each other in their values of support, confidence, number of attributes in the left side of the rule. MRMCAR uses these differences to rank different rules. If the user chooses {CONF_SUPPORT_ATT} then the algorithm compares rules based on confidence level first. The rule of higher confidence will have the higher rank. If the confidences are

equal, then MRMCAR will rank them on their support values. If confidence and support values are equal, then the ranking is based on the number of attributes in left side of the rule (the condition). Several possibilities of ranking are available. MRMCAR chose only 5 possibilities and did the experiments on them

- CONF, SUPPORT, ATT: ranking based on confidence then support then number of attributes.
- CONF, ATT, SUPPORT: ranking based on confidence, then number of attributes and finally based on the support level of rules.
- SUPPORT, ATT, CONF: ranking based on support level, then number of attributes, and finally based on confidence levels.
- ATT, CONF, SUPPORT
- SUPP, R_ATT, CONF: R_ATT means the less the number of attributes, the higher the rank is.

MRMCAR chooses the higher ranked RuleId based on ranking criteria as before, then it will look to the class label attached to this rule and will apply either of two label matching criteria:

Any Label Match: MRMCAR will choose the higher ranked rule as the covering rule for this training instance.

Exact Label Match: MRMCAR will choose the higher ranked rule. Then it will keep the rule if the class of rule matches the label of the line. If they do not match, then MRMCAR will choose the next higher ranked rule that has a class which matches the label of the line. If no RuleId matches the label, then this line is not covered by any classifier rules and will be dropped from line space.

If the rule ranking method is “CONF, SUPP, ATT” and the label matching method is “Exact Match” then MRMCAR will be pretty much similar to the MCAR algorithm [15]. The effect of these criteria on MRMCAR accuracy and classifier features is shown in the next chapter, Section 5.5.

5.6 Prediction and Test

In classifying a test case, two types of prediction are used in MRMCAR; single rule prediction and group of rules prediction.

5.6.1 Single Rule Prediction

The classifier finds the first highest ranked rule that matches the test case, and then uses it to decide the label of the test case. It is worth noting that single rule prediction works for both one-label prediction and multi-label prediction as the chosen rule is already a multi-label rule.

5.6.2 Prediction Based on Groups of Rules

In this configuration, the classifier divides all rules which fully match the test case into groups according to their class labels, and then assigns the test case to the class of the dominant group (the group which has the largest number of rules). This is unlike traditional AC methods such as MCAR and CBA which utilise a single rule for prediction. Several experiments on UCI [103] datasets showed that there is no obvious gain in accuracy using prediction based on groups of rules. However, this predication option is kept in the algorithm for future evaluation for datasets other than UCI. Lastly, in cases when no rules in the classifier are applicable to the test case, the default class will be assigned to that case.

5.7 Incremental Learning

Another advantage of adopting a Line and Space representation of dataset is the ability to do kind of incremental learning while building the classifier model. The learning scheme in MRMCAR can work in an incremental (instance-based) way if all intermediate data generated from the transformations are kept. However, the resulting model needs memory that can fit the dataset several times. Thus, the amount of available memory imposes constraints on it. Other instance-based algorithms with high memory demands may tend to keep data on the hard drive and then adopt sophisticated methods in cashing and indexing to keep the most frequently used part of the data in memory for rapid access time.

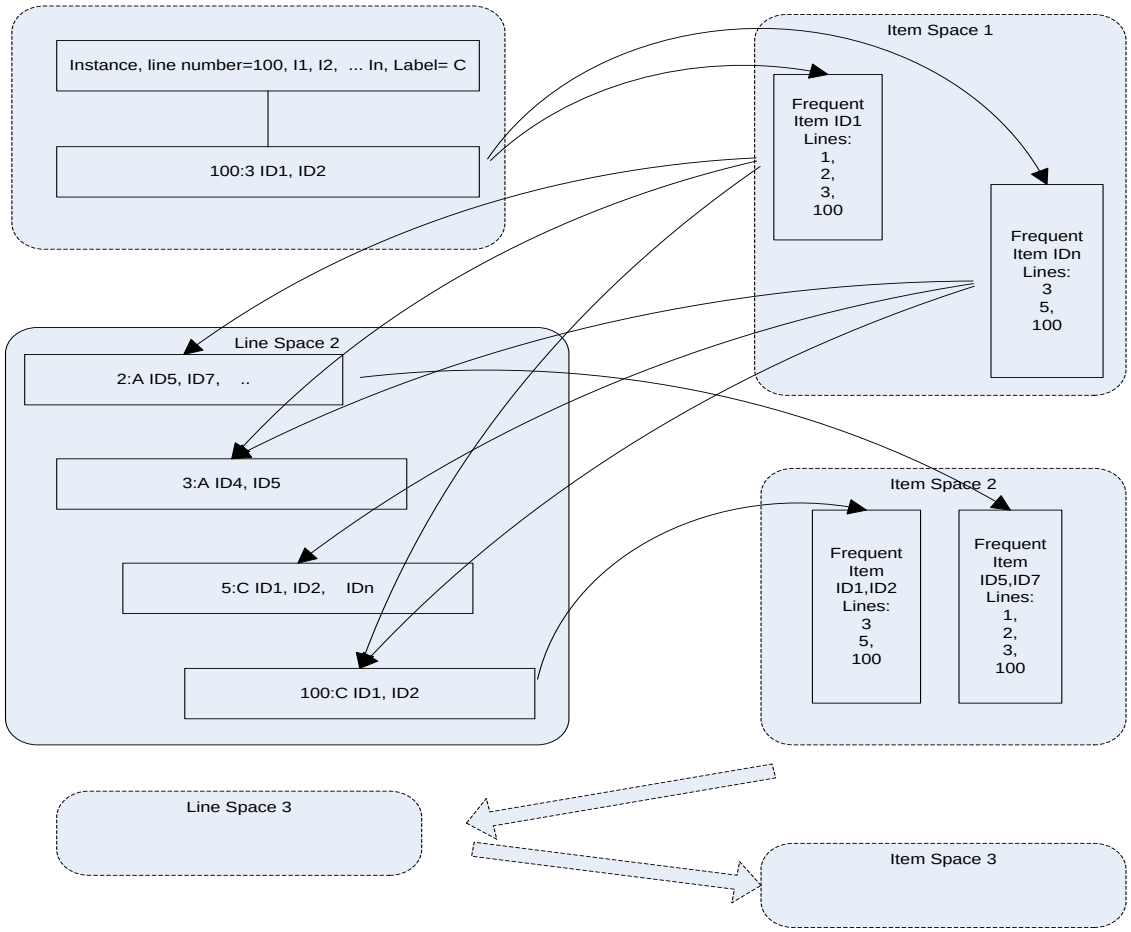


Figure 5-8: Incremental learning in MRMCAR

The MRMCAR intermediate data format is quite useful for incremental learning. It is naturally indexed so that when learning from new instances, the updates in the model are performed directly to positions that need updates with new information. The incremental learning is performed as follows:

5.7.1 Incremental Learning in the Frequent Item Discovery Step

When adding one or a group of instances to the classifier, they are first mapped to their integer values in Line space format. Then the same methods used in ruleitem discovery are used here to update the set of frequent items of size one. Only a few ruleitems are updated. If the added instances contain attribute values that do not exist before in the dataset, then a new ruleitem is created and added to the set of ruleitems. To get the frequent ruleitems of size 2, the algorithm targets only the recently updated frequent items and transforms them to Line space to update a few corresponding lines there. Figure 5-8 shows these procedures.

5.7.2 Incremental Learning in Rule Pruning and Classifier Building Step:

Adding one or more instances may result in adding a new rule in survived rules, or changing the rank value for the pack of rules related to that line. MRMCAR can do incremental learning in rule pruning and classifier building the same way it is done in the frequent item discovery step because all rules are saved in similar data structures of Frequent item space format and Line space format. Thus, a quick transformation is performed on the few targeted rules.

5.7.3 Incremental Learning Constraints and Solutions

In a stand-alone implementation of MRMCAR using WEKA software [42], the major constraint of using incremental learning in MRMCAR is the memory. MRMCAR scans the input dataset only once, and then iterates the process of transforming the data between Line space and Frequent Item space till it finds all the frequent itemsets. Each transformation usually results in lower data sizes because of dropping non-covered lines in the line space and dropping non-surviving ruleitems in the frequent item space. MRMCAR keeps iterating till it reaches the maximum number of iteration or if there are no data left in the current space. The default setting for the MRMCAR algorithm is to free memory from the current space (line or frequent) once the next space transformation is calculated because such data is no longer needed in the calculation. Trying to do incremental learning using MRMCAR means there is a need to keep all the intermediate data transformations for iterations in the Line space and in the frequent item discovery phase. More memory is needed to hold these extra data. The amount of data generated by the data transformations is highly related to the threshold value assigned to the support.

Figure 5-9 and Figure 5-10and show the effect of choosing different support values for the number of lines or frequent items in Line/Frequent-item space for each iteration. Figure 5-9 shows that the number of lines in line space decreases after each iteration. For the Mushroom dataset, the maximum iteration in frequent item discovery is 10, which is equal to the number of attributes excluding the class attribute. The higher the support threshold is, the lower number of lines.

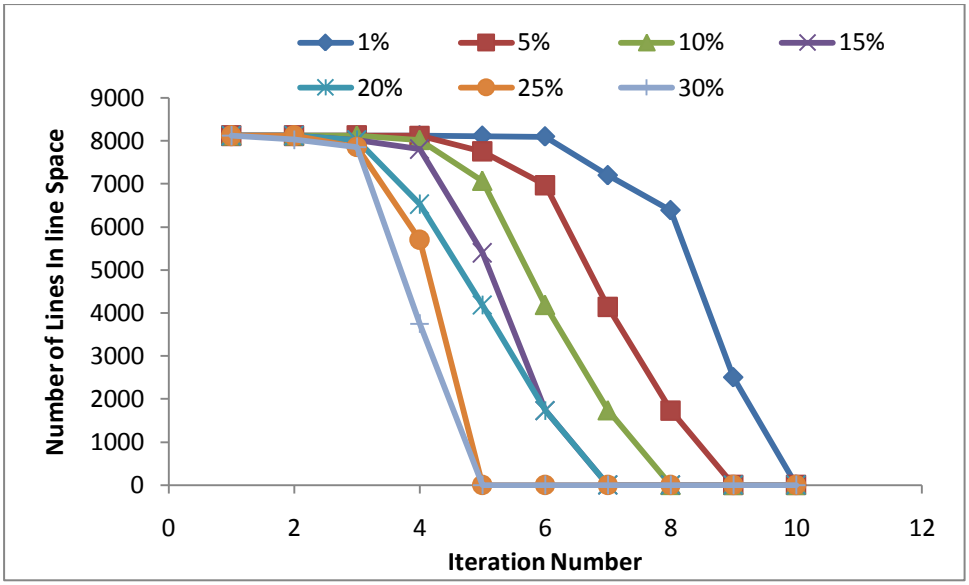


Figure 5-9: Number of Lines in Line Space vs. Iteration in Mushroom dataset for different support levels

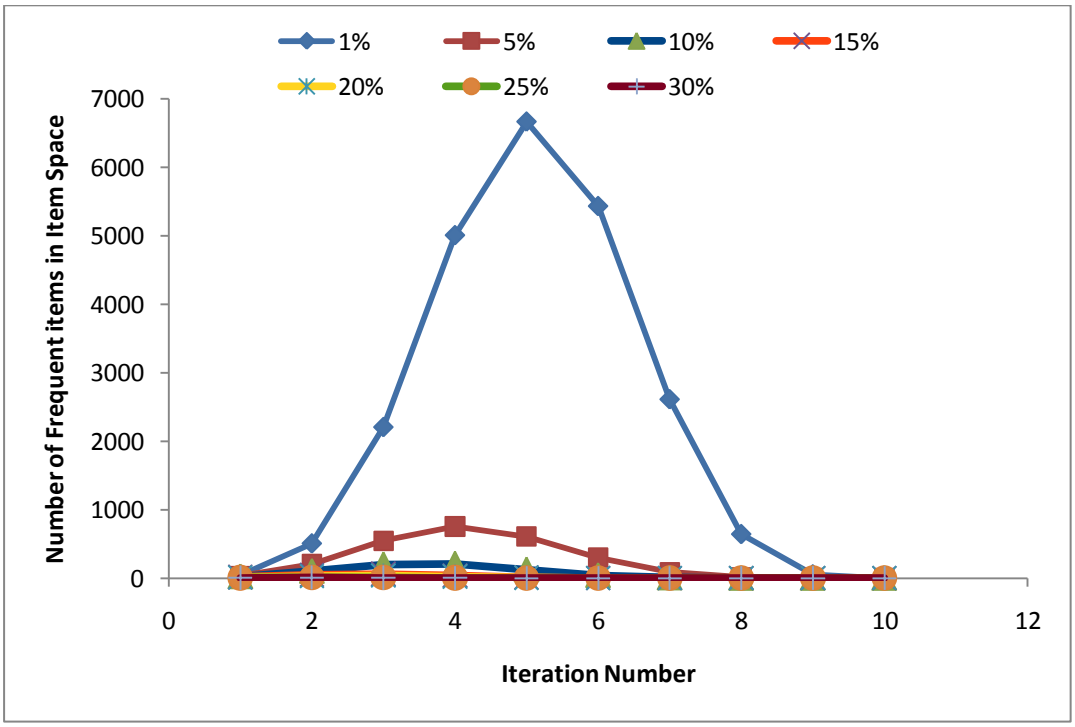


Figure 5-10: Number of Items in Frequent Items space vs. Iteration number for different support levels

Figure 5-10 shows the variation of the number of frequent items generated in each iteration. Different values of support thresholds were used. The maximum number of items increases dramatically for support thresholds less than 5%. Later results on accuracy in the next chapter (5.5.6) show that the maximum accuracy achieved using MRMCAR for each dataset does not always increase for lower support thresholds. This means that MRMCAR can reach optimum memory/time/accuracy performance for range of support and confidence thresholds for

certain input datasets. However, it is difficult to predict the optimum range for each dataset. Currently, user experience in each dataset is the key to tuning the threshold levels to reach such optimization.

Updating only the targeted frequent items in the class model needs indexing and fast access to the data. In-memory stand-alone MRMCAR implementation discussed in 5.2 provides such speed. However, as mentioned before, incremental learning works if the entire intermediate data for both line space and frequent items space were buffered. This works only for datasets of sizes that can fit in computer memory (tens of megabytes for example). Other concerns in in-memory incremental learning are the number of attributes of the data set and the level of the support threshold used for learning. Figure 5-9 and Figure 5-10 show how the number of attributes in data set and how the support threshold can affect the sizes of intermediate data used while discovering the frequent items.

Implementation of MRMCAR in the Hadoop distributed file system allows holding all the data on hard drives. The data sizes used in line and frequent item spaces for all iterations are no longer a problem. Unfortunately, using incremental learning, the Hadoop distributed File system will not provide the required performance. In traditional implementations with no incremental learning, several chunks of datasets are read from hard drives and the instances in each line or item space are processed in a sequential way. But incremental learning demands reading only pieces of data from random places saved on hard drives. Long search times to hard drives are the major constraint of distributed MRMCAR with I/O file system operations.

5.7.4 HBASE Data Structure and Implementation for Incremental Learning:

To solve the problem of low latency access in incremental learning on data saved on hard drives, a prototype using HBase [117] is proposed and implemented to hold the data for MRMCAR with incremental learning configuration. HBase is part of the Apache Hadoop open source project. It was started towards the end of 2006 and was modelled after Google's "Bigtable: A Distributed Storage System for Structured Data" by Chang et al. [5]. It is a distributed column-oriented database built on top of HDFS (Hadoop Distributed File System). HBase is the Hadoop application to use when applications require real-time

read/write random-access to very large datasets. Thus, HBase was chosen over other solutions for several reasons:

- Line space and Frequent Item space are naturally suitable for the HBase table format, because they use a kind of sparse matrix format to hold data. Both spaces are implemented as HBase tables that can spread to billions of records and has a sparse number of columns (attributes) that reaches millions.
- HBase is distributed and can scale very easily; this is suitable for managing massive data sizes over a distributed file system and allows distributed random access to the data.
- HBase can save copies of data efficiently with a time stamp tag. This is very useful for keeping copies of the classifier used in incremental learning for different parts of the input data sets.

5.8 Summary

This chapter started with introducing MRMCAR algorithm for associative classification. It described in details how datasets are represented in MRMCAR in two formats; Line space format (horizontal), and Item space format (vertical). Steps of MRMCAR algorithms were explained with example. This chapter also, described how MRMCAR generate rules of multiple-label predictions with probabilities attached to each label. Several configurations of the algorithm were discussed, especially rule ranking configurations. Using incremental learning in MRMCAR is also presented and limitations were discussed. A practical solution for MRMCAR incremental learning was proposed and implemented were introduced using Google's BigTable data structure.

Chapter 6

Implementation and Evaluation of MRMCAR Classifier Algorithm

6.1 Introduction:

This chapter introduces two implementations of the MRMCAR algorithm. It explains the parameters used with the algorithm. Also, it explained how to distribute MRMCAR and why it is efficient. In 6.4, the measurements used to evaluate MRMCAR are explained with an example using one real case. Section 6.5 shows results of extensive experiments performed using MRMCAR. In addition, it compares the accuracy of the classifier with other existing learning schemes. Finally, section 6.6 shows the time performance of a distributed implementation of MRMCAR on 4 PC machines and it shows other results obtained from the MRSim simulator to investigate the scalability of MRMCAR.

6.2 Sequential Implementation

The algorithm is embedded in Weka software [42]. Weka machine learning software was introduced in 3.3. Weka's class hierarchy was extended with the new MRMCAR classifier. The base class "Classifier" is included in the "weka.classifiers" package which contains implementations of most of the algorithms for classification and numeric prediction. The most important methods overridden in the subclass are "buildClassifier()",

"classifyInstance()", and "distributionForInstance()". More methods are overridden to provide generic information about the class such as documentation, its version, its authors and related papers.

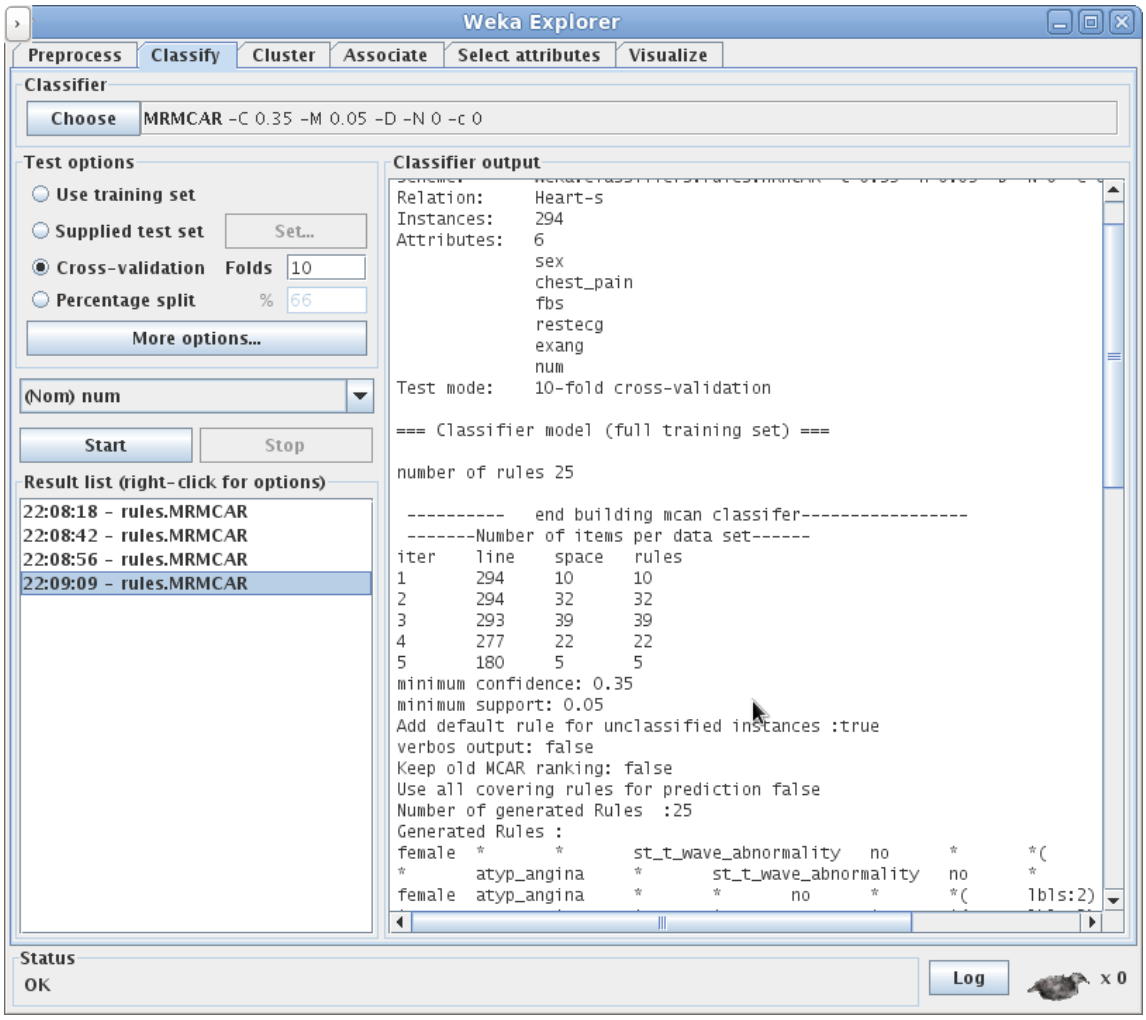


Figure 6-1: MRMCAR in Weka explorer

By embedding the new algorithm in Weka software (Figure 6-1) the MRMCAR benefits from the extensive tools available in WEKA to pre-process, filter, evaluate analysis and visualize the dataset in addition to applying the new MRMCAR learning scheme. Furthermore, Weka has uniform interface to all classifiers. This allows comparing MRMCAR with other classifiers included in the classifiers package.

The MRMCAR classifier has tuneable parameters which are accessed through a property sheet or object editor, as shown in Figure 6-2. A common evaluation module is used to measure the performance of all classifiers.

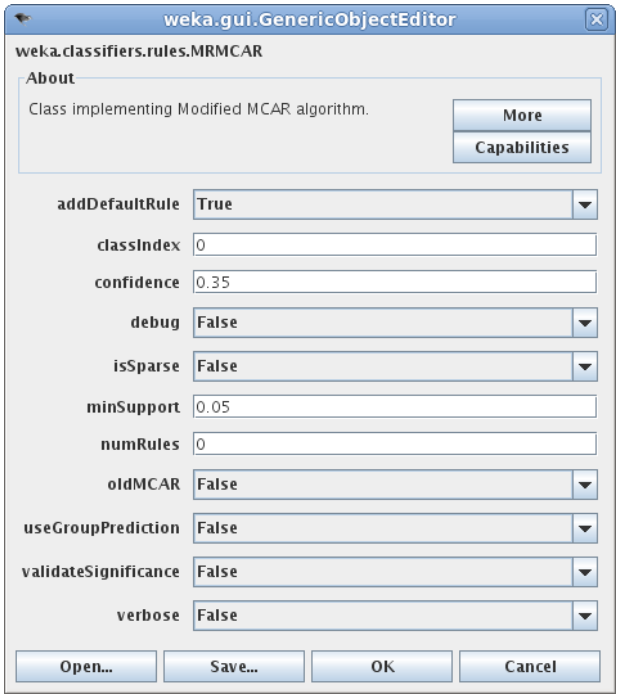


Figure 6-2: MRMCAR object editor form

This is a brief description of Object editor parameters for the MRMCAR classifier:

- Add Default Rule: In the training dataset, add the default rule for instances not covered by any of generated rules.
- Class Index: index of which attribute is used as label.
- Minimum Confidence: confidence threshold for surviving rules to be considered in the classifier model.
- Is Sparse: true if instances in the training dataset have sparse attributes.
- Minimum Support: support threshold for surviving rules to be considered in the training model.
- Number of Rules: build the classifier with support level that generates only this number of rules. The classifier model will start from high support levels then decrease the support in steps till generating the needed number of rules.
- Exact Label Match: True for “exact label match” in the rule pruning step, and false for “any label” pruning.
- Rank Method: choose from CONF_SUPP_ATT, CONF_ATT_SUPP, SUPP_CONF_ATT, SUPP_ATT_CONF, and ATT_CONF_SUPP.

- User group Predication: for predicting new instances. Use true for prediction based on all rules that have similar left sides of the new test instance. Use false for prediction based on single matched rule of the highest rank value.
- Verbose: Used for debugging and algorithm demonstration. If true, then the intermediate data in Item space and Line Space are printed to the output for all iterations. It will also print all surviving rules before pruning.

6.3 Parallel Implementation Using MapReduce

Map reduce implementation is done in the Java programming language. The source code is uploaded to Google code repository and is available on [44]. The Hadoop version used is hadoop-0.20.1 and HBase

6.3.1 Distribution Details

Map-reduce jobs used to implement MRMCAR:

- Initialization: one map-reduce job to map every item to a unique integer value, which is the line number of the first occurrence of the item in the data set. Lines in the data set should be numbered first. A small test on a hard drive of 5400 rpm and one thread application running on Intel Core2 Duo Processor showed that it took around 93 seconds to add line numbers to the lines of a data set of size 3 GB and of number of lines equal to 10 million lines. The result of the initialization step is the data represented in line space.
- Discovering frequent Item sets: starting from frequent item sets of attributes size equal to one. Two map-reduce jobs for each iteration; “toItemSpace” job, and “toLineSpace” job. The number of iteration is passed as parameter for each job.
- Rule Pruning: two Map-Reduce jobs. The first one transfers candidate rules from Item Space to Line space and adds rule rank for each item id. Then, it keeps at most one rule per line. The second map-reduce job is to transform the remaining lines with their remaining ranked rule id to frequent item space to get the final classifier as a list of rules sorted and with multi-labels with probability values attached to each label.
- Prediction: Use the predication function which can be distributed easily because the outcome set of rules for the classifier is of much smaller size than the data set size,

and there is no need to return to the original data set when predicting new instances. Thus several instances of the predication function can work in parallel without the need for network communications.

More details about the implementations are documented in the source code [44].

6.4 Evaluation of the MRMCAR Algorithm

MRMCAR is evaluated using repeated cross-validation. Also, other statistical tests are used to eliminate the effect of chance between different methods. Also, MRMCAR is able to predict the class probabilities rather than classes themselves. Few methods were investigated to help when considering the cost of misclassification without discussing the internals of the algorithm.

6.4.1 Cross-Validation

For classification problems, it is natural to measure a classifier's performance in terms of the error rate. The classifier predicts the class of each instance: if it is correct, it is counted as a success; if not, it is an error. The error rate is just the proportion of errors made over a whole set of instances, and it measures the overall performance of the classifier.

The standard way of predicting the error rate of a learning technique given a single, fixed sample of data is to use stratified 10-fold cross-validation [91] . The data is divided randomly into 10 parts in which the class is represented in approximately the same proportions as in the full dataset. Each part is held out in turn and the learning scheme trained on the remaining nine-tenths; then its error rate is calculated on the holdout set. Thus the learning procedure is executed a total of 10 times on different training sets (each of which have a lot in common). Finally, the 10 error estimates are averaged to get an overall error estimate. Random sampling is done in such a way as to guarantee that each class is properly represented in both training and test sets. This procedure is called *stratification*. Stratification provides a safeguard against uneven representation in training and test sets.

A single 10-fold cross-validation might not be sufficient to get a reliable error estimate because of the effect of random variation in choosing the folds themselves. Stratification reduces the variation, but it certainly does not eliminate it entirely. Thus, MRMCAR is

evaluated using the average of 10-fold cross-validation experiments repeated 10 times with the same dataset for each min-support and min-confidence value to produce reliable results. This involves invoking the MRMCAR algorithm 100 times on datasets that are all nine-tenths the size of the original.

Obtaining the results for the range of confidences and support thresholds and for all data sets and for the range of rule ranking criteria to measure the performance of MRMCAR is a computation-intensive undertaking. Approximately, 10,000,000 MRMCAR runs were carried out. To save time, several Amazon EC2 [37] cloud instances of high hardware capabilities were hired to perform the evaluation experiments. Results are shown later in this chapter.

6.4.2 Predicting Probabilities

In single label prediction, the outcome for each test instance is either correct, if the prediction agrees with the actual value for that instance, or incorrect, if it does not. If the classifier generates either correct or incorrect prediction, then success is the right measure to use. This is sometimes called a 0 - 1 loss function: the "loss" is either zero if the prediction is correct or one if it is not. However, in single label predication, MRMCAR can associate a probability with each prediction, because it knows how many training instances are actually covered by each resulting rule. It might be more natural to take this probability into account when judging correctness in certain applications. For example, a correct test case predicted with a probability of 96% should perhaps weigh more heavily than one predicted with a probability of 53%, and, in a two class situation, perhaps the latter is not all that much better than an incorrect outcome randomly predicted with a probability of 53%. Also, MRMCAR generates - for each test case - predictions of several labels with probabilities attached to each of them.

To compare MRMCAR with other classification algorithms such as C4.5 [25], J48 [43] , RIPPER, CBA [13], and MCAR [15] , MRMCAR used it as single label classifier and did not consider the probability of predicated classes. The 0-1 loss function is used because not all compared algorithms generate probabilities with the predications. This test used no award for a realistic assessment of the likelihood of the prediction. However - even though not evaluated - probabilities in MRMCAR are useful if the prediction is subject to further processing such as involving assessment by a person, or a cost analysis, or perhaps even

serving as input to a second-level learning process. This is an advantage of MRMCAR and depends on the application which MRMCAR is used in.

6.4.3 Counting the Cost

For some applications it is advantageous in evaluating the classifier to consider the cost of wrong decisions, wrong classifications. In such applications it is not sufficient to predict the performance of classifier on only error rate. For example, in a fire detection classifier, the cost of not predicting the fire is far greater than the cost of false alarms generated by the classifier. More measures can be applied to help evaluating the performance of classifier per predicted class. The following measures are by default calculated in each run of WEKA MRMCAR. Take, as an example, one run of the MRMCAR classifier on a Lymph dataset from UCI [103], with min-support = 5%, min-confidence =35% , rule ranking criteria of CONF_SUPP_ATT, and exact label match. Figure 6-3 shows the result of such a run.

Classifier output

```
Time taken to build model: 2.07 seconds

=== Evaluation on training set ===
=== Summary ===

Correctly Classified Instances         119               80.4054 %
Incorrectly Classified Instances        29               19.5946 %
Kappa statistic                          0.6702
Mean absolute error                      0.2167
Root mean squared error                  0.2949
Relative absolute error                 80.9222 %
Root relative squared error             81.022  %
Total Number of Instances              148

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.84      0.03       0.971     0.84      0.901      0.983    metastases
                 0.77      0.046      0.922     0.77      0.839      0.976    malign_lymph
                 1         0.16       0.148     1         0.258      0.993    fibrosis
                 0         0          0         0         0          0.986    normal
Weighted Avg.    0.804     0.04       0.915     0.804     0.846      0.98

=== Confusion Matrix ===

  a  b  c  d   <-- classified as
 68  4  9  0 |  a = metastases
  2 47 12  0 |  b = malign_lymph
  0  0  4  0 |  c = fibrosis
  0  0  2  0 |  d = normal
```

Figure 6-3: Evaluation of one MRMCAR run on Lymph dataset

Details of evaluation are explained in the following sections.

6.4.4 Confusion Matrix

In the two-class case with classes yes and no, a single prediction has the four different possible outcomes shown in Table 6-1.

Table 6-1: Predication results for two class classifier

		Predicted class	
		yes	no
Actual Class	yes	True Positive (TP)	False Negative (FN)
	no	False Positive (FP)	True Negative (TN)

A false positive (FP) occurs when the instance is incorrectly predicted as yes (or positive) when it is actually no (negative). A false negative (FN) occurs when the instance is incorrectly predicted as negative when it is actually positive. True positive rate $TP\ rate = \frac{TP}{TP+FN}$ and false positive rate $FP\ rate = \frac{FP}{FP+TN}$ .The overall success rate is the number of correct classifications divided by the total number of classifications: $Success\ rate = \frac{TP+TN}{TP+TN+FP+FN}$. Finally: $Error\ rate = 1 - success\ rate$.

In a multiclass prediction as in Table 6-3, the result on a test set is displayed as a two dimensional *confusion matrix* with a row and column for each class. Each matrix element shows the number of test examples for which the actual class is the row and the predicted class is the column. Good results correspond to large numbers down the main diagonal and small, ideally zero, off-diagonal elements.

6.4.5 Kappa Statistic

Continuing with the same pervious example,Table 6-2 (MRMCAR) shows MRMCAR prediction with four classes. In this case the test set has 148 instances (the sum of the 16 numbers in the matrix), and (68 + 47 + 4+0=119) of them are predicted correctly, so the success rate is 80.4%.

Table 6-2: Different outcomes of four-class prediction

		Predicted class				
		a	b	c	d	total
Actual class	a	68	4	9	0	81
	b	2	47	12	0	61
	c	0	0	4	0	4
	D	0	0	2	0	2
	total	70	51	25	0	
MRMCAR						

		Predicted class				
		a	b	c	d	total
Actual class	a	38	28	14	0	81
	b	29	22	10	0	61
	c	2	1	1	0	4
	d	1	1	0	0	2
	Total	70	51	25	0	
RANDOM						

To test if this is a fair measure of overall success, the number of agreements expected by chance were tested. MRMCAR predicts a total of 70 a's, 51 b's, 25 c's and 0 d's. To compare MRMCAR in this run with a random predictor that predicts the same total numbers of the four classes shown in Table 6-2 (RANDOM). Its first row divides the 81 a's in the test set into these overall proportions, and other rows do the same thing for the other classes. The row and column totals for this matrix are the same as before—the number of instances hasn't changed, and the random predictor was set to predicts the same number of a's, b's, c's and d's as the actual (MRMCAR) predictor.

This random predictor gets (38 + 21 + 1+ 1 = 60) instances correct. The *Kappa statistic* measure takes this expected figure into account by deducting it from the MRMCAR successes and expressing the result as a proportion of the total for a perfect predictor, to yield (119 - 60 = 59) extra successes out of a possible total of (148 - 60 = 88), or 67.04%. The maximum value of Kappa is 100%, and the expected value for a random predictor with the same column totals is zero. In summary, the Kappa statistic is used to measure the agreement between predicted and observed categorizations of a dataset, while correcting for agreement that occurs by chance.

6.4.6 Numeric Prediction in Evaluation

Continuing with the same pervious example, several alternative measures, are also calculated to evaluate the success of numeric prediction. And this is how the results showed in Figure 6-3 are calculated: If the predicted values on the test instances are p1, p2, . . ., pn ; and the actual values are a1, a2, . . ., an. Then:

Root Mean-squared error = $\sqrt{\frac{(p_1-a_1)^2+\ldots..+(p_n-a_n)^2}{n}}$

Mean absolute error= $\frac{|p_1-a_1|+\ldots..+|p_n-a_n|}{n}$

Relative absolute error = $\frac{|p_1-a_1|+\ldots..+|p_n-a_n|}{|a_1-\bar{a}|+\ldots..+|a_n-\bar{a}|}$ where $\bar{a}=\frac{1}{n}\sum_i a_i$

Root relative squared error = $\sqrt{\frac{(p_1-a_1)^2+\ldots..+(p_n-a_n)^2}{(a_1-\bar{a})^2+\ldots..+(a_n-\bar{a})^2}}$ where $\bar{a}=\frac{1}{n}\sum_i a_i$

MRMCAR works with nominal values, but it is able to predict classes with its probabilities. Thus in this case the error is defined as the difference between the probabilities of the actual classes and predicted classes .Results of calculations are shown in Figure 6-3.

Other measurements are used. Recall, precision, and F-measure is used to evaluate the return of search request in information retrieval domain

$$\text{recall} = \frac{\text{number of documents retrieved that are relevant}}{\text{total number of documents that are relevant}}$$

$$\text{precision} = \frac{\text{number of documents retrieved that are relevant}}{\text{total number of documents that are retrieved}}$$

$$\text{F}-\text{measure} = \frac{2 \times \text{recall} \times \text{precison}}{\text{recall} + \text{precision}}$$

Those measurements can be redefined in analogy to their original definitions to calculate them for MRMCAR classification algorithm.

$$\text{recall} = \frac{\text{number of instances truly predicted with class}}{\text{number of instances that have the class}} = \frac{\text{TP}}{\text{TP}+\text{FN}} = \text{TP rate}$$

$$\text{precision} = \frac{\text{number of instances that truly predicated with the class}}{\text{number of instances predicted with class}} = \frac{\text{TP}}{\text{TP}+\text{FP}}$$

$$\text{F}-\text{measure} = \frac{2.\text{TP}}{2.\text{TP}+\text{FP}+\text{FN}}$$

From confusion matrix for last run:

$$\text{TP rate}_{\text{class a}} = \frac{68}{68+4+9+0} = 0.8395$$

$$\text{FP rate}_{\text{class a}} = \frac{2}{70} = 0.0285$$

$$\text{recall}_{\text{class a}} = \frac{68}{70} = 0.971$$

$$\text{precision}_{\text{class a}} = \frac{68}{81} = 0.8395$$

$$\text{F} - \text{measure}_{\text{class a}} = \frac{2 \times 0.971 \times 0.8395}{0.971 + 0.8395} = 0.9004$$

Calculations for other classes are done the same way.

6.5 Experimental Results

In this section, different classification algorithms are compared with MRMCAR according to different evaluation measures including error rate, number of rules in the classifier, rule pruning impact, and the usefulness of rule ranking. Twenty different data sets shown in Table 6-3 from the UCI data repository [103] have been used in the experiments. The UCI Machine Learning Repository is a collection of databases, domain theories, and data generators that are used by the machine learning community for the empirical analysis of machine learning algorithms. The archive was created as an ftp archive in 1987 by David Aha and fellow graduate students at UC Irvine. Since that time, it has been widely used by students, educators, and researchers all over the world as a primary source of machine learning data sets. As an indication of the impact of the archive, it has been cited over 1000 times, making it one of the top 100 most cited "papers" in all of computer science [103].

The algorithms utilized in the comparison are: C4.5 [25], J48 [43] , RIPPER , CBA[13], and MCAR [15], and the MRMCAR. The reason behind selecting these algorithms is because the different training strategies they employ in discovering the rules. For example, C4.5 uses divide and conquer and RIPPER utilizes heuristic based strategy. On the other hand, CBA is known AC mining algorithms.

Ten-fold cross validation [91] is used as a testing method to derive the error rate numbers. Each Ten-fold cross validation is repeated 10 times with new random partitioning and averages for error rate where derived. The CBA results were produced from an implementation version used in [49], and the J48, C4.5 and RIPPER algorithms results are derived from WEKA open source machine learning tool [42]. The experiments of all learning algorithms were run on Intel Core2 Duo Processor T7300 with 2.0 GHz speed and 4 GB RAM.

The most important threshold in AC is the MinSupp threshold since it controls the number of rules generated, and is only parameter used to decide which frequent item is to survive to the next rule item discovery iteration. So, setting the MinSupp to a large value may result in discarding important knowledge, and setting it to a low value may produce massive numbers of rules, which possibly causes combinatorial explosion. It is the firms believe of the authors that there is no related research works that pointed out the optimum value of the MinSupp threshold since each data set has its own characteristics, thus good results that have been derived from a training data set X using a certain MinSupp may not necessarily means that this MinSupp also works well for data set Y.

6.5.1 Accuracy

Figure 6-4 and Table 6-3 display the error rate figures generated by the different algorithms.

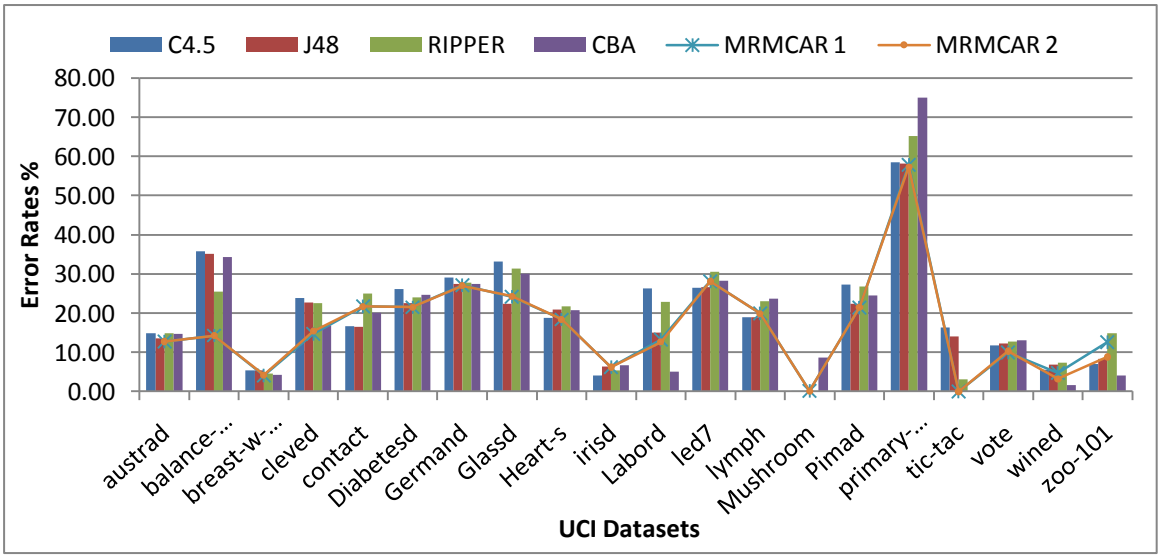


Figure 6-4: Error rate of the classification algorithms against 20 UCI data sets

Two configurations of MRMCAR are used; one is MRMCAR1 with "CONF_ATT_SUPP" rule ranking and with "Exact label match", the other is MRMCAR2 with "CONF_SUPP_ATT" rule ranking and "Exact label match" which is pretty close to MCAR algorithm [15]. It is obvious from the numbers that MRMCAR is highly competitive with regards to error rate if compared with the rest of the algorithms considered. Particularly, MRMCAR1 outperformed the rest of the algorithms on eight data sets, and on average for all data sets considered, it achieved 3.66%, 2.14%, 3.80%, and 2.55% less error rate than C4.5, J48, RIPPER, and CBA, respectively. MRMCAR2 achieved 0.26% less error rate than MRMCAR1.The results clearly indicate that AC algorithms can generate more predictive classifiers than traditional classification algorithms such as C4.5 and RIPPER.

Table 6-3: Error rate in MRMCAR vs. other classification algorithms, MRMCAR1= CONF_ATT_SUPP, MRMCAR2= CONF_SUPP_ATT

Dataset	Size	No. Attributes	Classes	C4.5	J48	RIPPER	CBA	MRMCAR1	MRMCAR2
Austrad	690	14	2	14.79	13.59	14.79	14.64	12.72	12.72
Balance	625	4	3	35.68	35.06	25.44	34.34	14.27	14.27
Breast	699	9	2	5.44	5.41	4.58	4.16	4.12	4.16
Cleved	303	11	2	23.77	22.72	22.45	16.87	14.72	15.38
Contact	24	4	3	16.67	16.50	25.00	20.00	21.67	21.67
Diabetes	768	6	2	26.18	22.54	23.96	24.66	21.47	21.48
German	1000	15	2	29.10	27.42	27.80	27.43	27.09	27.00
Glass	214	7	7	33.18	22.41	31.31	30.11	24.16	24.25
Heart	294	5	2	18.71	20.93	21.77	20.80	18.37	18.37
Iris	150	4	3	4.00	6.33	5.34	6.75	6.20	6.20
Labord	57	12	2	26.32	14.97	22.81	5.01	13.16	12.63
Led7	3200	7	10	26.44	26.61	30.47	28.26	28.10	28.10
Lymph	148	10	4	18.92	18.94	22.98	23.62	19.86	19.86
Mushroom	8124	10	2	0.23	0.20	0.10	8.71	0.14	0.12
Pimad	768	6	2	27.22	22.34	26.70	24.51	21.41	21.45
Primary-tumor	339	11	23	58.41	58.08	65.20	74.89	57.82	57.29
Tic-tac	958	9	2	16.29	13.98	3.03	0.00	0.00	0.00
Vote	435	10	2	11.73	12.16	12.65	13.09	9.89	10.25
Wine	178	13	3	5.62	6.79	7.31	1.67	4.78	3.20
Zoo	101	11	7	6.94	8.36	14.86	4.04	12.57	8.81
Average				20.28	18.77	20.43	19.18	16.63	16.36

6.5.2 Number of Rules

A deeper investigation on the numbers of rules generated by MRMCAR in two configurations; MRMCAR1 using "CONF_SUPP_ATT" with "Any Label Match" , and

MRMCAR2 which is using "CONF_SUPP_ATT" with "Exact Label Match". Tests were carried out against the 20 UCI data collections. Two investigations considered with two scenarios; one using standard support and confidence (MinSupp 5%, MinConf 50%), and one with low support and confidence thresholds (MinSupp 1%, MinConf 10%) since results for this thresholds it is probably indicates the behaviour of both algorithms in normal and sever cases in terms of amount of calculation needed to find frequent items. Figure 6-6 and Figure 6-7 show the number of rules derived by MRMCAR1 and MRMCAR2 according to the support and confidence thresholds mentioned above. Figure 6-5 demonstrates consistency on the numbers of rules for both algorithms with few exceptions. Some of the obvious exceptions are the "German" and the "Led" data sets where in the first case ("German"), the proposed method surprisingly generated 102 more rules than the MRMCAR2 and in the second case ("Led") MRMCAR2 produced 67 more rules than MRMCAR1. Though, both algorithms consistently behaved in a similar way in generating rules for the different data sets considered. Figure 6-7 Depicts the number of rules produced by MRMCAR1 and MRMCAR2 in sever situations particularly when the MinSupp and MinConf are set to very low values, i.e. 1% and 10%, respectively. In this case, MRMCAR2 algorithm produced more rules than MRMCAR1, and specifically it derived more rules on nine data sets. In the remaining eleven data sets, MRMCAR1 generated slightly more rules on five data sets, and both configurations derived the same number of rules on the remaining six data sets. Figure 6-5 displays the difference on the number of rules between MRMCAR1 and MRMCAR2 on all data set using MinSupp and MinConf of 1% and 10%, respectively. The positive values indicate the additional rules generated by MRMCAR2, and the negative values indicate the additional rules produced by MRMCAR1. The figure clearly indicates that MRMCAR2 often generates more rules than MRMCAR1 in circumstances when the support is set to low value by end-user. This increase in the number of rules often leads to an improvement on the classification accuracy as shown in Figure 6-8. In fact, Figure 6-8 reveals that MRMCAR2 algorithm outperformed MRMCAR1 configuration on 14 data sets when the support threshold got lowered to 1%, and achieved on average +1.52% improvement on the classification accuracy than MRMCAR1. It should be noted that on the five data sets which MRMCAR1 generated more rules than MRMCAR2.

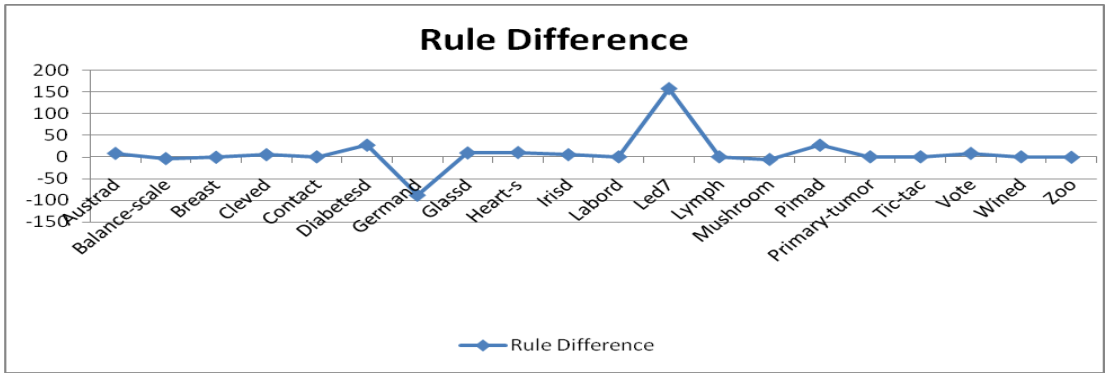


Figure 6-5: the difference of the number of rules derived by MRMCAR1 and MRMCAR2 algorithms

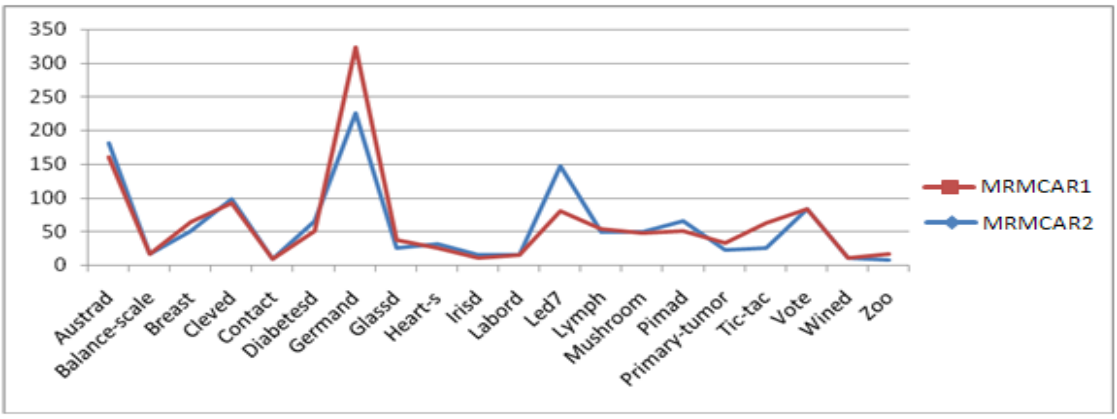


Figure 6-6: Number of rules derived by MRMCAR1 and MRMCAR2 algorithms against 20 UCI data sets with MinSupp 5% and MinConf 50%

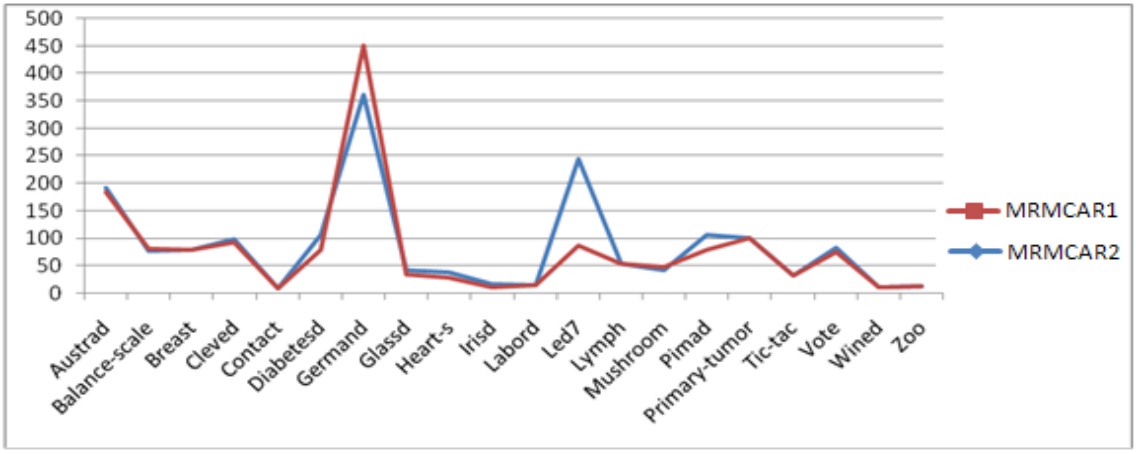


Figure 6-7: Number of rules derived by MCAR and MRMCAR algorithms against 20 UCI data sets with MinSupp 1% and MinConf 10%

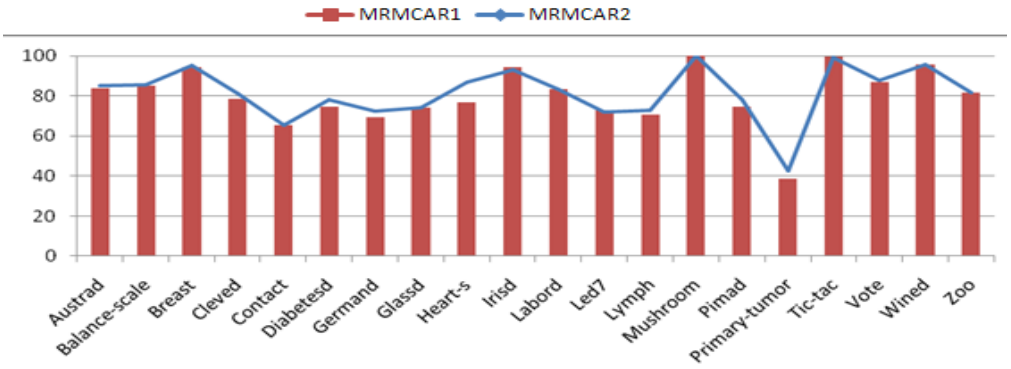


Figure 6-8: Classification accuracy of MRMCAR1 MCAR and MRMCAR2

6.5.3 Confidence vs. Support Effects

To study the sensitivity of algorithm to confidence and support levels, MRMCAR was tested on all datasets with incremental step values for both support and conference thresholds. Accuracy was monitored.

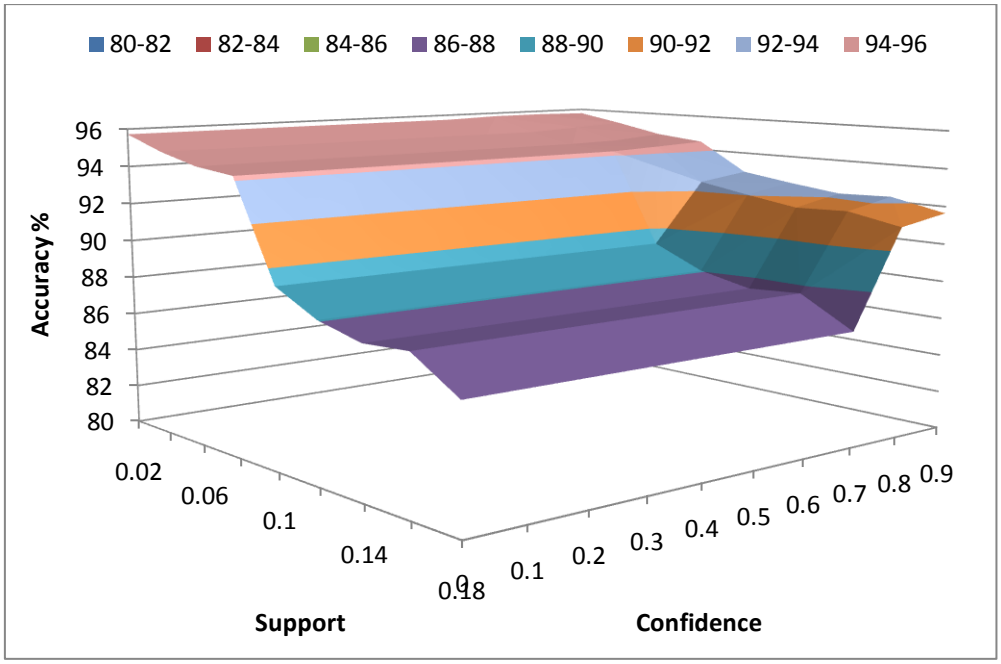


Figure 6-9: Effect of confidence vs. support levels on MRMCAR accuracy, (Breast dataset UCI)

One example of the results is on "Breast" dataset [103]. Figure 6-9 shows MRMCAR accuracy for range of support values 1- 20%, and for confidence values 0-100% and with "Exact Label Match" and "CONF_SUPP_ATT" rule ranking;

All tests on all data sets revealed that MRMCAR is more sensitive to changes in support levels than changes in confidence levels. Accuracy, usually, changes greatly when changing

support levels, but seems fixed for ranges of confidence levels. This behaviour seems constant no matter of label matching used or rule ranking criteria used in MRMCA.

6.5.4 Rule Sorting Effect on Accuracy

This is minimum error rate that MRMCAR achieved using 10 Fold cross-validation test for several ranking criteria and label matching.

Table 6-4: Impact of label matching and rule ranking on maximum accuracy achieved by MRMCAR

Label Match	Exact Label Match				Any Label Match			
Rank	ATT_CONF_SUPP	CONF_ATT_SUPP	CONF_SUPP_ATT	SUPP_ATT_CONF	SUPP_RATT_CONF	ATT_CONF_SUPP	CONF_ATT_SUPP	CONF_SUPP_ATT
Austrad	13.04	12.72	12.72	12.93	12.93	13.39	13.32	13.33
Balance	14.29	14.27	14.27	21.36	21.36	14.83	14.85	14.85
Breast	4.76	4.12	4.16	8.68	8.68	5.09	5.06	4.92
Cleved	17.26	14.72	15.38	17.16	17.10	18.22	16.01	16.50
Contact	15.00	21.67	21.67	22.08	22.92	14.17	20.42	20.42
Diabetes	21.88	21.47	21.48	23.80	23.80	22.85	22.81	22.81
German	27.19	27.09	27.00	27.01	27.02	28.64	27.96	27.79
Glass	25.23	24.16	24.25	26.45	26.54	25.75	24.35	24.35
Heart	17.21	18.37	18.37	19.39	19.29	19.56	19.25	19.25
Iris	6.00	6.20	6.20	4.67	4.67	4.53	4.53	4.53
Labord	13.33	13.16	12.63	16.49	14.74	13.86	13.16	12.63
Led7	26.68	28.10	28.10	30.95	30.88	27.03	26.96	26.96
Lymph	20.95	19.86	19.86	21.15	20.95	20.47	20.20	20.20
Mushroom	0.25	0.14	0.12	2.29	2.29	0.11	0.06	0.03
Pimad	21.98	21.41	21.45	23.78	23.78	22.77	22.75	22.75
Primary-tumor	58.32	57.82	57.29	57.20	57.64	57.76	54.48	54.60
Tic-tac	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Vote	10.46	9.89	10.25	12.34	12.23	12.09	11.84	10.67

Wine	4.72	4.78	3.20	3.03	1.35	5.45	5.34	3.76
Zoo	12.57	12.57	8.81	7.92	6.63	14.06	12.38	9.31
Average	16.56	16.63	16.36	17.93	17.74	17.03	16.79	16.48

- In general, best results are for "exact label matching" and ranking using CONF_ATT_SUPP and CONF_SUPP_ATT ranking criteria.
- MRMCAR achieved 100% accuracy in "Tic-Tac" dataset for all the criteria used for rule ranking.
- Other ranking methods achieved very good accuracies for only one or two datasets. For example, in contact dataset the best accuracy is achieved for any label matching with ranking based on ATT_CONF_SUPP configuration.

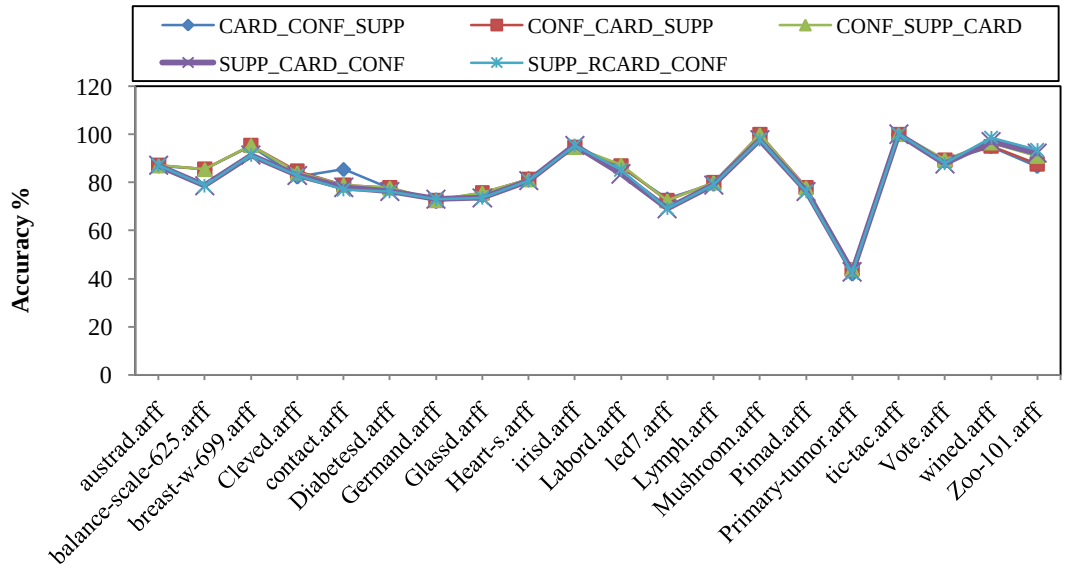


Figure 6-10: Impact of rule sorting on accuracy

6.5.5 Effect of Rule Ranking on Number of Rules in Classifier

All ranking methods used to have close accuracy results. However the number of rules in the classifier is not that close between each ranking methods. The following figure shows the number of rules in the classifier model for each dataset for each ranking method.

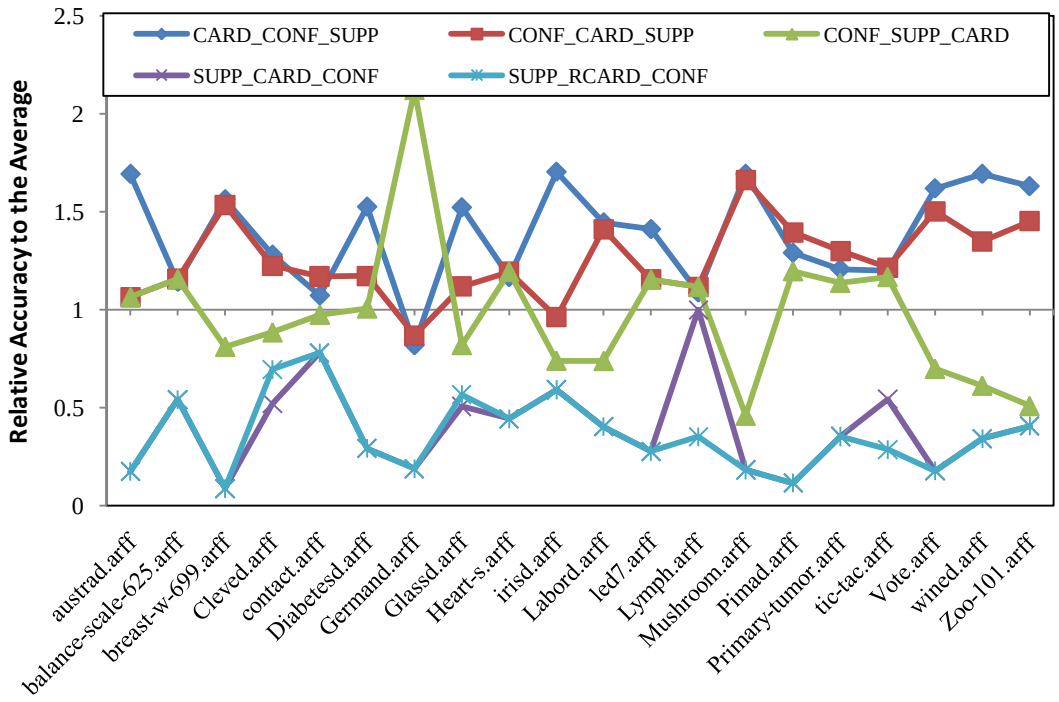


Figure 6-11: Impact of rule sorting on number of rules

The average number of rules was taken for each ranking method for best prediction achieved by this method for certain dataset. Then draw the ratio of each method to its average. All points located lower than value of one means the number of rules generated for certain dataset is below the average for this data set. There is big difference in the number of rules generated by each method. Taking the average rule ratio for whole dataset will generate results shown in Figure 6-12.

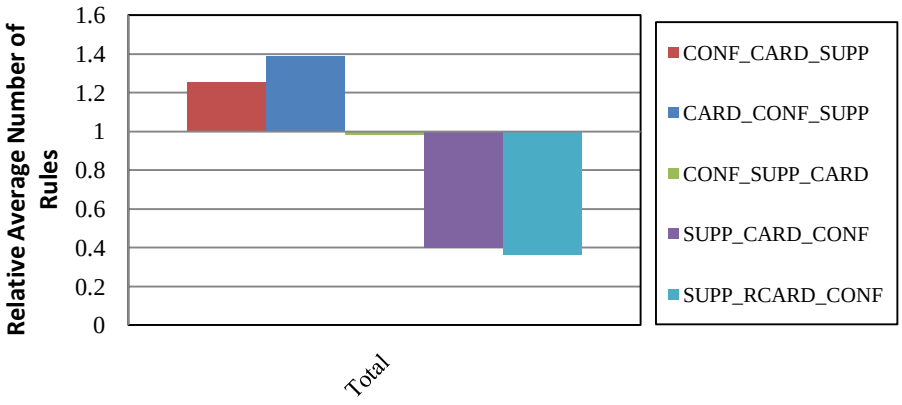


Figure 6-12: Average number of rules for different rule ranking criteria

Results shows that ranking method of CONF_SUPP_ATT usually generate number of rules close to the average.

6.5.6 Support and Confidences for Best Accuracies:

Figure 6-13 shows the result of best thresholds that produced best accuracies in different datasets and using different MRMCAR configurations. Each point is related to cross-validation test done on one dataset using one ranking method. Experiments showed that it is not necessary to lower the support and confidence threshold to get the best performance of algorithms. This is count intuitive to the thinking that lowering the thresholds will increase the algorithm accuracy.

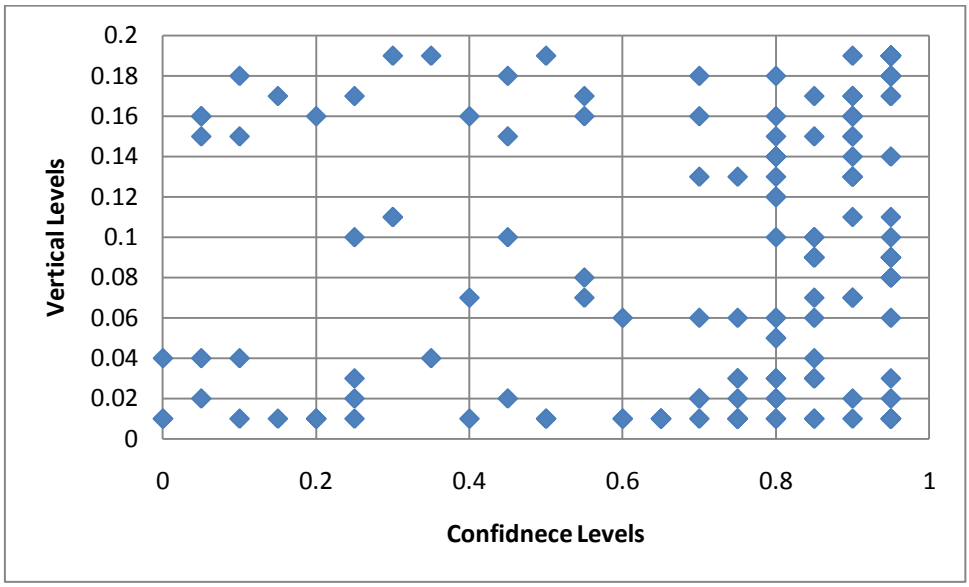


Figure 6-13: Distribution of confidence and support levels for best accuracy using all datasets and all rule ranking criteria

Using clustering techniques, it is possible to find several "centroids" of regions for best threshold values. So if MRMCAR is to be tested in four threshold configurations for example, then the number of regions to set in clustering algorithm is four. Results will show the four points, where x=confidence threshold and y=support thresholds. Then the user can use these values to train the classifier model. Using Weka machine learning software, the centroids of five clusters of the pervious points were calculated. The Results are shown in the following Figure 6-14.

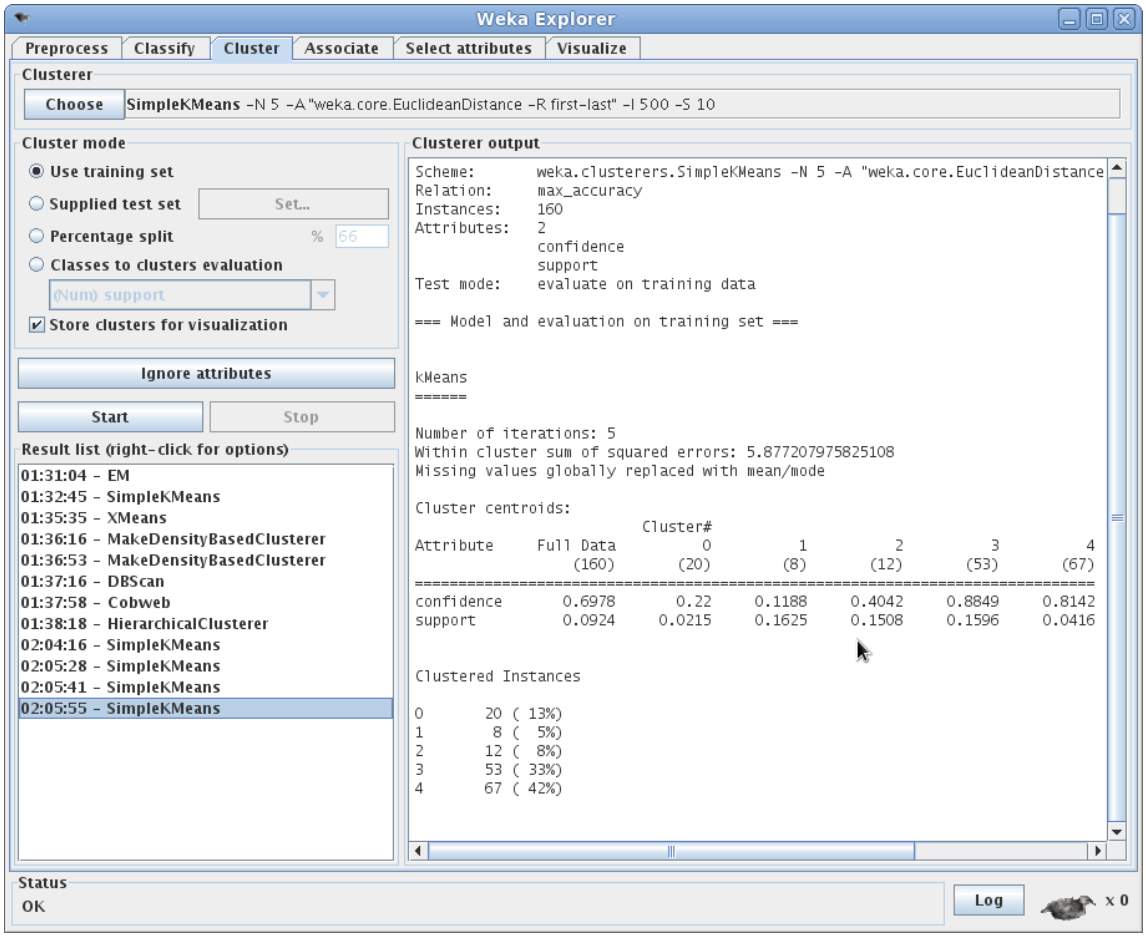


Figure 6-14: Clustering using Weka software

However, user experience with the dataset domain might enable him to choose manually - rather than using clustering techniques - better thresholds to train MRMCAR classifier.

6.6 Performance Evaluation, Scalability, and MRSim Results:

Experiments on cluster of three nodes and on dataset generated from "Mushroom" dataset UCI [103] showed very similar results as in MRApriori (chapter three sections 4.8, 4.8.1, and 4.8.2). This is predictable since MRMCAR is pretty much similar to MRApriori in two things: data representation, and finding frequent itemsets step. Most of time in MRApriori and MRMCAR usually is consumed to find frequent itemsets. When using Mushroom dataset the main difference is MRMCAR is trying to find frequent itemsets from ten attributes whereas the eleventh attribute is used as label.

6.7 Summary

This chapter discussed two implementations of MRMCAR classifier; the sequential and parallel implementation. Also it compared MRMCAR classifier with other classification algorithms of C4.5, J48, and MCAR. Extensive experiments using 10 fold cross-validations were carried out on MRMCAR using twenty datasets from UCI repository. Experiments focused on the optimum threshold values for better predication in addition to the impact of sorting criteria on accuracy and number of generated rules. At last, time performance and scalability evaluation in MRSim were also discussed.

Chapter 7

Conclusions and Future Work

This chapter presents the main conclusions of the thesis and highlights future research work in the related areas.

7.1 Conclusions

This thesis introduces MRSim, the MapReduce simulator that targets a Hadoop environment. This is due to the lack of tools to investigate the algorithms behaviour on MapReduce clusters. MRSim has the ability to predict the resource utilizations and execution times of Hadoop clusters with different configurations. This allows the user to use the simulator to tune the cluster for best optimization for certain algorithms. MRSim uses discrete event simulation to simulate system components. MRSim is using abstract classes to represent the system components allowing new additions to be plugged in the system to simulate different scheduling plans in the cluster environment. MRSim is open source and available for the community to download and to be used for further investigation and development.

The thesis has presented and evaluated MRApriori, a distributed associative rule algorithm that capitalizes on the scalability, parallelism and resiliency of MapReduce for large scale frequent items discovery. MRAPriori introduced a hybrid approach by representing intermediate data in both vertical and horizontal formats. MRApriori keeps reducing the data

(using support thresholds) while transforming the data between the two formats till it discovers all frequent items of long lengths. Then it derives association rules from discovered frequent items. MRApriori generates same number of rules generated by all association rule miners that use same support and confidence concepts. Two implementations of MRApriori have been presented. One is for a sequential algorithm and is available as plug-in to Weka software. The other implementation is for Hadoop MapReduce clusters. MRApriori allows the underlying MapReduce middleware to arbitrarily partition the training dataset into subsets. At the same time, it maintains the prediction accuracy. Optimization of the algorithm can be achieved using fewer configurations in the cluster. Also, MRApriori jobs that run on Hadoop cluster are naturally balanced and can optimize resource utilization in highly heterogeneous computing environments. MRApriori is open source and available for the community to download and to use for further investigation and development.

The thesis has presented and evaluated MRMCAR, a distributed multi-label associate classifier algorithm that capitalizes on the scalability, parallelism and resiliency of MapReduce for large classification based on association rules. MRMCAR uses the same frequent items mining mechanism of MRApriori. MRMCAR allows the underlying middleware to arbitrarily partition the training dataset into subsets while maintaining accuracy. Thus, the MRMCAR algorithm can optimize resource utilization in highly heterogeneous computing environments. MRMCAR showed good accuracy performance compared with several existing traditional classifiers. Several rule ranking methods were introduced and tested thoroughly. However, deciding the best ranking method with the best threshold conditions depends on the type of application and the dataset. MRMCAR produces multi-label classification rules with probabilities. This makes the results ready for further analysis to calculate the cost of classification per class. Two implementations of MRMCAR have been presented. One is for sequential algorithms and is available as a plug-in to Weka software. The other implementation is for Hadoop MapReduce clusters. Incremental learning was discussed and a proto-type was implemented using HBase, a Google Big table data structure. MRMCAR is open source and available for the community to download and to use for further investigation and development.

7.2 Future work

It is necessary to amend the implementation of MRSim to allow a dynamically inserting resource broker to the system. This will open MRSim to investigate the effect of different job scheduling plans on resource utilization and on the cluster's quality of service.

Several parameters in MRSim have to be set before using the simulator. Usually these parameters are set manually and depend on having some experience in real Hadoop cluster environments. However, to open MRSim to less experienced users, a pilot application can be designed and implemented to be run on one instance of cluster nodes. Then the pilot application will extract the best parameters values to set into MRSim simulator. Similar methodology is used in other simulators such as DiskSim hard drive simulators.

In this research a small scale cluster of participating nodes was employed to evaluate the performance of MapReduce based algorithms, in future work algorithms can be evaluated with a much larger cluster such as Amazon Elastic Compute Cloud (EC2).

As part of the future work, a hybrid implementation between in-memory and Hadoop implementations can be achieved. This will allow better execution times as the in-memory part will carry on execution when the current intermediate data is shrunk to fit one computer memory. Also, multi support levels can be introduced to MRApriori as the intermediate data has the clarity and independence to apply different support levels on it per different iteration.

Incremental learning in MRMCAR was discussed in chapter 5. The main constraint was the limitation in memory size needed. One solution was proposed to use the Google BigTable data structure to hold the data and to benefit from its low access time comparing with files I/O operations or comparing with RDBS databases. Further work can be done to using parallel MapReduce tasks over a BigTable data structure to boost the performance of the incremental learning algorithm.

Experiments show the accuracy of MRMCAR is highly dependent on support and confidence threshold levels. The experiments show that there is no obvious pattern of support and confidence thresholds for best accuracies. Each dataset has its own characteristics and performs best in different ranges of support and confidence levels. However, more study is encouraged to use statistical analysis on datasets to estimate the optimum support and

confidence levels. This further study would – at least – generate few generalized recommendations of how to choose the best threshold levels for certain datasets rather than leaving it to user experience.

Sorting criteria in MRMCAR are studied and tested extensively. However, further studies can concentrate on the cost of classification for certain labels for certain criteria.

It is an easy task to investigate using boosting and bagging methods with the MRMCAR classifier. Boosting and bagging can be parallel using the MapReduce framework. However, more consideration is needed for load balancing as boosting and bagging methods generate different accuracy for different partition sizes of the training datasets.

References

[1] O.O. Malley and A.C. Murthy, "Winning a 60 Second Dash with a Yellow Elephant Hadoop implementation," March 2009, URL http://sortbenchmark.org/Yahoo2009.pdf.

[2] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, Jan. 2008, pp. 107–113.

[3] Apache Software Foundation., "Apache Hadoop," Jan. 2010, URL http://hadoop.apache.org/.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies MSST*, 2010, pp. 1-10.

[5] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber, "Bigtable : A Distributed Storage System for Structured Data," *Sports Illustrated*, vol. 26, 2008, pp. 1-26.

[6] Apache Software Foundation, "Apache Mahout", June 2010 URL http://mahout.apache.org/.

[7] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in Action*, Manning Publications, 2010.

[8] R. Buyya and M. Murshed, "GridSim: A Toolkit for the Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing," *Concurrency and Computation Practice and Experience*, vol. 14, 2002, pp. 1175-1220.

[9] H. Casanova, "Simgrid: a toolkit for the simulation of application scheduling," *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 430-437.

[10] G. Wang, A.R. Butt, P. Pandey, and K. Gupta, "Using realistic simulation for performance analysis of mapreduce setups," *Proceedings of the 1st ACM workshop on LargeScale system and application performance LSAP 09*, 2009, pp. 19.

[11] U. Of Southern California, "Network Simulator - ns-2," 2008. URL http://isi.edu/nsnam/ns/.

[12] Apache JIRA, "Mumak Hadoop MapReduce Simulator," 2009. URL https://issues.apache.org/jira/browse/MAPREDUCE-728 .

[13] B. Liu, W. Hsu, and Y. Ma, "Integrating classification and association rule mining," *Knowledge discovery and data mining*, 1998, pp. 80–86.

[14] Z. Tang and Q. Liao, "A New Class Based Associative Classification Algorithm," *Training*, 2007, pp. 1-5.

[15] F. Thabtah, P. Cowling, and Y. Peng, "MCAR: multi-class classification based on association rule," *Computer Systems and Applications, 2005. The 3rd ACS/IEEE International Conference on*, 2005, pp. 33.

[16] F.A. Thabtah, P. Cowling, and Y. Peng, "MMAC: A New Multi-Class, Multi-Label Associative Classification Approach," *Proceedings of the Fourth IEEE International Conference on Data Mining*, 2004, p. 217–224.

[17] J. Pei, J. Han, B. Mortazavi-asl, H. Pinto, Q. Chen, U. Dayal, and M.-chun Hsu, "PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth," 2001, pp. 215--224.

[18] M.J. Zaki and K. Gouda, "Fast vertical mining using diffsets," *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '03*, New York, New York, USA: ACM Press, 2003, pp. 326.

[19] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "New Algorithms for Fast Discovery of Association Rules," *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, vol. 20, 1997, pp. 283--286.

[20] J.S. Park, M.-S. Chen, and P.S. Yu, "An effective hash-based algorithm for mining association rules," *Proceedings of the 1995 ACM SIGMOD international conference on Management of data - SIGMOD '95*, New York, New York, USA: ACM Press, 1995, pp. 175-186.

[21] [1] S. Brin, R. Motwani, J.D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," *Proceedings of the 1997 ACM SIGMOD international conference on Management of data - SIGMOD '97*, New York, New York, USA: ACM Press, 1997, pp. 255-264..

[22] L. Breiman, "Bagging predictors" *Machine Learning*, vol. 24, Aug. 1996, pp. 123-140.

[23] Y. Freund, "Boosting a Weak Learning Algorithm by Majority," *Information and Computation*, vol. 121, Sep. 1995, pp. 256-285.

[24] B. Liu, W. Hsu, and Y. Ma, *Mining association rules with multiple minimum supports*, New York, New York, USA: ACM Press, 1999.

[25] J.R. Quinlan, "C4.5: Programs for Machine Learning," *Morgan Kaufmann San Mateo California*, 1993, pp. 302.

[26] W. Li, J. Han, and J. Pei, "CMAR: Accurate and Efficient Classification Based on Multiple Class-Association Rules," Washington, DC, USA: IEEE Computer Society, 2001, p. 369–376.

[27] X. Yin and J. Han. CPAR: Classification based on predictive association rules. In Proc. of SDM, pp 331–335, 2003.

[28] R. Agrawal and J.C. Shafer, "Parallel mining of association rules," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 8, Dec. 1996, pp. 962-969.

[29] J.S. Park, M.-syan Chen, and P.S. Yu, "Efficient parallel data mining for association rules," *Proceedings of the fourth international conference on Information and knowledge management - CIKM '95*, 1995, pp. 31-36.

[30] H. Kargupta, J. Han, P. Yu, R. Motwani, and V. Kumar, eds., "Next Generation of Data Mining," vol. 7, Dec. 2008, pp. 152-168.

[31] E.-H. Han, G. Karypis, and V. Kumar, "Scalable parallel data mining for association rules," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, 2000, pp. 337-352.

[32] M.J. Zaki, "Scalable Algorithms for Association Mining," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, 2000, p. 372--390.

[33] M.J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li, "Parallel Algorithms for Discovery of Association Rules," *Data Mining and Knowledge Discovery*, vol. 1, Dec. 1997, p. 343–373.

[34] "PoweredBy Hadoop", June 2010 URL http://wiki.apache.org/hadoop/PoweredBy

[35] S. Hammoud, "MRSim: A discrete event based MapReduce simulator," *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, Aug. 2010, pp. 2993-2997.

[36] T. White, *Hadoop: The Definitive Guide*, Yahoo Press, 2009.

[37] "Amazon Elastic Compute Cloud (Amazon EC2). June 2010 URL http://aws.amazon.com/ec2/"

[38] F.A. Thabtah, P. Cowling, and Y. Peng, "MMAC: A New Multi-Class, Multi-Label Associative Classification Approach," *Proceedings of the Fourth IEEE International Conference on Data Mining*, 2004, p. 217–224.

[39] W. Kreutzer, J. Hopkins, and M. van Mierlo, "SimJAVA---a framework for modeling queueing networks in Java," *Proceedings of the 29th conference on Winter simulation - WSC '97*, 1997, pp. 483-488.

[40] A. Sulistio, G. Poduval, R. Buyya, and C.K. Tham, "Constructing A Grid Simulation with Differentiated Network Service Using GridSim," *Computing*, vol. 27, 2005, pp. -302005.

[41] "MRSim. Simulator Project Site" June 2010, URL http://code.google.com/p/mrsim.

[42] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. Witten, "The WEKA data mining software," *ACM SIGKDD Explorations Newsletter*, vol. 11, 2009, pp. 10.

[43] J.R. Quinlan, "Improved Use of Continuous Attributes in C4.5," *Journal of Artificial Intelligence Research*, vol. 4, 1996, pp. 77-90.

[44] S. Hammoud, "DataMiningGrid Project Site ", June 2010 URL http://code.google.com/p/datamininggrid.

[45] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," *Journal of Computer Science and Technology*, vol. 15, 1994, pp. 487-499.

[46] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *Proc of the ACM SIGMOD International Conference on*, vol. 1, 2000, pp. 1-12.

[47] A. Savasere, E. Omiecinski, and S.B. Navathe, "An Efficient Algorithm for Mining Association Rules in Large Databases," San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1995, pp. 432–444.

[48] Y. Yoon and G.G. Lee, "Text Categorization based on Boosting Association Rules," *Second IEEE International Conference on Semantic Computing*, 2008, pp. 136-143.

[49] F. Thabtah, P. Cowling, and S. Hammoud, "Improving rule sorting, predictive accuracy and training time in associative classification," *Expert Systems with Applications*, vol. 31, Aug. 2006, pp. 414-426.

[50] S. Papadimitriou and J. Sun, "DisCo: Distributed Co-clustering with Map-Reduce: A Case Study towards Petabyte-Scale End-to-End Mining," *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 512-521.

[51] C.-T. Chu, S.K. Kim, Y.-A. Lin, and A.Y. Ng, "Map-Reduce for Machine Learning on Multicore," *Architecture*, vol. 19, 2007, p. 281.

[52] G. Kovoor, J. Singer, and M. Luján, "Building a Java MapReduce Framework for Multi-core Architectures."

[1] G. Kovoor, J. Singer, and M. Luján, "Building a Java MapReduce Framework for Multi-core Architectures." In Proceedings of the Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG). Jan 2010.

[53] R.M. Yoo, A. Romano, and C. Kozyrakis, "Phoenix rebirth: Scalable MapReduce on a large-scale shared-memory system," *2009 IEEE International Symposium on Workload Characterization IISWC*, 2009, pp. 198-207.

[54] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems," *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, vol. 0, 2007, pp. 13-24.

[55] "Microsoft Dryad." June 2010 URL http://research.microsoft.com/en-us/projects/dryad/

[56] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad and DryadLINQ Introduction," *ACM SIGOPS Operating Systems Review*, vol. 41, 2007, p. 59.

[57] "Greenplum MapReduce - Bringing Next-Generation Analytics Technology to the Enterprise - Petabyte Database MapReduce." June 2010 URL http://www.greenplum.com/technology/mapreduce

[58] Aster Data "MapReduce, SQL-MR Resources and Hadoop Integration" June 2010 URL http://www.asterdata.com/resources/mapreduce.php#SQLMR

[59] R. Lammel, "Google□ s MapReduce programming model- Revisited," *Science of Computer Programming*, vol. 70, 2008, pp. 1-30.

[60] Apache Software Foundation, "Apache License, Version 2.0," 2004, URL http://www.apache.org/licenses/LICENSE-2.0.

[61] A.R. Butt, P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in MapReduce setups," *2009 IEEE International Symposium on Modeling Analysis Simulation of Computer and Telecommunication Systems*, 2009, pp. 1-11.

[62] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann, 2003.

[63] K. Aida, "Performance Evaluation Model for Scheduling in Global Computing Systems," *International Journal of High Performance Computing Applications*, vol. 14, Aug. 2000, pp. 268-279.

[64] H.J. Song, X. Liu, D. Jakobsen, R. Bhagwan, X. Zhang, K. Taura, and A. Chien, "The MicroGrid: A scientific tool for modeling Computational Grids," *Sci. Program.*, vol. 8, 2000, pp. 127–141.

[65] J. Boulon, A. Rabkin, U.C. Berkeley, A. Konwinski, and M. Yang, "Chukwa : A large-scale monitoring system," *Architecture*, vol. 8, 2008, pp. 1-5.

[66] A. Konwinski, M. Zaharia, R. Katz, and I. Stoica, "Monitoring Hadoop using X-Trace," *Methodology*, 2007.

[67] F. Howell and R. Mcnab, "A DISCRETE EVENT SIMULATION LIBRARY FOR JAVA 1 Introduction 2 Other java simulation environments 4 How to build a simulation," *Computer*, vol. 30, 1998, pp. 51-56.

[68] M. Bateman and S. Bhatti, "TCP testing : How well does ns2 match reality ?," *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, 2010, pp. 276-284.

[69] Y. Liu, M. Li, N.K. Alham, and S. Hammoud, "HSim: A MapReduce simulator in enabling Cloud Computing," *Future Generation Computer Systems*, May. 2011.

[70] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," *ACM SIGMOD Record*, vol. 22, 1993, pp. 207-216.

[71] M.J. Mackinnon and N. Glick, "Data Mining and Knowledge Discovery in Databases - An Overview," *Australian New Zealand Journal of Statistics*, vol. 41, Sep. 1999, pp. 255-275.

[72] I. Pramudiono, T. Shintani, K. Takahashi, and M. Kitsuregawa, "User Behavior Analysis of Location Aware Search Engine," *Proceedings of the Third International Conference on Mobile Data Management*, 2002, pp. 139–145.

[73] M.H. Dunham, "Mining association rules: anti-skew algorithms," *Proceedings 14th International Conference on Data Engineering*, pp. 486-493.

[74] R.J. Bayardo and R. Agrawal, "Mining the most interesting rules," *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '99*, Aug. 1999, pp. 145-154.

[75] J. Li, X. Zhang, G. Dong, K. Ramamohanarao, and Q. Sun, "Efficient Mining of High Confidence Association Rules without Support Thresholds," *Principles of Data Mining and Knowledge Discovery*, vol. 1704, 1999, pp. 406-411.

[76] D. Mining, M. Holsheimer, M. Kersten, H. Mannila, H. Toivonen, I. -x, M.H.M. Kersten, M. Kersten, H. Mannila, and H. Toivonen, "A Perspective on Databases and Data Mining," 1995.

[77] P. Shenoy, J.R. Haritsa, S. Sundarshan, G. Bhalotia, M. Bawa, and D. Shah, "Turbo-charging vertical mining of large databases," *ACM SIGMOD Record*, vol. 29, Jun. 2000, pp. 22-33.

[78] J. Han, J. Pei, Y. Yin, and R. Mao, "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," *Data Mining and Knowledge Discovery*, vol. 8, 2004, pp. 53-87.

[79] K. Wang, Y. He, and D.W. Cheung, "Mining confident rules without support requirement," New York, NY, USA: ACM, 2001, pp. 89–96.

[80] G. Dong, X. Zhang, L. Wong, and J. Li, "CAEP: Classification by aggregating emerging patterns," *Discovery Science*, 1999, pp. 30-42.

[81] J.R. Quinlan, "Generating production rules from decision trees," *Proceedings of the 10th international joint conference on Artificial intelligence - Volume 1*, 1987, pp. 304–307.

[82] W.W. Cohen, "Efficient Pruning Methods for Separate-and-Conquer Rule Learning Systems," *IJCAI93*, 1993, pp. 988-994.

[83] J. Fürnkranz, "Separate-and-Conquer Rule Learning," *Artificial Intelligence Review*, vol. 13, 1999, pp. 3–54.

[84] J. Cendrowska, "PRISM: An algorithm for inducing modular rules," *International Journal of Man-Machine Studies*, vol. 27, Oct. 1987, pp. 349-370.

[85] M. Thompson, R.O. Duda, and P.E. Hart, "Pattern Classification and Scene Analysis," *Leonardo*, vol. 7, Jan. 1974, pp. 370.

[86] M. Thompson, R.O. Duda, and P.E. Hart, "Pattern Classification and Scene Analysis," *Leonardo*, vol. 7, Jan. 1974, p. 370.

[87] D. Meretakis and B. Wüthrich, "Extending naïve Bayes classifiers using long itemsets," *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '99*, New York, New York, USA: ACM Press, 1999, pp. 165-174.

[88] E. Frank and I.H. Witten, "Generating accurate rule sets without global optimization," *Proc 15th International Conf on Machine Learning*, 1998, pp. 144-151.

[89] W.W. Cohen, "Fast Effective Rule Induction," *In Proceedings of the Twelfth International Conference on Machine Learning*, 1995, pp. 115-123.

[90] R.C. Holte, "Very Simple Classification Rules Perform Well on Most Commonly Used Datasets," *Mach. Learn.*, vol. 11, 1993, p. 63–90.

[91] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Morgan Kaufmann, 2000.

[92] J.R. Quinlan, "Discovering rules by induction from large collections of examples," *Expert Systems in the Microelectronic Age*, 1979, pp. 168-201.

[93] J.R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, Mar. 1986, pp. 81-106.

[94] P.E. Utgoff, "Incremental Induction of Decision Trees," *Machine Learning*, vol. 4, 1989, pp. 161-186.

[95] "Data Mining Tools See5 and C5.0," 2009, URL http://www.rulequest.com/see5-info.html.

[96] U.M. Fayyad and K.B. Irani, "Multi-interval discretization of continuous-valued attributes for classification learning," *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, vol. 2, 1993, pp. 1022-1027.

[97] L. Breiman, J.H. Friedman, R.A. Olshen, and C.J. Stone, "Classification and Regression Trees," *Wadsworth International Group*, vol. p, 1984, pp. 368.

[98] J.R. Quinlan, "Simplifying decision trees," *International Journal of Man-Machine Studies*, vol. 27, Sep. 1987, pp. 221–234.

[99] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian Network Classifiers," *Machine Learning*, vol. 29, 1997, pp. 131-163.

[100] J. Fürnkranz and G. Widmer, "Incremental reduced error pruning," *Proceedings of the 25th international conference on Machine learning*, 1994, pp. 70-77.

[101] J.R. Quinlan and R.M. Cameron-Jones, "FOIL: A Midterm Report," *Machine Learning ECML93 European Conference on Machine Learning Proceedings*, vol. 667, 1993, pp. 3-20.

[102] J. Rissanen, "An introduction to the MDL principle," *Helsinki Institute for Information Technology, Tampere*, 2006, pp. 1-10.

[103] C.L. Blake and C.J. Merz, "UCI Repository of machine learning databases," *UCI Repository of Machine Learning Databases*, 1998, URL http://archive.ics.uci.edu/ml/.

[104] F. Thabtah, P. Cowling, and Y. Peng, "MCAR: multi-class classification based on association rule," *Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications*, Washington: IEEE Computer Society, 2005, p. 33–I.

[105] F. Thabtah, "A review of associative classification mining," *The Knowledge Engineering Review*, vol. 22, Mar. 2007, pp. 37–65.

[106] Y. Freund and R.E. Schapire, "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting," *Journal of Computer and System Sciences*, vol. 55, 1997, pp. 119-139.

[107] J. Wang and G. Karypis, "HARMONY: Efficiently Mining the Best Rules for Classification," *Proc of the SIAM International Conference on Data Mining SDM05*, 2005, pp. 205-216.

[108] D.D. Jensen and P.R. Cohen, "Multiple Comparisons in Induction Algorithms," *Machine Learning*, vol. 38, 2000, pp. 309-338.

[109] J. Liu, Y. Pan, K. Wang, and J. Han, "Mining Frequent Item Sets by Opportunistic Projection," *In Proceedings. 2002 International Conference on Knowledge Discovery in Databases (KDD'02*, 2002, pp. 229--238.

[110] A.A. Freitas, "Understanding the crucial differences between classification and discovery of association rules: a position paper," *ACM SIGKDD Explorations Newsletter*, vol. 2, Jun. 2000, pp. 65–69.

[111] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)" URL http://www.ietf.org/rfc/rfc4627.txt.

[112] G.R. Ganger, B.L. Worthington, and Y.N. Patt, "The DiskSim Simulation Environment - Version 2.0 Reference Manual," 1999 URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.3570.

[113] A. Sulistio, U. Cibej, S. Venugopal, B. Robic, and R. Buyya, "A toolkit for modelling and simulating data Grids: an extension to GridSim," *Concurrency and Computation: Practice and Experience*, vol. 20, Sep. 2008, pp. 1591-1609.

[114] O.O. Malley, "TeraByte Sort on Apache Hadoop," 2008,URL sortbenchmark.org/YahooHadoop.pdf.

[115] C. Nyberg, M. Shah, and N. Govindaraju, "Sort Benchmark," 2010, URL http://sortbenchmark.org/.

[116] H.K. Mehta, M. Chandwani, and P. Kanungo, "Performance evaluation of Grid simulators using profilers," *2010 The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, Feb. 2010, pp. 74-78.

[117] R.C. Taylor, "An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics," *BMC Bioinformatics*, vol. 11, 2010, pp. S1.

[118] Oracle Java , June 2010 URL "http://java.sun.com/developer/onlineTraining/collections/Collection.html."

[119] B. Liu, Y. Ma, and C.K. Wong, "Improving an Association Rule Based Classifier," *Principles of Data Mining and Knowledge Discovery*, 2000, pp. 504--509.