Libyan Academy –Misurata

School Of Engineering And

Applied Science

Department Of Information

Technology

Integration of Apriori Algorithm and MapReduce Model to Evaluate Data Processing in Big Data Environment

A Thesis Submitted in Partial Fulfillment of the Requirements

For The Master Degree in Information Technology

By:

Alaa Eldin Mohamed Mahmoud

Supervised by:

Dr. Mohammed Mosbah Elsheh

2019

Acknowledgement

As much as this study has been an individual project, there are still many people who helped to make it possible.

First of all, I would like to thank my mother, my father, my wife and my son Mohammed for their support. I dedicate this work for them and for all my family members.

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 information technology, in particular to my supervisor, Dr. Mohammed Elsheh for his guidance and valuable advice. He gave me his full support even though he had his doubts about the plausibility of my initial idea.

Also, my thanks to all my teachers for providing me assistance and direction whenever I needed it.

I would like to thank all the staff and all my colleagues in the Information Technology Department at the Libyan Academy in Misrata for their support and assistance during this work to make it possible.

Abstract

Searching for frequent patterns in datasets is one of the most important data mining issue. The development of fast and efficient algorithms that can handle large amounts of data becomes a difficult task because of high volume of databases.

The Apriori algorithm is one of the most common and widely used data extraction algorithms. Many algorithms have now been proposed on parallel and distributed platforms to improve the performance of the Apriori algorithm in big data. The problems in most of the distributed framework are the overhead of distributed system management and the lack of a high-level parallel programming language. Also with retinal computing, there are always potential opportunities for node failure that causes multiple re-execution of tasks. These problems can be overcomes through the MapReduce framework.

Most of MapReduce implementations were focused on the technique of MapReduce for Apriori algorithm design. In our thesis the focus is on the size of dataset and the capacity of the Hadoop HDFS, how much the Hadoop system can process simultaneous. Our proposal system Partitioned MapReduce Apriori algorithm (PMRA), aims to solve the latency and even provide solution for small companies or organizations whom want to process their data locally for security or cost reasons.

All of these reasons encourage us to propose this solution trying to solve previous problems. The basic idea behind this research is applying Apriori algorithm using Hadoop MapReduce on a divided dataset, and comparing the results with the same process and dataset performed using Traditional Apriori algorithm.

The obtained results show that, the proposed approach arrive a solution for big data analysis using Apriori algorithm in distributed system by utilize a pre-decision-making.

الملخص

يعد البحث عن أنماط متكررة في البيانات أحد أهم المشكلات في التنقيب عن البيانات. اصبحت تطوير خوارزميات سريعة وفعالة يمكنها التعامل مع كميات كبيرة من البيانات مهمة صعبة بسبب الحجم الكبير لقواعد البيانات.

تعد خوارزمية Apriori واحدة من أكثر خوارزميات التنقيب عن البيانات شيوعًا والأكثر استخدامًا. العديد من الخوارزميات تم اقتراحها على منصات الانظمة المتوازية والموزعة لتحسين أداء خوارزمية Apriori في البيانات الكبيرة. تعد الصعوبات والمشاكل في معظم الانظمة الموزعة هي النفقات العامة لإدارة النظام الموزع وعدم وجود لغة برمجة متوازية عالية المستوى. أيضًا مع الحوسبة الشبكية، هناك دائمًا فرص محتملة لفشل النقاط مما يؤدي إلى إعادة تنفيذ المهام المتعددة. يمكن التغلب على هذه المشكلات من خلال الإطار MapReduce.

ركزت معظم تطبيقات MapReduce على تقنية MapReduce لتصميم خوارزمية Apriori. في أطروحتنا، ينصب تركيزنا على حجم البيانات وسعة ال Hadoop HDFS، وسعة البيانات التي يمكن لنظام Hadoop معالجتها في وقت واحد. يهدف النظام المقترح Partitioned MapReduce Apriori algorithm (PMRA) ، إلى حل التأخير و توفير حلول للشركات الصغيرة أو المؤسسات التي ترغب في معالجة بياناتها محليًا لأسباب تتعلق بالأمان أو التكلفة.

كل هذه الأسباب دفعتنا على اقتراح هذا الحل لمحاولة حل المشاكل السابقة. الفكرة الأساسية وراء هذا البحث هي تطبيق خوارزمية Apriori باستخدام Hadoop MapReduce على مجموعة من البيانات المقسمة، ومقارنة النتائج من نفس العملية ومجموعة البيانات التي يتم تنفيذها باستخدام خوارزمية Apriori التقليدية.

توضح النتائج التي تم الحصول عليها أن النهج المقترح يصل إلى حل لتحليل البيانات الضخمة باستخدام خوارزمية Apriori في النظام الموزع من خلال الاستفادة من عملية اتخاذ القرارات المسبقة.

Table of contents

List of Figures

List of Tables

Chapter One

Introduction

This chapter contains the following subsections: Introduction to big data creation, sources of big data, big data definitions, challenges and opportunities, big data mining, distributes and parallel systems, Hadoop eco-system and problem statement, declaration the major aims of the research by develop a modified approach model to implement Apriori algorithm using MapReduce and evaluation method .

1.1. Background

The amount of data that is generated and stored at the global level is almost inconceivable, and it continues to grow rapidly. This means that there are more potentials for extracting key insights from business information, but only a small fraction of the data is actually analyzed.

Nowadays, the amount of data generated every two days is estimated at five Exabyte's. This quantity of data is similar to the amount of data generated from the dawn of time until 2003. In addition, it was appreciated that 2007 was the first year in which all the data we produce could not be stored. This huge amount of data opens new difficult discovery tasks [1].

The use of data today changes the way we live, work and play. Companies in industries around the world use data to transform themselves into more flexible, improve customer experience, introduce new business models, and develop new sources of competitive advantage. Consumers live in an increasingly digital world, relying on internet and mobile channels to connect with friends and family, access goods and services, and run almost every aspect of their lives, even while they sleep. Much of today's economy is data-driven, and this reliance will only increase in the

future as companies capture, index and criticize data at every step of their supply chain; Social media, entertainment, cloud storage and real-time personal services in the streams of their lives. The result of this increased reliance on data will be an endless expansion of the global DataSphere. Estimated to be 33 ZB in 2018, IDC expects Global DataSphere to grow to 175 ZB by 2025 [2].

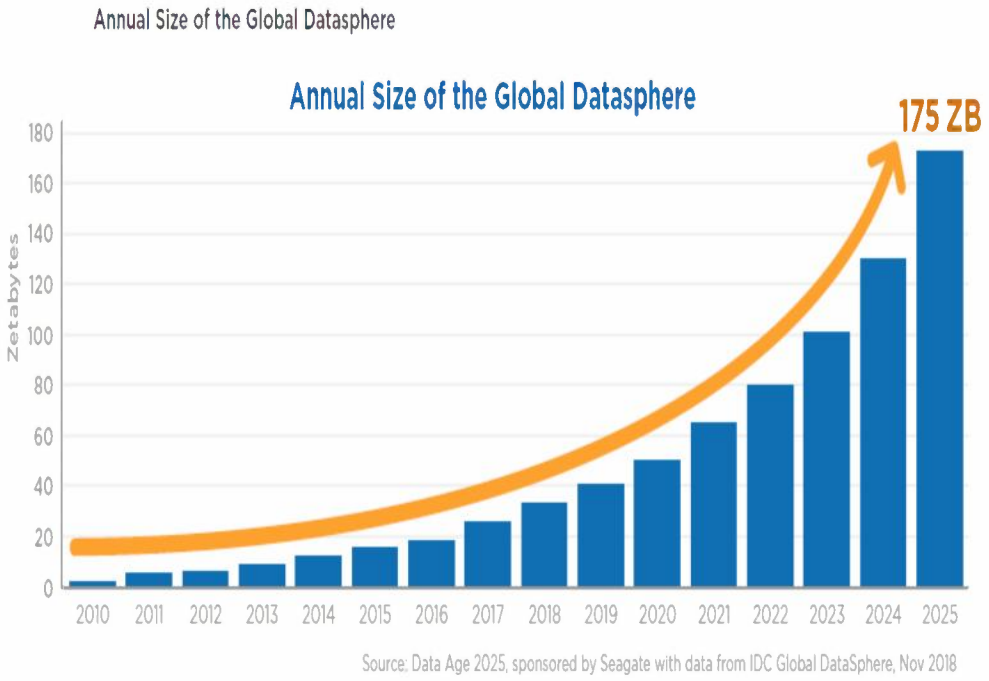


Figure 1.1 Annual Size of the Global DataSphere. (Source: [2])

1.2. Data progression

Traditional database systems are based on structured data. Traditional data is stored in a fixed form or in fields in a file. Examples of structural data including the relational database system (RDBMS, which answer only questions about what happened. The traditional database only provide an idea of a small-scale problem. However, the unstructured metadata used in order to improve and reinforce the ability of an organization to acquisition more insight into the data and also to learn the information [3]. Large (Big) data uses both semi-structured and unstructured data and improves

the diversity (variety) of data collected from different sources such as customers, the public or subscribers [4].

The traditional source of data was personal files, documents, finances, stock records and so on, which entered and stored by workers from the begging of computer technologies until the earlier decade of internet. And with new technologies such as social media, the data starts to come from users. Furthermore, in the last decade the machines start accumulating data (mobile networks, cameras, GPS, scanners, sensors, satellite monitoring, IOT … i.e.).

1.3. Big data definition

From that large amount of data comes new terms, one of them is big data. Big data has several definitions from several groups work on it. One of these groups focuses on the inclusion of their characteristics. When presenting the challenges of data management faced by companies in response to the rise of e-commerce in early 2000's, Doug Laney provided a framework reflecting the three-dimensional increase in data: Volume, Velocity and Variety. The need to draw a new practices involving the "tradeoffs" and architectural solutions that impact application decisions and business strategy decisions [5].

Although this definition did not explicitly mention big data, later the form definition of big data known as " the 3 Vs. ", was linked to the concept of big data and used to define it [6].

Another definition, big data is a collection of datasets with sizes beyond the ability of commonly used software tools to capture, mine, manage and process data within a reasonable time. Big data requires a range of techniques and technologies with new forms of integration to discover complex and large-scale insights from datasets. As of

2012, approximately 2.5 Exabyte's of data are created every day, and this figure doubles almost every 40 months [7]. More data over the internet every second of the internet was stored just 20 years ago. This gives companies a chance to work with many data Petabytes in a single dataset and not just from the Internet.

For example, it estimated that Walmart collects more than 2.5 petabytes of data every hour of customer transactions. A Petabyte is one of quadrillion bytes, or equivalent to about 20 million text storage files. An Exabyte is 1,000 times that amount, or a billion gigabytes. [7].

Gartner [8] defines big data as, "Big data is high volume, high velocity, and/or high variety information assets that require new forms of processing to enhance decision making, insight discovery and process optimization. "The 3 Vs. definition of Gartner is still widely used and is in agreement with a consensual definition that states that "Big Data represents the Information assets characterized by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value"[8].

The 3Vs has been expanded to other complementary characteristics of big data:

Volume: Which means the size of the data. Volume is the V most closely associated with large data, because the volume can be large. The amount of data generated and stored. What we are talking about here is the amount of data that reaches almost incomprehensible proportions. The size of the data determines the potential value and insight whether the data can be considered large or not.

Velocity: Which means speed. Big data is often available in real-time. For many applications, the speed of data creation is more important than size. Actual real-time information allows the company to be more agile than its competitors [7].

Variety: Means diversity, big data draws from text, images, audio, and video. In addition they complement the lost pieces by merging data, in other word it completes missing pieces through data fusion[9].

As we have mentioned earlier, big data has multiple definitions, with the progression in big data, the new dimensions become important and widely used. The importance of the information quality (IQ), with calls for the characterization of large data not only along the three dimensions specified, it has been recently recognized and is called "Vs., volume, variety and velocity, but also along the fourth dimension "V": veracity [10].

Veracity: is the quality of data captured that can vary dramatically and greatly, which affects the accurate careful analysis.

Big data contains many different types of organized structured and non-structured data. Structured data is well defined and can normally be represented as numbers or categories: for example, your income, your age, your gender, and marital status. Unstructured data is not well-defined. It is often difficult to categorize and categorize texts: for example e-mails, blogs, web pages, and transcripts of phone [11].

1.4. Big data Challenges

Anyhow, the importance of big data is not about how much data we have, but what can we do with it. You can take any kind of data from any source and analyze them to find answers that enable you to reduce costs and time, develop new products and offers improve make smart decisions. When you merge large data with high performance analytics, you can accomplish business-related tasks such as:

- Identify the root causes of failures issues and flaws in almost real time.
- Establishment of voucher at the point of sale based on customer's purchase habits.
- Fully recalculate the risk portfolio in minutes.

- Detecting fraudulent behavior before it affects your organization.

1.5. Big data mining

This type of data analysis called Data Mining. It is a way to get undiscovered patterns or facts from a massive huge amount of data in the database. Data mining also known as a one-step in the Knowledge discovery in Databases (KDD). The need for data mining is increased as it helps to reduce cost and increase profits [12]. Data mining is the effective detection of previously unknown patterns in large datasets [13].

Apriori algorithm is one of the most common algorithms in data mining to learn the concept of association rules. It used by many people specifically for transaction operations, and can be used in real-time applications (for example, shop grocery, public store, library, etc.) by collecting the materials purchased by customers over time.

Apriori algorithm is very widely used in data mining application, it has some limitations. It is costly expensive to deal with a large number of candidate sets. It exhausted to scan the database frequently and check the large selection of candidates by matching the pattern, which is especially true for long mining patterns. The Apriori algorithm in general has two major deficiencies. First, you need to scan the database repeatedly and second you need to generate a large number of candidate item set [13].

1.6. Parallel and distributed computing

Unfortunately, in parallel and distributed computing, when the size of a dataset is huge, the memory usage and computational cost can be extremely expensive. In addition, single processor's memory and CPU resources are very limited, which make the Apriori algorithm performance inefficient. Parallel and distributed computing are effective strategies to speed up the performance of algorithms. Parallel and distributed

computing offer a potential solution for the above problems if the efficient and scalable parallel and distributed algorithm can be implemented. Such easy and efficient implementation can be achieved by using Hadoop-MapReduce model[14]. Consider Hadoop as a set of open source programs and procedures (that is, anyone can use or modify, with some exceptions) that anyone can use as backbone for large data operations.

Hadoop is not a type of database, but rather a software ecosystem that allows for massively parallel computing. It is enabled for certain types of distributed NoSQL databases (such as HBase), which can allow the spread data across thousands of servers with a slight decrease in performance. There are four units of Hadoop, each of which performs a specific task that is necessary for a computer system designed for large data analytics. These modules are, Distributed File-System, MapReduce, Hadoop Common and YARN.

MapReduce is named after the two basic operations this module carries out, reading data from the database, putting it into a format suitable for analysis, and performing mathematical operations [15].

Hadoop- MapReduce is a programming model for easy and efficient writing applications, which handles process of a huge amount of data (terabytes or more datasets) in parallel with large clusters of commodity devices in a reliable manner, and fault tolerance. The MapReduce (Task or Job) program partitions (separate) the input dataset into independent partitions, which are processed by map tasks (the Map task for each division) in a completely parallel way. Hadoop framework combines map output and stores it as a set of intermediate key/values pairs that are then fetched as a gateway to reduce tasks [15]. Figure 1.2 shows the Hadoop-MapReduce architecture.

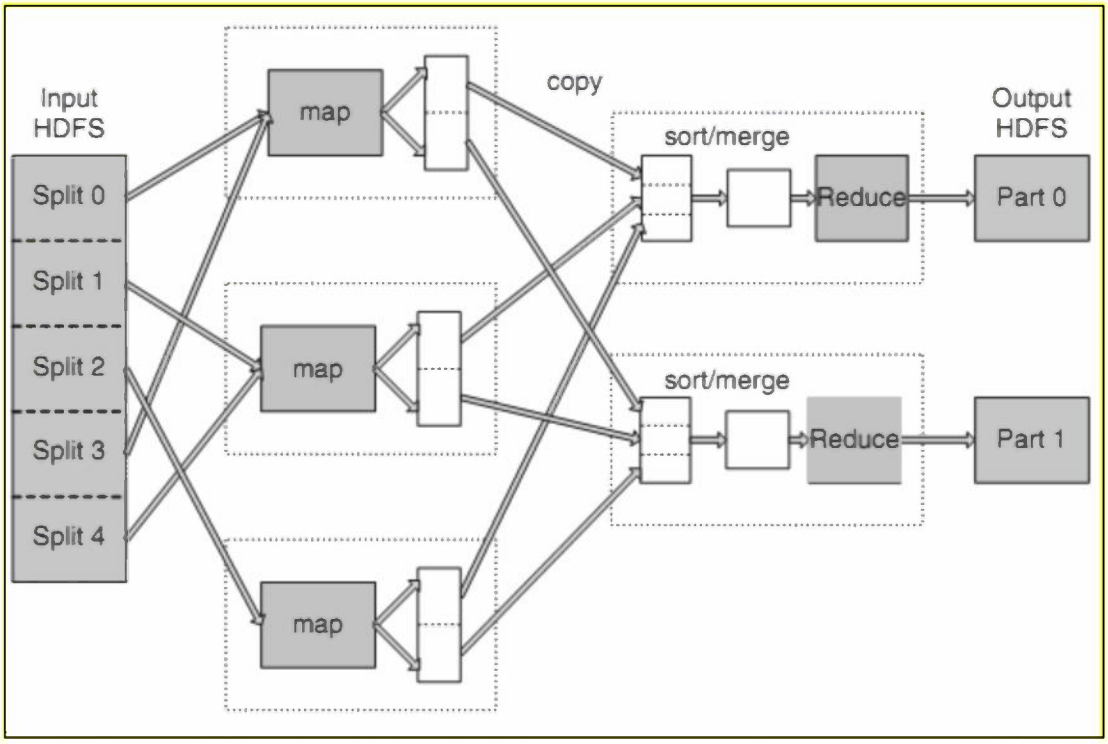


Figure 1.2. Hadoop MapReduce Architecture. (Source: [16]).

1.7. Distributed database

Distributed database system technology (DDBS) is a union of what appears to be have two diametrically opposed approaches to data processing: the database system and computer network technologies. Database systems have taken us from a model processing data in which each application selects and maintains its own data to one where data is centrally defined and managed. This new trend leads to data independence, where the application programs are immune to changes in the logical or physical organization of data, the opposite is true. One of the main motives behind the use of database systems is desire to integrate the operational data of the enterprise and provide centralized, and thus access controls that data. Computer networking technology, on the other hand it promotes a work mode that runs counter to all central efforts. At first glance it might be difficult to understand how these two contrasting approaches can possibly be synthesized to produce a technology that is more powerful and more promising than either one alone [17].

1.8. NoSQL and Big data

NoSQL usually used to store big data. This is a new type of database, which has become more and more popular among the internet companies today.

Types of NoSQL databases:

- A key-value store (also known as a key-value database and key-value store database), which is the simplest type of NoSQL databases. Each item in the database is stored as an attribute name (or "key ") with its value. The most famous databases in this category are Riak, Voldemort, and Redis.
- The Wide column stores data together as columns rather than rows and are optimized for queries across larger datasets. The most popular database in this type are Cassandra and HBase.
- Graph databases which is used to store information about networks, such as social connections. Examples are Neo4J and HyperGraphDB
- Document databases which associate each key with a complex data structure that is known as a document name. Documents can contain many different key-value pairs, key-array pairs, or even nested documents. MongoDB is the most common of these databases. MongoDB is the most popular of all NoSQL databases because it maintains of the best relational database features with the integration of NoSQL.

  The main MongoDB features are: it is an Open Source, Replication, Sharding, Schemaless and Cloud for big data.

In the document database, the database schema idea is dynamic: Each document can contain different fields. This flexibility can be particularly useful for modeling non-structured and polymorphic data. It also makes it easy to evolve an application during development, such as adding new fields. In addition, document databases generally

provide the query power that developers expect from relational databases. Specifically, you can query data based on any fields in a document [18].

1.9. Problem statement

Parallel and distributed computing can be defined as the use of a distributed system to solve one big problem by dividing it into several tasks where each task is counted on individual computers of the distributed system. Hadoop as parallel and distributed system composed of more than one self-routing computer connected over the network. All networked computers connect to each other to achieve a popular goal through the use of their local memory.

From the previous section (1.6) explanation about parallel and distribute computing, we can conclude some drawbacks about Hadoop as parallel and distributed computing.

1. Slow Processing Speed

In Hadoop, with a parallel and distributed algorithm, MapReduce processes of large datasets. There are tasks that you need to execute out Map and Reduce, the MapReduce requires a lot of time to perform these tasks and thus increase latency. To decrease the time and increase processing speed, the data will distributed and processed through the group in MapReduce.

2. No Real-time Data Processing

Apache Hadoop is designed to handle batch processing, which means it takes a huge amount of data into the input, processing and producing the result. Although batch processing is very effective to handle a high amount of data, it depending on the size of the data being processed and the computational power of the system, their output can be significantly delayed. As a conclusion, Hadoop is not suitable for data processing in the real time.

3. Latency

In Hadoop, the MapReduce framework is relatively slower, because it designed to support a different format, structure and large volume of data. MapReduce requires a lot of time to perform these tasks and thus increase latency.

4. Security

Some organizations have security restrictions to process data on the cloud systems, and the local system capacity can not handle processing data in reasonable time. In addition, the high cost of buying and maintaining powerful software, servers and storage hardware that handle the processing of large amounts of data, prevents them to gain benefits from data analysis.

As a result, we need new approaches of data analysis helping in saving time, reducing hardware and software cost.

1.10. Motivation

Agrawal and Srikant proposed the Apriori algorithm in 1994. One of common use of it is market basket analysis. MapReduce designed at Google for use with web indexing technologies. Many approaches implemented in applying Apriori algorithm using Hadoop MapReduce each of which present from different perspective. The proposed approach suggest a new method to apply Apriori algorithm in big data using parallel and distributed computing.

1.11. Aim an objectives

The proposed research aims to develop a modified approach to implement Apriori algorithm using MapReduce. To achieve this aims, the following objectives are specified:

1. Investigate the current state of the art of big data issues and developments.

2. Develop a modified approach to implement Apriori algorithm using MapReduce to save time and reduce hardware cost in processing data by getting a pre-result that will help to make pre-decisions.
3. Evaluate the proposed implementation approach by applying Traditional Apriori algorithm at the same dataset without MapReduce and comparing the results and processing time.

1.12. Structure of the Thesis

The remaining chapters of this thesis are organized as follows:

Chapter 2 discusses several research papers implantations improving Apriori algorithm using Hadoop/MapReduce, and different design implementation.

Chapter 3 describes the methodology of the proposed approach for applying new method in Apriori algorithm using Hadoop MapReduce.

Chapter 4 explores the details of the implementation of for testing the proposed approach.

Chapter 5 the proposed experiments results and evaluating are presented.

Chapter 6 concludes the thesis, it summarizes the observation made through the project and suggest some future avenue research directions.

Chapter Two

Literature Review

Chapter Two

Literature Review

2.1. Introduction

In this chapter, we provide an overview of approaches related to the main topic of this thesis. The first section presents the Apriori algorithm for association rule mining and the improvements that have been done to improve the performance of Apriori algorithm. The second section includes related works using MapReduce as a parallel programming model that is used to manipulate data across large datasets using Apriori algorithm.

Today, huge amounts of data are being collected in many areas, creating new opportunities for understanding meteorological, health, financing and many other sectors. Big data are valuable assets for companies, organizations and even governments. Converting this large data into real treasures requires the support of large data systems and platforms. However, the large volume of big data requires large storage capacity, bandwidth, calculation, and energy consumption. It is expected that unprecedented systems can solve problems arising from items of big data with huge amounts.

Complexity, diversity, often-changing workloads and the rapid development of large data systems pose significant challenges in measuring large data. Without large data standards, it is very difficult for large data owners to decide on which system is best to meet their specific requirements. In addition, they face challenges on how to enhance systems and their solutions to certain or until inclusive workload.

At the same time, researchers are also working on innovative data management systems, hardware architecture, operating systems, and programming systems to improve performance of handling big data.

Data mining means data extraction. It is refers to the activity through large datasets searching for relevant or related information. The idea is that companies collect huge sets

of data that may be homogeneous or automatically collected. Decision makers need to obtain smaller, more specific data from these large groups. They use data mining to uncover a piece of information that will help to drive and assist in charting a course for business. Big data contain a huge amount of data and information and are worth searching in depth. Large data, also known as massive data or collective data, indicate the amount of data involved is too large to be interpreted by human. Currently appropriate technologies are available including data mining, data fusion and integration, machine learning, natural language processing, simulation, time series analysis, and visualization. It is important to find new ways to enhance the effectiveness of big data analysis. With large data analysis solutions and intelligent computing techniques, we face new challenges to make information transparent and understandable.

Frequent sets groups play a key role in many Data Mining tasks that attempt to find interesting patterns of databases, such as association rules, correlations, sequences, loops, classifications, and clusters. The mining of association rules is one of the most common problems of all these Data Mining tasks. Identifying sets of items, products, symptoms, and properties, which often occur together in the selected database, can considered as one of the basic tasks in Data Mining.

The original motiving was to search for repeated (frequent) sets from the need to analyze the so-called supermarket transaction data, which is to examine the behavior of customers in terms of purchased products [19]. Frequent sets of products describe how often items are purchased together.

The important of discovering, detection, figure out and explore all frequent sets is extremely difficult. The search area is rapid and exponential in the total of items that occurrence in the database and the targeted databases tempted to be large, and contain

millions of transactions. Each of these features makes the endeavor to search for the most efficiently and powerful techniques to resolve this task.

2.2. The Apriori algorithm

Apriori algorithm is one of the main algorithms for generating frequent itemsets. The analysis of frequent itemset is a critical step in the analysis of structured data and in the creation of correlation between itemset. This stands as the primary basis for learning under supervised learning, which includes the classifier and feature extraction methods. Applying this algorithm is critical to understanding the behavior of structured data. Most of the data organized in the scientific field is voluminous data.

The processing of this type of huge data requires modern computer hardware. The establishment of such infrastructure is costly. You then need to use a distributed environment such as a clustered setting to handle such scenarios. The distribution of Hadoop MapReduce is one of the cluster frameworks in the distributed environment that helps in distributing huge data across a number of nodes in the frame.

With the introduction of the frequent itemset mining problem, also the first algorithm to solve it was proposed, later denoted as AIS. Shortly after that the algorithm was improved by R. Agrawal and R Srikant [20], and called Apriori. It is a seminal algorithm, which uses an iterative approach known as a level-wise search, where k-itemsets are used to explore (k+1)-itemsets.

2.3. Improvements the efficiency of Apriori

Many discrepancy of the Apriori algorithm have proposed that concentrate on improving the performance of the original Apriori algorithm. Several of these improvements are summarize as follows:

2.3.1. Hash-based

This method attempts to generate large itemsets efficiently and reduces the transaction database size. When generating *L1* (*L1*: First frequent itemset), the algorithm also generates all of the 2-itemsets for each transaction, hashes them to a hash table and keeps a count. As example, when scanning each transaction in the database to generate the frequent 1-itemsets, *L1*, from the candidate 1-itemsets in *C1* (*C1*: candidate itemset), we can generate all of the 2-itemsets for each transaction, hash them into different buckets of a hash table structure and increase the corresponding bucket counts.

The storage data structure in this method is an array and it is suitable for medium size databases. The algorithm was proposed by [21].

2.3.2. Transaction reduction

A transaction that does not contain any frequent itemsets cannot contain any frequent $k+1$ itemsets. Therefore, such a transaction can be marked or removed from further consideration because subsequent scans of the database for j-itemsets, where $j>k$, will not require it.

The storage data structure in this method is an array and it is Suitable for small and medium size databases. The algorithm was proposed by [22].

2.3.3. Partitioning

Partitioning the data to find candidate itemsets. A partitioning technique can be used since it requires just two database scans to mine the frequent itemsets. It consists of two phases. First one, the set of transactions may be divided into a number of disjoint subsets. Then, each partition is searched for frequent itemsets. These frequent itemsets called local frequent itemsets. The storage data structure

in this method is an array and it is more suitable for huge-size databases. Algorithm was proposed by [23].

2.3.4. Sampling

Sampling refers to mining on a subset of a given data. A random sample (usually large enough to fit in the main memory) may be obtained from the overall set of transactions, and the sample is searching for frequent itemset. The essential concept of the sampling approach is to select a random sample *S* of the given data *D*, and therefore search for frequent itemsets in *S* instead of *D*. That way, we swap a certain degree of precision against efficiency. The size of the sample *S* must be convenient that you can perform a search for frequent items in the *S* in the main memory. Because we are looking for repetitive elements (frequent items) in *S* instead of *D*, it is probable that we will miss some of the frequent global elements. To reduce this capability, we use a support threshold below the minimum support to find the local elements that are frequent to *S*.

The storage data structure in this method is an array and it is right fit for all sizes of database. Algorithm was proposed by [24].

2.3.5. Dynamic itemset counting

This method adds the candidate items in different points during the scan procedure. The dynamic method of counting items was suggest in the database being split into marked blocks with starting points. In this format, new candidate itemsets can be added at any starting point, which identify the new candidate only directly before each scan of a complete database. The resulting algorithm requires that you scan a database that is less than Apriori algorithm.

The storage data structure in this method is an array and it is appropriate for small and medium size databases. The algorithm was proposed by [25].

2.3.6. ECLAT ALGORITHM

Éclat algorithm is a depth first search based algorithm. It uses a vertical database layout instance of a horizontal layout, i.e., instead of inserting all transactions explicitly, each item is stored with its cover (also called Tidlist), and the intersection-based approach is used to calculate the support of an itemset.

The storage data structure in this method is an array and it is suitable for medium size and dense datasets but not small size datasets. The algorithm was proposed by [26].

2.4. Related Works Using Hadoop MapReduce

Many different implementations of the MapReduce interface are possible. The right choice depends on the environment. For example, one implementation might be a suitable for a small-shared memory machine, another for a large multi-processor, another for a larger set of network machines.

2.4.1. Parallel implementation of Apriori algorithm based on MapReduce

In study [27], the authors implemented a parallel Apriori algorithm in the context of the MapReduce paradigm. MapReduce is a framework to parallel data processing in a high performing cluster- computing environment.

The parallel implementation of Apriori algorithm based on MapReduce framework was suggest for processing enormous datasets using a large number of computers.

The authors proposed a k-phase parallel Apriori algorithm based on MapReduce. It needs k scans (MapReduce jobs) to find k-frequent items. The algorithm uses two different map functions: one for the first phase and one for rest of the phases. Although the algorithm was successful in finding the k-frequent itemsets using the parallel method, it has a massive amount of reading frequent Itemsets in the previous phase each time of HDFS. The principals of the Apriori algorithm parallel in the MapReduce framework is the

design of the map and reduce the functions of the candidate generation and counting support.

Each mapper calculates each candidate's accounts from its own partition, and then each candidate is ejected and the corresponding number. After the map phase, the candidates are collected, enumerated and grouped in the reduce phase to get partial frequent Itemsets. By using count, distribution between map phase and reduce phase, the communication cost can be decreased as much as possible. Since frequent 1-itemsets has been found in the pass-1 by simple counting of items. Phase-1 of the algorithms are the straight forward. The mapper outputs <item, 1> pair's for each item contained in the transaction. The reducer assemble each enumeration of support for an element, and pairs <item, count> as frequent 1-itemset to L1, when the number is greater than the minimum support. The k-itemsets are passed as an input to the mapper function and the mapper outputs <item, 1>, then the reducer collects all the support counts of an item and outputs the <item, count> pairs as a frequent k-itemset to the Lk. Figure (2.1) shows the flow chart of the parallel Apriori algorithm.

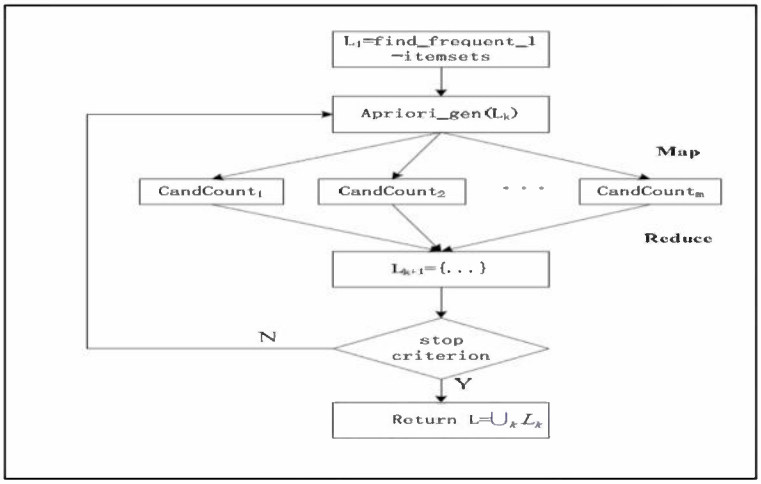


Figure 2.1: The flow chart of the parallel Apriori algorithm

So one-phase class; the algorithm needs only one phase (MapReduce job) to find all frequent k-itemsets, it sounds so good, easy to implement, but its execution time is very slow and its performance is inefficient. In k-phases class (k is maximum length of frequent itemsets), the algorithm needs k phases (MapReduce jobs) to find all frequent k-itemsets, phase one to find frequent 1-itemset, phase two to find frequent 2-itemset, and so on.

In this study, they used the transactional data for an all-electronics branch and the Tl0l4D100K dataset. They replicated it to obtain 1 GB, 2 GB, 4 GB, and 8 GB. For the T10l4D100K dataset, they have replicated it into 2 times, 4 times, and 8 times and got 0.6 GB, 1.2 GB and 2.4 GB datasets, respectively. They denoted those datasets as T10l4D200K, T10l4D400K and Tl0l4D800K. Additionally; they used some transactional logs from a telecommunication company.

The experimental results show that the program is more efficient with the size of the database increasing. Therefore, the proposed algorithm can effectively handle large datasets on commodity devices.

2.4.2. An improved Apriori algorithm based on the Boolean matrix and Hadoop

In other study [28], the researchers proved their improved Apriori algorithm on a theoretical basis. First, they replace the transaction dataset using the Boolean matrix array, by this method, non-frequent item sets can be removed from the matrix, and there is no need to repeatedly scan the original database. You only need to work on the Boolean matrix using the vector "*AND*" operation and the random access properties of the matrix so that it can be generated directly the k- frequent itemsets. The objectives of this study were to find the base frequent itemsets and association rule in transaction database with *min_sup* and *min_conf* pre-defined user on the Hadoop-MapReduce framework.

Typically, the standard Apriori algorithm has many challenges to discovering the frequent itemsets of a massive dataset efficiently and quickly.

We can follow their proposed system form the tables (1, 2, 3, and 4):

1. First the transaction database's data format is vertical data format as table 1.
2. Table 2 is the Boolean expression of the transaction database.
3. In table 3, there are three frequent 1-items I0, I1, and I2.
4. Table 4 the two blocks_block1 and blocks_block2.

Table 1: vertical data format of transactions

Item	TID
I0	T1, T2, T4
I1	T1, T2, T4
I2	T1, T2, T3
I3	T3
I4	T1, T4

Table 2: Boolean Matrix of transaction database

Item	T1	T2	T3	T4	Support
I0	1	1	0	1	3
I1	1	1	0	1	3
I2	1	1	1	0	3
I3	0	0	1	0	1
I4	1	0	0	1	2

Assume the *minsup* = 3. It deletes the items whose support is less than the *minsup* in the Boolean matrix. Table 3 is the new Boolean matrix.

Table 3: The New Boolean Matrix

Item	T1	T2	T3	T4	Support
I0	1	1	0	1	3
I1	1	1	0	1	3
I2	1	1	1	0	3

Table 4: The two blocks_block1 and blocks_block2

Item	T1	T2
I0	1	1
I1	1	1
I2	1	1
I3	0	0
I4	1	0

And

Item	T3	T4
I0	0	1
I1	0	1
I2	1	0
I3	1	0
I4	0	1

Support
3
3
3
1
2

A Boolean matrix is used to replace the transaction database, so non-recurring item groups can be removed from the matrix. It does not need to scan the original database, it just needs to work on The Boolean matrix that uses the vector operation "AND" and the array random access properties so that it can create k-frequent item sets. The algorithm is implemented on the Hadoop platform, and thus can significantly increase the efficiency of the algorithm.

2.4.3. Implementation of parallel Apriori algorithm on Hadoop cluster

In [29], the authors extracted frequent patterns among itemsets in the transaction databases and other repositories reported that Apriori algorithms have a great impact to find repetitive materials using the candidate generation. The Apache Hadoop software framework relies on the MapReduce programming model to enhance the processing of large-scale data on a high performance cluster to handle a huge amount of data in parallel with large scale of computer nodes resulting in reliable, scalable and distributed computing.

Parallel Apriori algorithm was implemented using Apache Hadoop Framework software that improves performance. Hadoop is the program's framework for writing applications that quickly handle huge amounts of data in parallel to large groups of account nodes. Its work is based on the model of the MapReduce.

This single node sheet was implemented by the Hadoop cluster that operates based on the model of the MapReduce. Using this Hadoop cluster, the *wordcount* example was performed. This paper [29], also extracts repetitive patterns between a set of elements in transaction databases or other repositories using the Apriori algorithm in a single node. The Hadoop cluster can easily paralleled and easy to implement. The extracted frequent patterns between items in the transaction databases and other repositories, and they mentioned that the Apriori algorithms have a great impact on finding the Itemsets iterative using the candidate generation.

The authors have improved the Apriori algorithm implementation with MapReduce Programming model as shown below:

- Split the transaction database horizontally into n data subsets and distribute them to '*m*' nodes.
- Each node scans its own datasets and generates a set of candidate itemsets *Cp*
- Then, the support count of each candidate itemset is set to one. This candidate itemset *Cp* is divided into *r* partitions and sent to *r* nodes with their support count. Nodes *r* successively and respectively accumulate the same number of support elements to output the final practical support and identify the recurring *Lp* elements in the section after comparing with *min_sup*.
- Finally merge the output of nodes *r* to generate a set of frequent global itemset *L*.

2.4.4. An Efficient Implementation of A-Priori algorithm based on Hadoop-MapReduce model

In [14], they presents a new implementation of the Apriori algorithm based on the Hadoop-MapReduce model where called the MapReduce Apriori algorithm (MRApriori) was proposed.

They implement an effective MapReduce Apriori algorithm based on the Hadoop-MapReduce model, which only needs two stages to find all the frequent itemsets. They also compared the new algorithm with two existing algorithms that either need one or *K* stages to find the same repetitive elements.

They suggest to use Hadoop Map Reduce programming model for parallel and distributed computing. It is an effective model for writing easy and effective applications where large data sets can be processed on collections of nodes computing, this also in a way that is fault tolerant.

To compare and validate the good performance of the newly proposed two-stage algorithm with the pre-existing phase *I* and *K-phase* scanning algorithms they frequently changing number of transactions and minimum support.

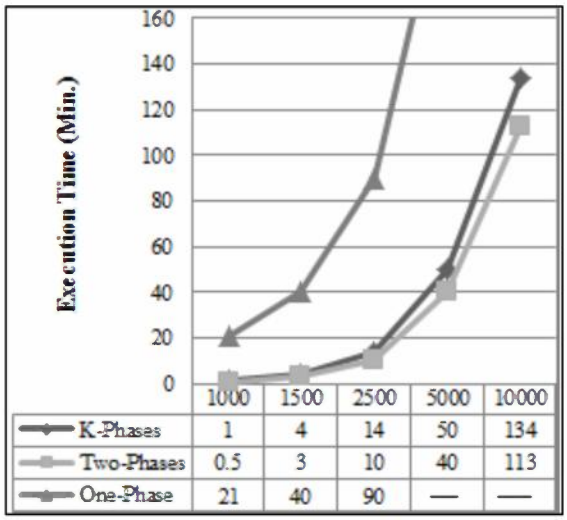
They have introduced the ability to find all the *K-iterative* elements within only two stages of scanning and implementing the entire set of data in the MapReduce Apriori algorithm on the Hadoop MapReduce model efficiently and effectively compared with phase I algorithms and K-phase algorithms.

In study [14], the tl0l4d100k data set was used to obtain the results of the experiment generated by the IBM's quest synthetic data generator. The total number of transactions is 100000, and each transaction contains 10 items on average. The complete number of items is 1000, and the average length of the frequent itemsets is four.

They evaluated the performance of their proposed algorithm (MRApriori) by comparing the implementation time with the other two existing algorithms (one and K-stages).

In study [14], the author implement the Apriori algorithm on a single device or can say stand-alone so there is some chance to execute on a multiple node. Three algorithms have been implemented; MRApriori and the other exists two algorithms are present (one and

K stages) based on the Hadoop MapReduce programming model on the platform is working on a standalone mode and comparing the performance of those algorithms. Figure 2.4. Shows algorithms Performance with different datasets.



	1000	1500	2500	5000	10000
K-Phases	1	4	14	50	134
Two-Phases	0.5	3	10	40	113
One-Phase	21	40	90	—	—

Figure 2.4. Algorithms Performance with Different Datasets

They claimed that the results showed that: one-phase algorithm is ineffective and not practical, K- phases is effective algorithm and its implementation time close to their proposed algorithm. The experiments have been conducted on one machine and the combination of production records has not moved from a map factor to reduce the operator over the network. Their proposed algorithm, MRApriori, efficient and superior to the other 2 algorithm in all experiments.

The empirical results showed that the proposed Apriori algorithm is effective and exceeds the other two algorithms. This study provides insight into the implementation of the Apriori over MapReduce model and suggest a new algorithm called *MRApriori* (MapReduce Apriori Algorithm).

2.4.5. Improving Apriori algorithm to get better performance with cloud computing

The study on [30], claims that Apriori algorithm is a famous algorithm for association rule mining and that the traditional Apriori algorithm is not suitable for the cloud-computing model because it was not designed for parallel and distributed computing.

Cloud computing has become a big name in the current era with probability to be main core of most future technologies. It has been proven that mining techniques implemented with cloud computing model can be useful for analyzing huge data in the cloud. In study [30], researchers used the Apriori algorithm for association rules in cloud environments. So in this study[30], they optimize the Apriori algorithm that is used on the cloud platform. The current implementations have a drawback that they do not scale linearly as the number of records increases, and the execution time increases when a higher value of k-itemsets is required.

The authors try to overcome the above limitations, and they have improved the Apriori algorithm such that it now has the following features:

1. The linear scale will have a number of records increases.
2. The time taken is proximate of the value *K*. This is anything K-itemsets running appears, it will take the same time to given the number of records.

The implementation time of the existing Apriori algorithm increases exponentially with a decrease in the number of support.

Hence, in order to minimize desired string comparisons and possibly one of the obstacles in previous implementation processes that do not attempt to get out in two steps, they will now implement a custom key format that would take the same set as the key instead of text/string. This will be achieved using the Java Collection library.

The improvement of the Apriori algorithm on Amazon *EC2* (Amazon Elastic Compute Cloud) has been implemented to assess performance. Data entry and application files have

been saved on *S3* (Amazon Simple Storage Service), which is the data storage service. Data transfer between Amazon *S3* and Amazon *EC2* is free making *S3* attractive to users of the *EC2*. Output data is also written in *S3* buckets at the end. The temporary data is written in the HDVs files.

Amazon Elastic MapReduce takes care of the provision of the Hadoop cluster, running the job flow, terminating the job flow, transferring data between Amazon *EC2* and Amazon *S3*, and optimizing the Hadoop. Amazon command removes most difficulties associated with configuring Hadoop, such as creating devices and networks required by The Hadoop group, including the setup monitor, configuration of Hadoop, and the execution of the job flow.

Hadoop job flows are using the Cloud Service command, *EC2* and *S3* cloud. To start the task, a request sent from Host for order. Then, after creating the Hadoop block with the main instances and the slave. This group is doing everything, treatment in the job. Temporary files created during task execution and output files are stored on the *S3*. Once the task is completed, a message sent to the user.

Cloud computing is the next development of online computing which provides cost effective solutions for storage and analyze a huge amount of data. Extracting data on a cloud computing model can greatly benefit us. That is why data extraction technology implemented on the cloud platform of many of our data extraction techniques

The association rule-mining base used as a data mining technology. The Apriori algorithm has been improved to fit that for a parallel account platform. Using Amazon Web Services *EC2*, *S3* and order for cloud computing, the proposed algorithm reduces the execution time of values less than the support count, the authors did not mention or explain the algorithm also the result was unclear.

The current implementation processes has some disadvantage that:

1. They did not record in writing as the number of records increases.
2. The execution time increases when a value higher than k-itemsets needed.

2.5. Summary

In this chapter, we presented a review of existing works closely related to proposed research and identified some drawbacks of existing approaches. From the previous works, there is a need to work on enhancing the performance of Apriori by implementing it in parallel using MapReduce.

Here is a review of the previous improved Apriori algorithms on Hadoop-MapReduce.

Ref	Methodology	Description	Storage data structure	Dataset size
[27]	To evaluate the performance of their study in terms of size-up, speedup, and scale-up to address massive-scale datasets.	In this study, the authors implemented a parallel Apriori algorithm in the context of the MapReduce paradigm. MapReduce is a framework for parallel data processing in a high-performance cluster-computing environment. .	Transactional data for an all-electronics branch and the T1014D100K dataset. They replicated it to obtain 1 GB, 2 Gb, 4 GB, and 8 GB.	Large dataset
[28]	Improved Apriori algorithm on a theoretical basis. First, they replace the transaction dataset using the Boolean matrix array.	The aims of this study were to find the frequent itemsets and association rule in the transactional database with the *min_sup* and *min_conf*.	Sample	Small and mediu m dataset
[29]	Improved the Apriori algorithm by split the transaction database.	Implemented a revised Apriori algorithm to extract frequent pattern itemsets from transactional databases based on	Word count Example	Small and mediu m dataset

		the Hadoop-MapReduce framework. They used the single-node Hadoop cluster mode to evaluate the performance.		
[14]	To compare and prove the good performance of the newly proposed 2-phase algorithm with previously existing 1-phase and k-phase scanning algorithms repeatedly changing the number of transaction and minimum support.	They introduced the ability to find all k-frequent itemsets within only two phases of scanning the entire dataset and implemented that in a MapReduce Apriori algorithm on the Hadoop-MapReduce model efficiently and effectively compared with the 1-phase and k-phase algorithms.	Dataset was used to obtain the experiment results generated by IBM' s quest synthetic data generator.	Mediu m and large dataset
[30]	Traditional Apriori algorithm is not suitable for the cloud-computing paradigm because it was not designed for parallel and distributed computing.	Appling Data mining techniques implemented with the cloud-computing paradigm can be useful for analyzing big data in the cloud.	INA (Information Not Available)	Large dataset
Proposed Approach PMRA	Integration of Apriori algorithm and MapReduce model to evaluate Data processing in big data environment by dividing dataset before pass it to the HDFS.	Hence, Hadoop MapReduce depend on HDFS system to split the data, the HDFS size capacity effect the process time, by dividing the dataset to fit to the HDFS according to the HDFS block size and the number of datanodes helps to speed up the process and avoid latency.	Grocery store sales dataset	Large dataset

Chapter Three

The proposed system

Chapter Three

The proposed system

3.1. Introduction

This thesis focuses on usage of Apriori algorithm in combination with MapReduce Hadoop system. The advantage of Apriori algorithm using MapReduce Hadoop system will be more faster to process a large amount of data in parallel computing which is the main purpose of Hadoop.

By working with a large number of computing nodes in the cluster network or grid, a potential opportunity for the node to fail is expected, that causes many tasks to be re-performed. On the other hand, the Message Pass Interface (MPI) represents the most common framework for distributed scientific computing, but only works with low-level language such as C and Fortran. All these problems can be overcome through the MapReduce framework developed by Google. MapReduce is a simplified programming model for processing widely distributed data and also used in cloud computing. Hadoop is a Google MapReduce environment from Apache that is available as an open source [31].

The research methodology is based on studying and implementing the Apriori algorithm MapReduce approach, observing the performance of the algorithm with several parameters.

To overcome all drawbacks of previous models [27,28,29,14,30]; this study proposed a new approach of apriori algorithm using MapReduce based on parallel approach model. This approach model is built on merging of two models of Apriori algorithms:

1) Sampling.
2) Partitioning.

3.2. Apriori algorithm models.

One of the most popular algorithm in Market Basket analysis is Apriori algorithm. In order to develop the Apriori algorithm, to reach the best results and to avoid the defects there were many implementations developing of the Apriori algorithm.

3.2.1. Sampling model

Sampling refers to mining on a subset of a given data. A random sample (usually large enough to fit in the main memory) may be obtain from the overall set of transactions, and the sample is searched for frequent itemset.

- Sampling can reduce I/O costs by drastically shrinking the number of transaction to be considered.
- Sampling can provide great accuracy with respect to the association rules.

3.2.2. Partitioning model

Partitioning the data to find candidate itemsets. A partitioning model technique can be used that requires just two database scans to mine the frequent itemsets.

3.3. Apriori algorithm on Hadoop MapReduce

To apply Apriori algorithm to a MapReduce framework, the main tasks are to design two independent Map function and Reduce function. The functionality of the algorithm is converting the datasets into pairs (key, value). In MapReduce programming model, all mapper and reducer are implemented on different machines in parallel way, but the final result is obtained only after the reducer is finished. If the algorithm is repetitive, we have to implement a multiple phase of the Map-Reduce to get the final result [32].

3.4. The proposed model Partitioned MapReduce Apriori algorithm (PMRA).

The basic idea behind proposed model is a combination of two Apriori algorithm models sampling and partitioning. It goes through several stages, starting with splitting the dataset into several parts (partitioning), and using MapReduce function for applying Apriori

algorithm on each partition separately from other parts (sampling). The proposed model (PMRA) gives pre-results from each partition. These results are changed after each MapReduce completes the processing; this is because the system will add the new results to the one before. This pre-result gives the ability to make decisions faster than applying the whole Apriori algorithm on the complete dataset.

The size of each partition must fit to the Hadoop system, so we must understand how Hadoop distributed file system HDFS work. HDFS is designed to support massive large files. HDFS-compliant applications are those deals with large datasets. These applications write their data only once but read the one or more times and need to satisfy these readings at flow speeds. HDFS supports semantics write-once-read-many to files. The typical block size that HDFS uses is 64 Megabytes (MB). Thus, the HDFS file has been divided into 64 MB chunks, and if possible, each part will have a different datanode.

Each single block is processed by one mapper at a time. Therefore, if we have N datanodes that mean we need N mapper and this will take more time if we do not have enough processors to run N maps in parallel.

From the previous impediment, the proposed approach partitions out large dataset to several partitions of datasets then those, partitions are send to the Hadoop MapReduce Apriori implementation. In addition, the result will not eliminate any item, it keeps all the results waiting for the other partitions finishing process and add its result to the results table and count the items again to give us the new result.

The Hadoop system works here as a parallel and distributed system if the system have one node or more even one cluster or more.

3.5. Hadoop MapReduce-Apriori proposed model (PMRA)

Suppose we know the number of nodes and we wanted to send the partitions to fit to these nodes, so each datanode will have only one block to run at a time, here in Hadoop by default will be 64MB per block for each node.

3.6. The PMRA process steps:

1. Count how many nodes in your Hadoop system.
2. Partitioning the dataset in blocks based on the equation no (3.1).

$$N = \frac{M}{n \times BS} \quad \text{_____(3.1).}$$

Where:

N: Number of partitions.

M: Size of datasets.

n: Number of nodes.

BS: Default block size in Hadoop distributed file system (64MB) of dataset send to each data-node, which can changed for special purpose to 128MB or 256MB … etc.

3. From the equation no (3.1), we will get the number of partitions that we need to partitioning our dataset, so when passing first partition to Hadoop system. The Hadoop system, in turn will pass it to the HDFS, the HDFS partition it again to the number of datanodes in the Hadoop system and each datanode will has only one block.

   The size for each partition will be known from the equation no (3.2).

$$PS = \frac{M}{N} \quad \text{_____(3.2).}$$

Where:

PS: Partition size.

M: complete dataset size.

N: Number of partitions.

4. The Hadoop system receives a block of dataset (Partition), and then this block fits directly to its nodes because the size of the partition is depending on how many nodes Hadoop has.
5. Each partition is sent to the Hadoop HDFS as on file and the Hadoop split it to parts. Each part is divided to blocks. Each block will be 64MB or less as the default size of HDFS block size.
6. Each input division is assigned to a map task (performed by the map worker) that calls the map function to handle this partition, and then the Traditional Apriori algorithm is applied.
7. The map task is design to process the partitions one by one, this will be through works on these partitions as files. One block processed by one mapper at a time. In mapper, the developer can determine his/her own trade area according to the requirements. In this manner, Map runs on all the nodes of the cluster and process the data blocks (for the target partition) in parallel.
8. The result of a Mapper also known as medium or intermediate output written on the local disk. Specific output is not stored on HDFS because they are temporary data and if they written to HDFS will generate many unnecessary copies.

9. The output of the mapper is shuffled (mixed) to minimize the node (which is a regular slave node but the lower phase will work here is called a reduced node). Shuffled is a physical copying movement of data, which done over the network.
10. Once all mappers have finished and output their shuffled in a reduced nodes, this medium output is merged and categorized. Then they are provided as an inputs to the reduce phase.
11. The second phase of processing is Reduce, where the user can specify his/her business area according to requirements. Input to the reducer of all map designers. The reduced output is the final output, which written on HDFS.
12. The reduce task (executed by reduce worker) is started directly after all maps from first partition finished giving a pre-result without waiting for other partitions maps to be finished. When the maps from first partition complete their cycle, the second maps cycle for the next partition start directly, applying the traditional Apriori algorithm on the second maps cycle. The output will be a list of intermediate key/value pairs, adding the results from the first maps cycle to the second. And so on, until reading the last partition maps. The last cycle must have the same results or more for applying traditional Apriori algorithm overall dataset.
13. When MapReduce function run, each node will processed on one block and send the result to the reducer and the reducer will collect the results from the mappers to give us the result for this partition alone without waiting for other partitions. This result is a pre-result for our complete dataset.
14. The system will pass the second partition after the map cycle complete and pass its results to the reducer. The reducer here will add the new results to the previous results.
15. The system will continue for N cycles until passing all the N partitions.

16. The results must be the same results or at least contain the same results if we run traditional Apriori algorithm overall dataset directly.

The problem of mining association rule is to find only interesting rule while running all uninteresting rules. Support and confidence are the two interestingness criteria used to measure the strength of association rules, but there are another measure can be used and it is more powerful which called a *lift*.

To understand how the proposed approach works, following points are discussed:

1. How the Hadoop HDFS data flow work.
2. The purpose from the proposal.
3. The situation that our proposal will work on it.

For inspect these points it assumed the following assumption.

Suppose we have *n* nodes and each node have 64MB Block size, and we have dataset with size *M*, and we run a MapReduce procedure on this dataset so we need to copy the whole dataset to the HDFS (which will divide it to chunks "Blocks" in 64MB size for each).

Here the HDFS system will have two scenarios:

1. First scenario is when the n (number of nodes) is bigger or equal to the number of chunks (Blocks), in this case our proposal not needed. Figure 3.1 shows the first scenario data flow.

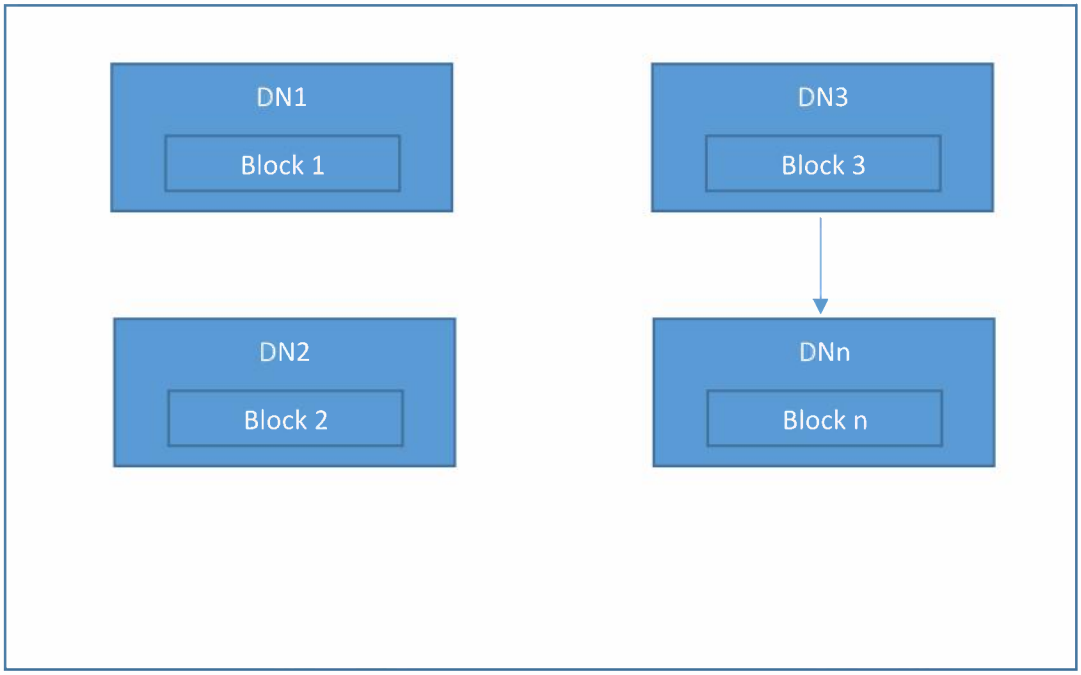


Figure 3.1. The first scenario data flow.

Where:

- DN: DataNode

2. Second scenario is when the *n* (number of nodes) is smaller than the number of chunks (Blocks), in this case the Hadoop system will pass the divided chunks to the nodes until Hadoop system pass chunks to all nodes. Running the MapReduce procedure and the remaining chunks will wait until any node finished the MapReduce procedure. And then, the HDFS will pass one of the reaming chunks to the free node.

   This will cause latency because the Hadoop MapReduce function will not completed and give results until all the chunks be process. Figure 3.2 shows the second scenario data flow.

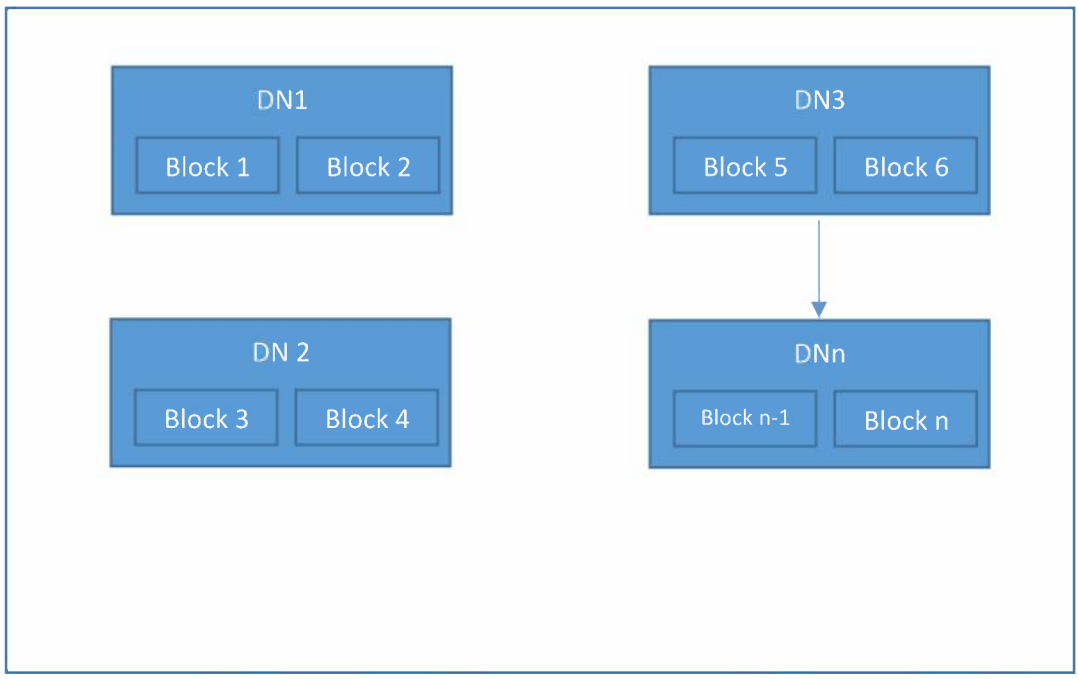


Figure 3.2. The second scenario data flow.

In the second scenario, the proposed approach will be the solution, especially in analytic systems that depends on time and the dataset is large to fit to the system simultaneously. Suppose we implement the proposal in the second scenario, the prototype system will divide the dataset into partitions to fit into the Hadoop HDFS system and after the Hadoop MapReduce completes the first cycle, which applied on the first partition and give us the results (Pre-result) the implementation will pass the next partition, which results in avoiding the latency. Figure 3.3 shows the proposed model data flow.

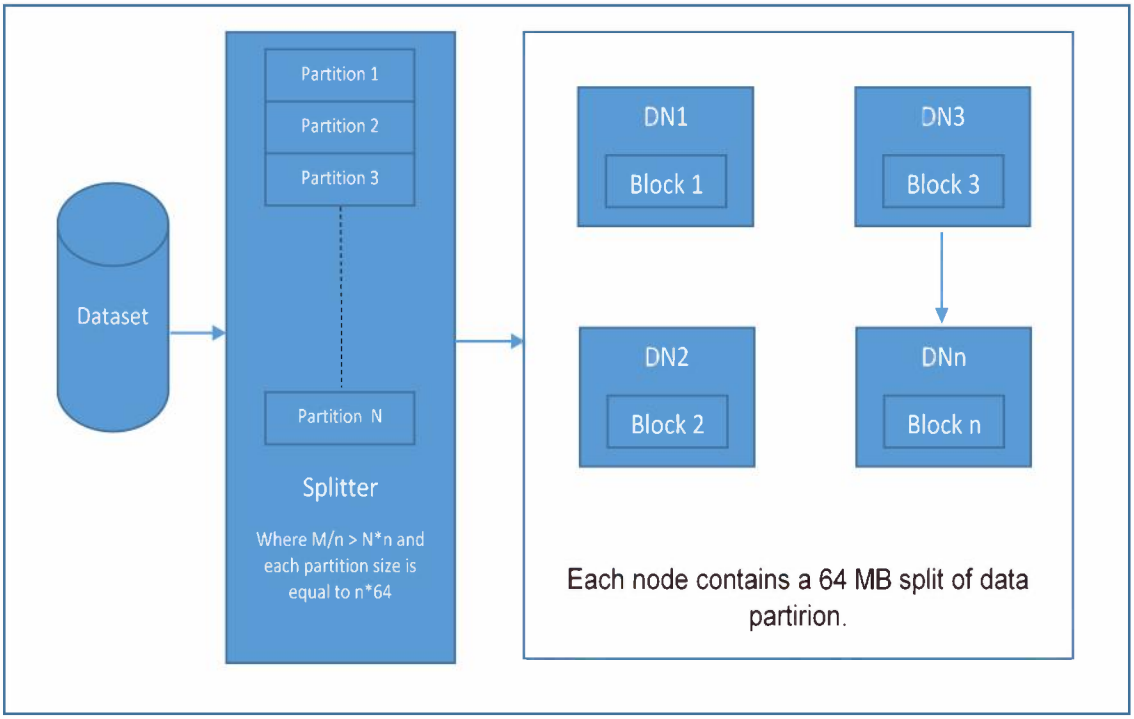


Figure 3.3. The proposed model data flow.

3.7. Summary

In this chapter, the proposed Apriori algorithm approach based on the MapReduce model on the Hadoop system presented. First, produce the Apriori algorithm two models combination used in our proposal. Second, present the theoretical base behind the proposal by explaining how the proposal system work. Finally, assumed two different scenarios for handling datasets in Hadoop HDFS to represent when the proposal would be efficient.

Chapter Four

Implementation

Chapter Four

Implementation

4.1. Introduction

In this chapter, the implementation of the proposed method presented. Described all steps of the proposed MapReduce-Apriori using the codes and diagrams. Using the MapReduce model to solve the problem of processing a large-scale dataset. First, presented the steps of building a Traditional Apriori algorithm and explaining all the steps through this program. Second, described the split procedure, and how it works. Third, described the steps of converting the MapReduce-Apriori program code to be run as the Hadoop MapReduce function and how it works. Finally, presented the merging and comparing the prototype using diagrams. The proposed method implemented with the following technologies:

- Python language as a core-programming tool.
- Cloudera CDH.
- Anaconda and Jupyter Notebook with Python Language as core programming tool to merge and compare for analytic and evaluate the results.

4.2. The system specifications.

1. Hardware specifications:

CPU:	Intel Core i5-4570
Ram:	32 GB
HDD:	500 GB

2. Software specifications:

Operating system:	Windows 10 Professional edition
Virtual operating system application :	Oracle VM VirtualBox ver. 6.0
Operating system on virtual machine :	CentOS-7-x86_64-DVD-1810
Hadoop Eco System:	Cloudera CDH 5.13

4.3. Dataset structure

Our dataset container is a 'csv'; file with the following specifications

File Name: Sales Grocery dataset.

Format: CSV.

File size: 565 MB.

Documents (Rows): more than 32 million, containing 3.2 million unique orders and about 50 thousand unique items.

Fields: four fields for each row (order_id,product_id,add_to_cart_order,

Reordered).

Our work will focus on the first two fields (order_id,product_id), both of these fields needed and important for applying Apriori algorithm and the other fields not important to the algorithm .

4.4. Hadoop HDFS file format

A storage format is just a way to define how information is stored in a file. This is usually indicated by the extension of the file (informally at least).

When dealing with Hadoop's file system not only do you have all of these traditional storage formats available to you (as if you can store PNG and JPG images on HDFS if

you like), but you also have some Hadoop-focused file formats to use for structured and unstructured data.

Some common storage formats for Hadoop include:

1. Text/CSV Files.
2. JSON Records.
3. Avro Files.
4. Sequence Files.
5. RC Files.
6. ORC Files.
7. Parquet Files.

Text and CSV files are quite common and frequently Hadoop developers and data scientists received text and CSV files to work upon. However, CSV files do not support block compression, thus compressing a CSV file in Hadoop often comes at a significant read performance cost. CSV files also easy to export and import from any database.

Choosing an appropriate file format can have some significant benefits:

1. Faster read times.
2. Faster write times.
3. Splittable files (so you do not need to read the whole file, just a part of it).
4. Schema evolution support (allowing you to change the fields in a dataset).
5. Advanced compression support (compress the files with a compression codec without sacrificing these features).

Some file formats designed for general use (like MapReduce or Spark); others designed for more specific use cases (like powering a database).

From the previous clarification, which specifies the type of data files that can used in the Hadoop HDFS system; to deal with the Hadoop HDFS system, the MongoDB dataset must be converted to csv file format.

4.5. Traditional Apriori algorithm implementation

Work with Apriori algorithm in a large dataset needs some filters and conditions to write an efficient code.

First, when start writing the code using lists and dictionaries which considered more than good for a small Dataset , the program run efficiently for a small training set, but the system crushes when we try to test the program with a large dataset. The reason for system crushes is that, the process of large dataset with that type for data analysis of search for frequent items in large dataset contain more than 32 million rows (record) with about 50 thousand unique items which need a lot of repetitive loops and a huge amount of memory.

Therefore, start writing the code for Apriori algorithm using what called "Python Generators" in python programming language. Generator functions allow developers to declare a function that behaves like an iterator, i.e. it can be used for loops. As shown in Appendix A.

4.5.1. Python Generator

The Python generator is a function, which returns a *iterate* generator (just an object we can replicate) by calling the yield. The yield, may be called a value, in which case this value is treated as a "generated" value. The next time you call Next () on a iterate generator (that is, in the next step in the for loop, for example), the generator resumes execution from where it is called the yield, not from the beginning of the job. Each case,

such as local variable values, retrieved and the generator continues to execute itself until the next call is called.

This is a great property for generators because it means that we do not have to store all values in memory once. Generator can load and process one value at a time, when finished and going to process the next value. This feature makes generators ideal for creating and calculating the recurrence of item pairs.

4.5.2. Traditional Apriori algorithm Design

Apriori is an algorithm used to identify frequent item sets (in our case, item pairs). It does so using a "bottom up" approach. First, identifying individual items that satisfy a minimum occurrence threshold. It then extends the item set, adding one item at a time and checking if the resulting item set still satisfies the specified threshold. The algorithm stops when there are no more items to add that meet the minimum occurrence requirement.

4.5.3. Association Rules Mining

Once the item sets have been generated using Apriori, mining association rules can be started. In the proposal, it is satisfied with looking at itemsets of size 2, the association rules will generate of the form $\{A\}$ -> $\{B\}$. One common application of these rules is in the domain of recommender systems, where customers who purchased item A are recommended item B.

The reason we will look for 2-itemsets frequency is:

1. It requires many dataset scans.
2. It is very slow.

3. In particular, 2-itemset will be enough to evaluate our proposal method, because the proposal focus on sampling and partitioning using distributed system.

There are three key metrics to consider when evaluating association rules:

1. Support

   This is the percentage of orders that contains the item set. The minimum support threshold required by Apriori can be set based on knowledge of your domain. In this example for dataset grocery, since there could be thousands of distinct items and an order can contain only a small fraction of these items, setting the support threshold to 0.01% is reasonable.

2. Confidence

   Given two items, A and B, the confidence measures is the percentage of times that item B is purchased and that item A was purchased. This is expressed by the equation no (4.1).

$$confidence\{A->B\} = support\{A,B\} / support\{A\} \quad \text{(4.1)}$$

Confidence values range from 0 to 1, where 0 indicates that B is never purchased when A is purchased, and 1 indicates that B is always purchased whenever A is purchased. Note that the confidence measure is directional. This means that we can also compute the percentage of times that item A is purchased, given that item B was purchased This is expressed by the equation no (4.2).

$$confidence\ \{B->A\} = support\{A,B\} / support\{B\}\ \} \quad \text{(4.2).}$$

3. Lift

Given two items, A and B, lift indicates whether there is a relationship between A and B, or whether the two items are occurring together in the same orders simply by chance. Unlike the confidence metric whose value may vary depending on direction, lift measure has no direction. This means that the lift{A,B} is always equal to the lift{B,A}, based on the equation no (4.3).

$$lift\{A,B\} = lift\{B,A\} = support\{A,B\} / (support\{A\} * support\{B\}) \} \quad \text{(4.3)}.$$

Therefore, the lift measuring chosen to locate and determine the frequent items, which considered more reliable and reduce the calculating and comparing processes.

- The prototype system is divided in three main parts:
  - Part A: Data Preparation, which include:
    1. Load order data.
    2. Convert order data into format expected by the association rules function.
    3. Display summary statistics for order data.
  - Part B: Association Rules Function, which include:
    1. Helper functions to the main association rules function.
    2. Association rules function.
  - Part C: Association Rules Mining

The proposed system uses Apriori algorithm in Hadoop and compares the results with Traditional Apriori algorithm.

First, Traditional Apriori algorithm prototype in large dataset is implemented and the results is saved in that file for comparing with the proposed-implemented system.

The Traditional Apriori algorithm prototype system consists of the following

components:

- The Libraries, which included in the prototype which are:

pandas, numpy, sys, itertools,collections ,IPython.display,time and random.

- The prototype includes the following functions:

1. A Function that load the orders 'csv' file to DataFrame and convert the DataFrame (Two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes rows and columns) to Series (One-dimensional *ndarray* with axis labels including time series) then returns the size of an object in MB.
2. A Function Returns the frequency counts for items and item pairs.
3. A Function Returns the number of unique orders.
4. A Function Returns a generator that yields item pairs, one at a time.
5. A Function Returns the frequency and support associated with item.
6. A Function Returns the name associated with item.
7. A Function Association rules, which include the following producers:
    a) Calculate item frequency and support.
    b) Filter for items below minimum support.
    c) Filter for orders with less than two items.
    d) Recalculate item frequency and support.
    e) Get item pairs generator.
    f) Calculate item pair frequency and support.
    g) Filter order for item pairs those below minimum support.
    h) Create table of association rules and compute relevant metrics.
    i) Return association rules sorted by lift in descending order.
8. A Function Replaces item ID with item name and displays association rules.

The above steps are shown in Figure 4.1.

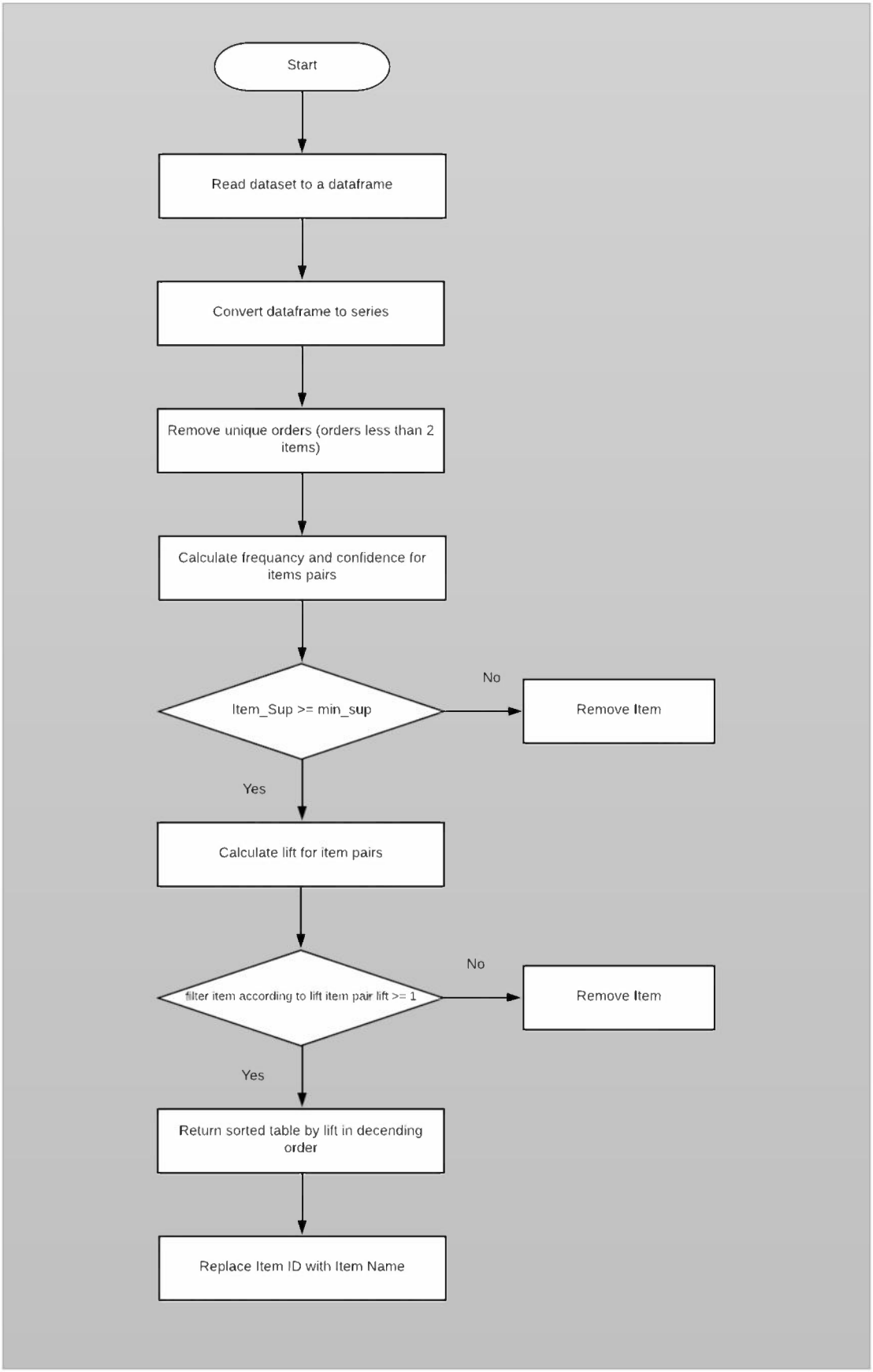


Figure 4.1 Traditional Apriori model diagram.

4.6. The implementation of proposed PMRA

This section contains two main tasks:

- First task is splitting the dataset to several partitions.
- Second task is running MapReduce Apriori algorithm for each partition separately from other partitions.

4.6.1. Dataset splitter

The idea here is very simple, the dataset (the orders file) in 'csv' format with a standard separated ',', this file (the orders file) contains more than 32 million line stored in a file with size 565 MB, and the prototype system uses a Hadoop system with a single node for testing purpose. For explanation, if we used 64 MB block size in HDFS that means we will divide the file size by 64MB, which will be:

$$Split\ size = 565/64 = 8.82$$

It means, that the dataset file is split to nine files at least.

In addition, we have 32 million line, so we need to divide the dataset to 10 files at least. Which means, each file contains 3.3 million line. As shown in Appendix D.

The structure of the splitter prototype is shown in figure 4.2 and explained in the following steps:

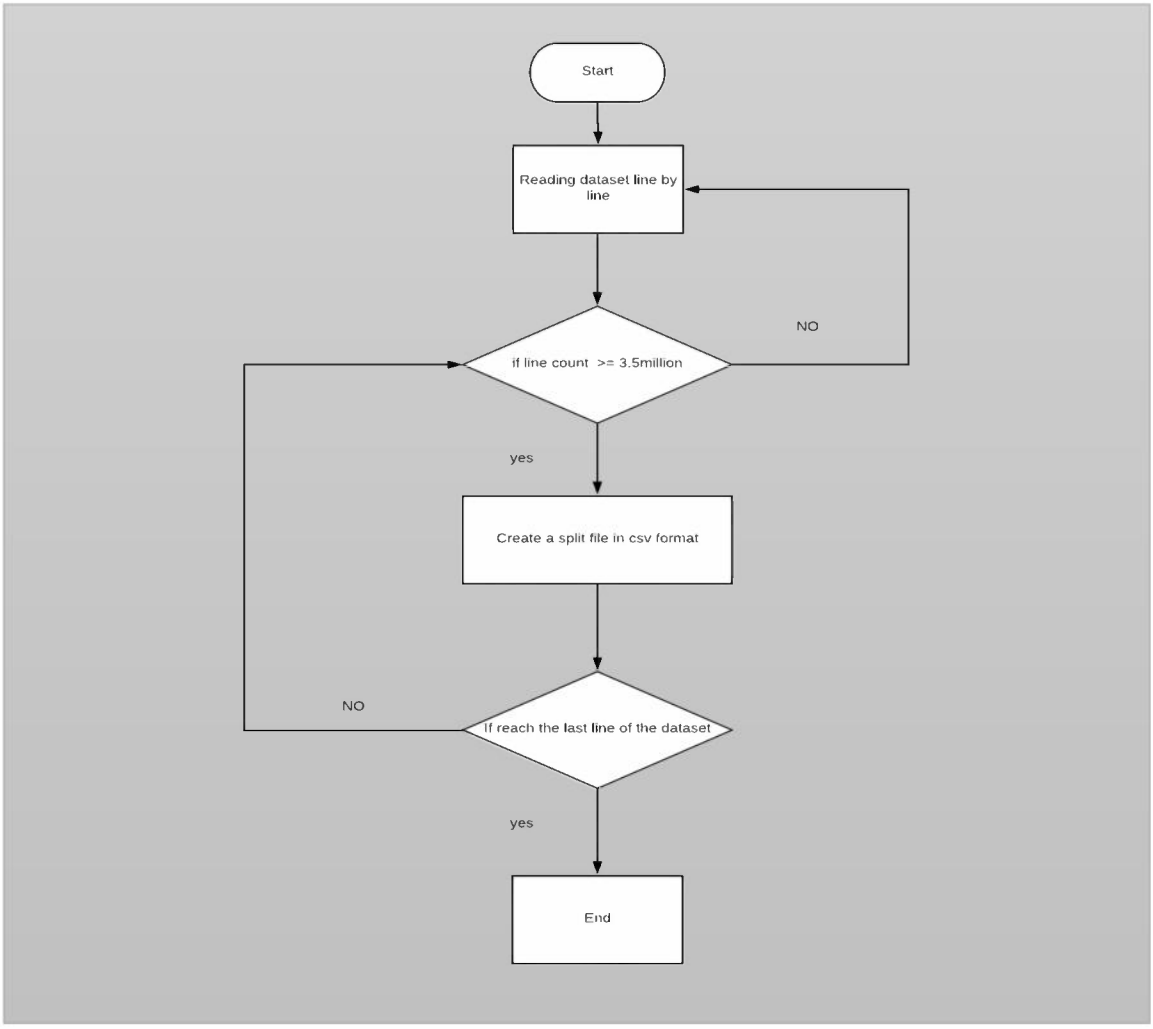


Figure 4.2: The splitter diagram.

1. Reading the dataset file using standard input.
2. Count the lines entered by standard input until reach the first 3.3 million line, then create the first file using the same file name for the dataset and add number 1 at the end of the file name.
3. Start count from the line comes next the line we entered in the file before and create the second file using the same file name for the dataset and add number 2 at the end of the file name.
4. Continue with the same procedure until dataset file reaches the last line. At the end, 10 csv files present the result of the splitter.

4.6.2. Partitioned MapReduce Apriori algorithm prototype PMRA.

In Hadoop MapReduce, there are some changes comparing with the Traditional Apriori algorithm prototype, which illustrated in section (4.5).

The obvious difference is needed to be consider is the way the data is feed to the prototype system.

- The MapReduce architecture needs that map and reduce jobs are written as programs that read from standard input and write to standard output.
- Hadoop reads input files and streams them to standard output.
- For each line in standard input, Map function is called.
- Map is responsible for interpretation the input and formatting the output.
- Maps writes lines to standard output.

There are several techniques to write MapReduce program depending on the purpose of it, like data cleansing, data filtering, data summarization, data joining, text processing and computing.

Since this thesis, focuses on data partitioning before pass it to the HDFS. Therefore, data-cleansing technique in a simple format is used. As shown in Appendix B.

The MapReduce Apriori algorithm partitions prototype system is shown in Figure 4.3 and it includes the following functions:

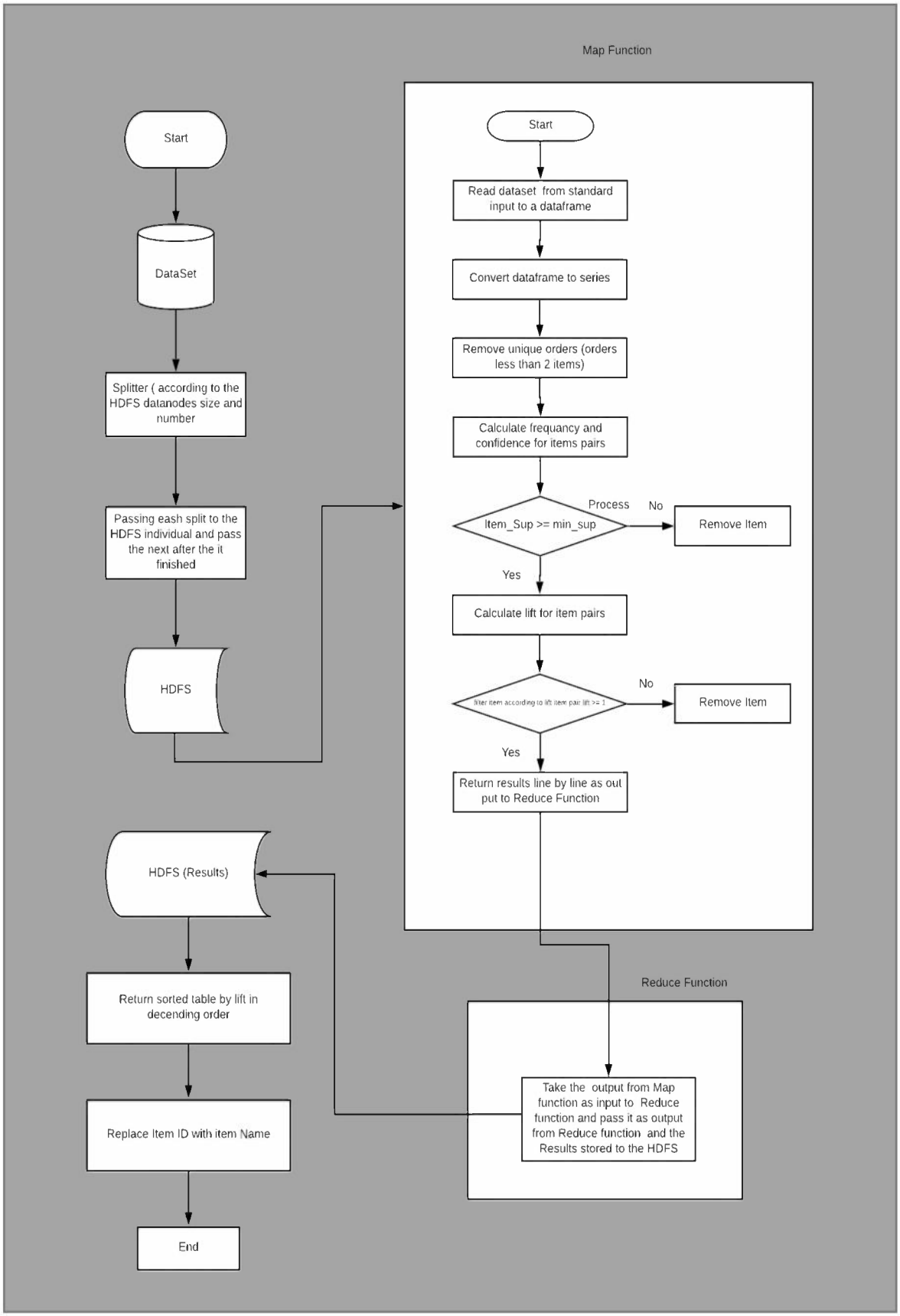


Figure 4.3: The proposal PMRA diagram.

1. Split the dataset according to the HDFS capacity.
2. Pass first split (partition) to the HDFS.
3. The HDFS takes care of the passed partition and split it again to the datanodes. In our case and for experimental purpose is used a single node. Therefore, the HDFS pass it directly without splitting and the partition will fit directly to the datanode because it is split before passing to HDFS according to HDFS capacity in step 1.
4. Hadoop streaming function runs the MapReduce function.
5. The MapReduce function executes the map function first, which includes the same procedures like the Traditional Apriori algorithm with some modifications.
6. The main modification is the way feeding the data to the map function and how the map function outputs the results. Both of input and output must use standard input/output functions.
7. The reducer function here in cleansing technique takes the output from the map function as an input and directly pass it as output to HDFS.
8. Taking the results from HDFS and returns association rules sorted by lift in descending order.
9. Replaces item ID with item name and displays association rules.

4.7. The merge and comparison process.

Applying both prototypes represented before (in sections 4.5 and 4.6) creates the result files, one result file for the whole dataset from experimental one in section (4.5) and ten results files from the experimental two in section (4.6).

The result files have the same headers, but with different values. The result file from applying the prototype in section (4.5) contains information about the whole dataset, but the results files from applying section (4.6) each one contains information about the partition it belongs to. Table 4.1 shows the results tables.

Table 4.1: The results tables.

itemA	itemB	freqAB	supportAB	freqA	supportA	freqB	supportB	confAtoB	confBtoA	lift
Organic Raspberry Yogurt	Organic Wildberry Yogurt	31	0.010102	109	0.035521	112	0.036498	0.284404	0.276786	7.792254
Organic Grapefruit Ginger Sparkling Yerba Mate	Cranberry Pomegranate Sparkling Yerba Mate	45	0.014664	177	0.05768	118	0.038454	0.254237	0.381356	6.611548
Yerba Mate Sparkling Classic Gold	Cranberry Pomegranate Sparkling Yerba Mate	36	0.011732	142	0.046275	118	0.038454	0.253521	0.305085	6.592924
Baby Food Pouch - Roasted Carrot Spinach & Beans	Baby Food Pouch - Spinach Pumpkin & Chickpea	37	0.012057	182	0.05931	97	0.03161	0.203297	0.381443	6.431386
Baby Food Pouch - Roasted Carrot Spinach & Beans	Baby Food Pouch - Butternut Squash, Carrot & Chickpea	51	0.01662	182	0.05931	147	0.047904	0.28022	0.346939	5.849617
Strawberry and Banana Fruit Puree	Peter Rabbit Organics Mango, Banana and Orange Puree	32	0.010428	236	0.076907	73	0.023789	0.135593	0.438356	5.699819
Lactose Free Blueberry Yogurt	Organic Lactose Free Strawberry Yogurt	35	0.011406	133	0.043342	155	0.050511	0.263158	0.225806	5.209915
Unsweetened Whole Milk Strawberry Yogurt	Unsweetened Whole Milk Blueberry Greek Yogurt	49	0.015968	194	0.06322	155	0.050511	0.252577	0.316129	5.000444
Organic Lactose Free Strawberry Yogurt	Lactose Free Blueberry Yogurt	33	0.010754	155	0.050511	133	0.043342	0.212903	0.24812	4.912206
Unsweetened Whole Milk Peach Greek Yogurt	Unsweetened Whole Milk Mixed Berry Greek Yogurt	47	0.015316	176	0.057354	168	0.054747	0.267045	0.279762	4.877776
Unsweetened Whole Milk Mixed Berry Greek Yogurt	Unsweetened Whole Milk Blueberry Greek Yogurt	41	0.013361	168	0.054747	155	0.050511	0.244048	0.264516	4.831576

The comparing process compares the values obtained from experimental one in section (4.5) with the results files obtained from experimental two in section (4.6) sequentially. After merging each result file with the result file from the previous cycle. The process of comparing the results file of the Traditional Apriori algorithm will continue with each merge file form the proposed MapReduce Apriori algorithm until the merge of the results files is completed.

For comparison purpose, some operations to speed up the process are used:

1. Comparison is conducted between itemA, itemB and lift.
2. All lift values less than '1' is eliminate. As shown in Appendix C section A.

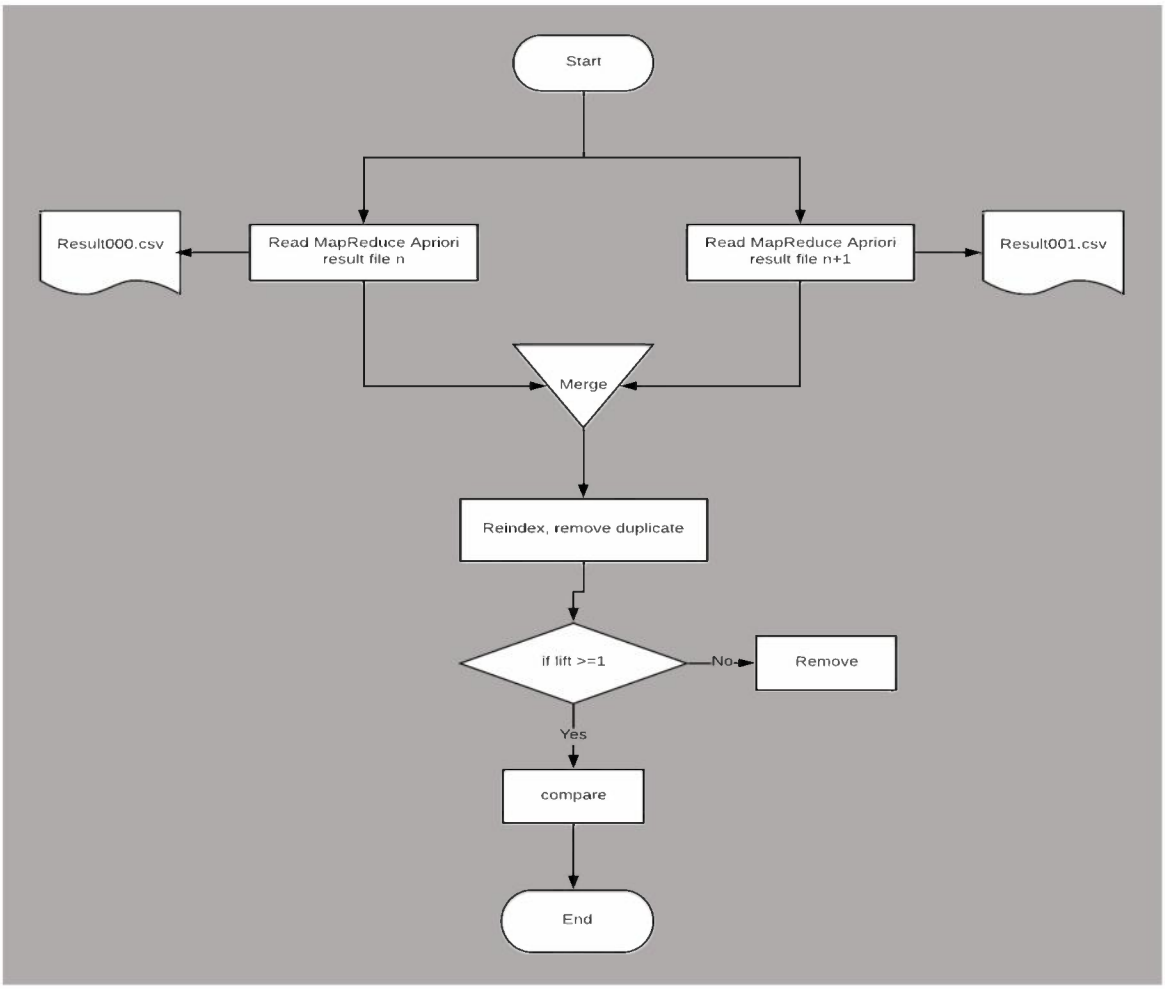


Figure 4.4: The merge diagram

In addition, in merge file some operations are performed as shown in Appendix C section B:

1. After merging the result files in one DataFrame, it is noted that the new merged DataFrame has duplicated indexes.
2. Re-index the sorted merged DataFrame to solve the duplicated indexes issue.
3. Sorting the merge DataFrame by higher lift.
4. Search for duplicated rows depending on the itemA and itemB columns, keep the first row of frequent items, and drop the other (second row) because after sorting the first row of frequent items would have the higher lift. These operations are shown in Figure 4.4.

4.8. Summary

This chapter explained the implementation of the proposed model. First, the Traditional Apriori algorithm on two frequent itemsets is implemented. Second, the Association Rules Mining and the three key metrics to consider when evaluating association rules are presented. Third, the proposed dataset splitter and the MapReduce Apriori algorithm for each part separated from other parts are implemented. Finally, the merge and comparing process implemented.

Chapter Five

Experiments, Results and Discussion

Chapter Five

Experiments, Results and Discussion

5.1. Introduction

This chapter, explores the dataset structure, presents, explains and analyzes the experimental results. It starts by explaining the experiments and results of the traditional model and the proposed model. Then, it demonstrates the measurement of performance. Finally, it discusses and compares the experiments results together through time execution to generate strong association rules using the lift factor in the association rules mining using Apriori algorithm.

5.2. Experiments Infrastructure.

To test the proposal performance, there are two main prototypes. First one, for traditional Apriori algorithm experiment and the second one for Partitioned MapReduce Apriori algorithm (PMRA).

5.3. Traditional Apriori algorithm Experiment

Applying the dataset to the Traditional Apriori algorithm on 2-items frequent and save the results to 'csv' file. The result file contains the following details:

1. Traditional Apriori algorithm prototype generates 12 fields (Transition Number, itemA, itemB, freqAB, supportAB,freqA, supportA, freqB, supportB, confidenceAtoB, confidenceBtoA, lift), these fields contain calculated data extracted from the given dataset.
2. The result file contains around (48752) rows.
3. These rows contain all the frequent two items.
4. These rows sorted in descending order according to lift column.

5. The most important rows in this experiment are the rows with 'lift' field value equal or greater than '1', which were the first 208 rows.

5.4. PMRA experiment

In this experiment, applying the dataset on Hadoop eco system. As it mentioned earlier in section (4.6.2.), for experimental purpose proposed method is applied on Hadoop with a single node cluster.

This single node can receive 64MB (the default size of the HDFS block in the datanode) of dataset and passes it to the MapReduce function implemented in section (4.6).

5.4.1. Splitting the dataset

As explained in section (4.3), the dataset size is 565MB. So when splitting the dataset to a 64MB or fewer, the results file from the splitter was 10 files.

These files had the same name of the original dataset added to the end of the splits partitions name a number 000, 001 until the last one 009 (order_products-000.csv, order_products-001.csv, … order_products-009.csv).

5.4.2. Applying the MapReduce Apriori algorithm on the splits partitions

After splitting the dataset to fit in the HDFS system, in this experiment there are 10 files to pass to the HDFS respectively.

The process is illustrated in the following steps

1. Copy the first split file to the HDFS.
2. Run the implemented MapReduce Apriori algorithm function.
3. The MapReduce creates result file.
4. This result file is compared with the result file generated from the traditional result file on the original dataset. Table (5.1) summarize this operation.

Table 5.1: Operations summarization

File	RRows	MRows	FreqI	ExcTS	MExcT	CMRISR	CSRISR	PRN	PoD
Result_000	50172	50172	227	93.0028	0.0000	83.17	76.21	3300000	10.17

Where:

- RRows : Rows number of result file.
- MRows : Rows number after merging.
- FreqI : Freq. items in result file after filtering the lift.
- ExcTS : Execution time for each result file separately in seconds.
- MExcT : Merged execution time in seconds.
- CMRISR : Compatibility between the main result file and identical rows in sub result file.
- CSRISR : Compatibility between the sub result file and identical rows in sub result file.
- PRN. : Partition rows number.PoD: Percentage of partitions after merge to the original dataset.
- File: Refers to the result file name.
    - Result_000 to Result_009: The result files from the ***PMRA*** experiments.
    - Result : The result file form the Traditional Apriori algorithm.

These abbreviations are used in the next tables.

The previous steps are the first cycle to our experiment. The following steps illustrate the second cycle in the experiment.

1. Copy the second split file to the HDFS.
2. Run the implemented Apriori MapReduce function.
3. The MapReduce create result file.
4. This result file from the second cycle will merged with the result file from the previous cycle (first cycle).
5. After merging the results files and sort them according the descending order of the 'lift' column.
6. This merged result file compared with the result file generated from the traditional result file on the original dataset.

Table (5.2) summarize the two cycle operations.

Table 5.2: Two cycle operations

File	RRows.	MRows	FreqI	ExcTS	MExcT	CMRISR	CSRISR	PRN	PoD
Result_001	50397	100569	295	93.1667	186.1695	94.71	66.78	3300000	20.35

Table 5.3: Total cycles operations results summarization table.

File	RRows.	MRows	FreqI	ExcTS	MExcT	CMRISR	CSRISR	PRN	PoD
Result_000	50172	50172	227	93.0028	0.0000	83.17	76.21	3300000	10.17
Result_001	50397	100569	295	93.1667	186.1695	94.71	66.78	3300000	20.35
Result_002	50446	151015	355	95.8360	282.0055	99.52	58.31	3300000	30.52
Result_003	50196	201211	380	95.5155	377.5210	99.52	54.47	3300000	40.70
Result_004	50724	251935	410	96.5416	474.0626	100.00	50.73	3300000	50.87
Result_005	50467	302402	441	95.3093	569.3718	100.00	47.17	3300000	61.05
Result_006	50593	352995	459	95.7507	665.1225	100.00	45.32	3300000	71.22
Result_007	50297	403292	480	96.0524	761.1749	100.00	43.33	3300000	81.39
Result_008	50330	453622	493	94.8186	855.9935	100.00	42.19	3300000	91.57
Result_009	49715	503337	504	90.8185	946.8120	100.00	41.27	2734489	100.00
Result	48752		208	583.7603				32434489	

Applying the same procedures to all the dataset splits partitions will generate results can be summarized in table (5.3).

5.5. Discussion

The result of the proposed model experiment was compared with the result of Traditional Apriori algorithm experiment. The results and information, which demonstrated in table (5.3), illustrate and discuss through execution time, compatibility between frequent items in result files after each merge to evaluate our proposal efficiency.

5.5.1. Execution Time

One of the most important factors consider in the experiment is the execution time, and here explore it from several points.

Table 5.4: Execution separated and merged time for result files

File	ExcTS	MExcT
Result_000	93.0028	0.0000
Result_001	93.1667	186.1695
Result_002	95.8360	282.0055
Result_003	95.5155	377.5210
Result_004	96.5416	474.0626
Result_005	95.3093	569.3718
Result_006	95.7507	665.1225
Result_007	96.0524	761.1749
Result_008	94.8186	855.9935
Result_009	90.8185	946.8120
Result	583.7603	

As shown in table (5.4), the first column in the table shows the result file name, the second one shows the execution time separately for each split partition passed to the Hadoop system in a single node cluster. The third one shows the merged execution time for merged result files in a single node cluster and the last row shows the execution time for the result file for the Traditional Apriori algorithm.

From this table it concluded that the highest execution time in separated results file is in the fifth file (Result_004 = 96.5416 seconds) and the lowest execution time in separated results file is in the last file (Result_009 = 90.8185 seconds). Both of them is less than the execution time for the result file from the Traditional Apriori algorithm (Result = 583.7603 seconds).

On the other side, the merged execution time for merged result files is (946.8120 seconds) which bigger than the execution time for the result file for the Traditional Apriori algorithm (583.7603 seconds).

The different between the execution time in both experiments is that the first experiment proceed on a real hardware system and the second experiment on virtual system and also run on Hadoop system containing single node and Hadoop needs to run HDFS and Yarn services which means more load on the hardware system.

Figure (5.1) shows the difference between the ExcTS and the MExcT columns, the execution time of the results file in a single node is increased after every merging until it exceed the execution time of the Traditional Apriori algorithm after the fifth result file, and that mean after 50% of the dataset.

Figure 5.1: Difference between the ExcTS and the MExcT columns

5.5.2. Testing compatibility between the separated result file and the Traditional Apriori.

In table (5.5), the first column shows the result file name, the second one shows the separated execution time in seconds and the third column shows the compatibility between the separated result file and the Traditional Apriori result file which means how many frequent items in the separated files after merging appears in the result file for the Traditional Apriori.

It is easy to noting that with each merged result files the compatibility is increased. It is almost beginning with high score in the first result file (Result_000 = 83.17%), and drive up in increasing manner with every merging process until it reaches the full compatibility in the fifth result file (Result_004 = 100%). Figure (5.2) shows the execution time for separated results file and the compatibility with Traditional Apriori.

Table 5.5: Compatibility between the separated results and the Traditional Apriori

File	ExcTS	CMRISR
Result_000	93.0028	83.17
Result_001	93.1667	94.71
Result_002	95.8360	99.52
Result_003	95.5155	99.52
Result_004	96.5416	100.00
Result_005	95.3093	100.00
Result_006	95.7507	100.00
Result_007	96.0524	100.00
Result_008	94.8186	100.00
Result_009	90.8185	100.00

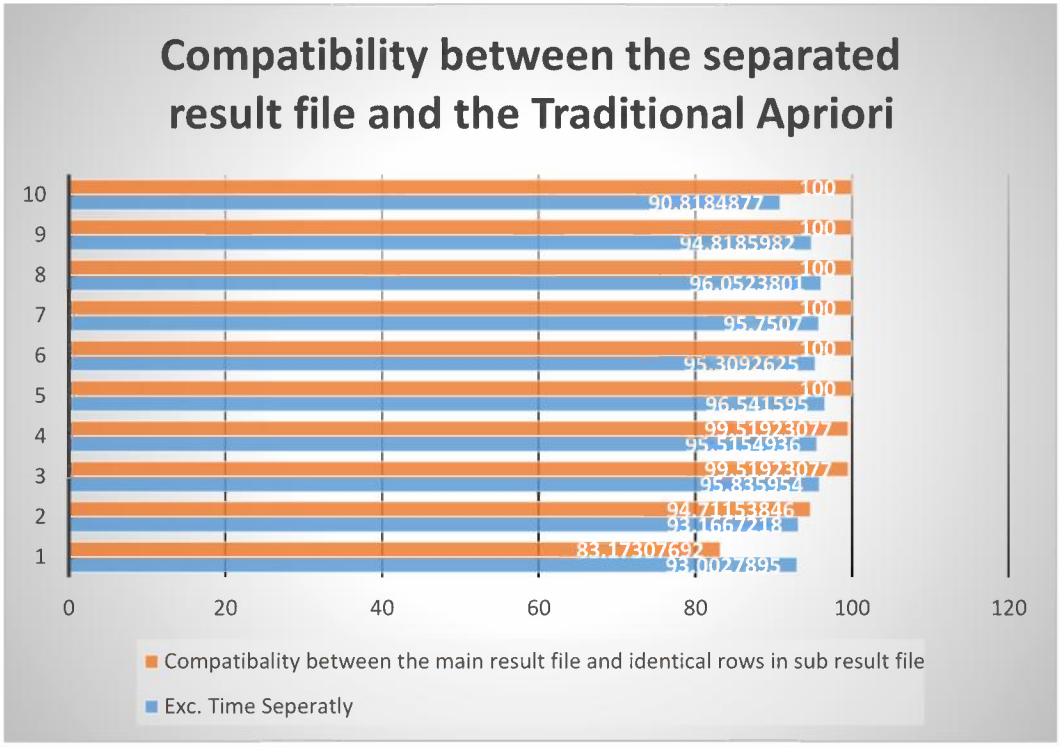


Figure 5.2: Execution time for separated results file and the compatibility with Traditional Apriori

5.5.3. Traditional Apriori result VS Proposed PMRA Results.

Also with a high compatibly between the merged results file there is incompatibility appears in separated results file.

As shown in table (5.6), the Traditional Apriori algorithm gives us a 208 frequent for two itemset. In the first separated result file there are 227 frequent for two itemset and there

is about 83.17 % of Traditional Apriori algorithm founded in first separated result file (173 the identical rows). That is mean, there is another frequent items appears in the separated result file from section 5.4, that are not belong to the Traditional Apriori result file from section 5.3.

Table 5.6: Frequent items, compatibly and incompatibility.

File	FreqI	CMRISR	CSRISR
Result_000	227	83.17	76.21
Result_001	295	94.71	66.78
Result_002	355	99.52	58.31
Result_003	380	99.52	54.47
Result_004	410	100.00	50.73
Result_005	441	100.00	47.17
Result_006	459	100.00	45.32
Result_007	480	100.00	43.33
Result_008	493	100.00	42.19
Result_009	504	100.00	41.27
Result	208		

These incompatible rows (incompatible rows mean nonidentical frequent items) in this separated result file about 54 rows so the compatibility between the separated result file and identical rows (identical rows mean identical frequent items) in separated result file here is about 76.21%.

The second separated result file and after merged the result with the first separated result file has more frequent items which about 295 rows, 94.71% from the Traditional Apriori algorithm frequent items were found in the merged these two separated result files. But also there are more frequent items in this merged are not in the Traditional Apriori algorithm which about 98 rows. Therefore, the compatibility between the merged separated result files and identical rows in separated result file here is about 66.78 %.

With each merged the identical rows (identical frequent items) is increased with the rows in the result file from the Traditional Apriori algorithm but the incompatibility between results files also increased, that is with each merging process, there are more incompatibility frequent items appear in the merged result files.

Figure (5.3) shows that the relation between compatibility and incompatibility, and from it and from the table (5.6) it concluded that whenever the compatibility is increased the incompatible is increased too.

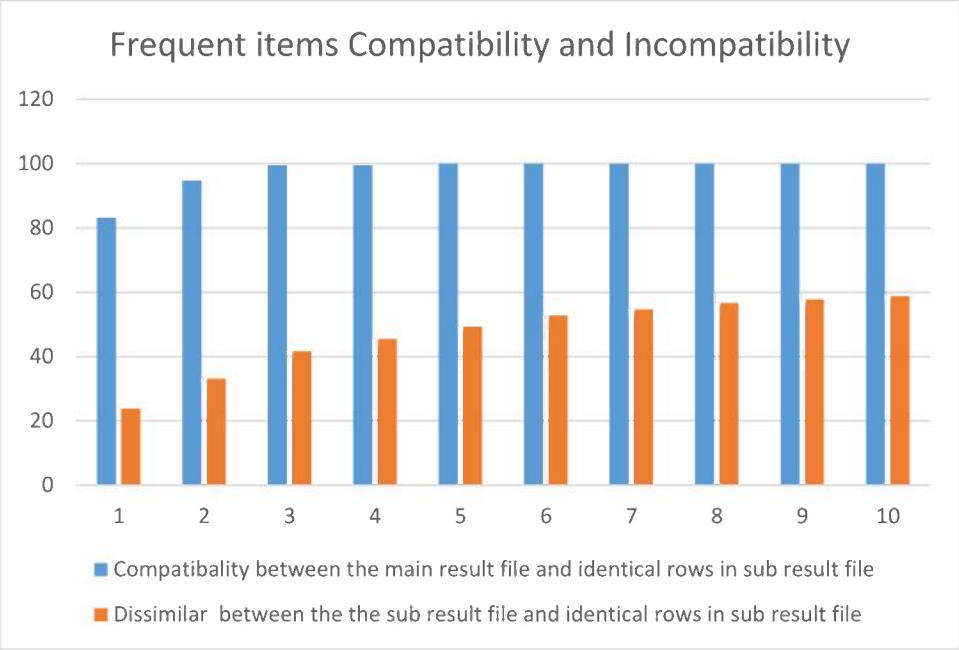


Figure 5.3: Frequent items Compatibility and Incompatibility

5.5.4. Enhance execution time.

In experiment in section (5.4), applying the implementation on a single node cluster; both table (5.7) and figure (5.4) shows the execution time for each partition separately.

Table (5.7): The execution time for each partition separately.

File	ExcTS
Result_000	93.0028
Result_001	93.1667
Result_002	95.8360
Result_003	95.5155
Result_004	96.5416
Result_005	95.3093
Result_006	95.7507
Result_007	96.0524
Result_008	94.8186
Result_009	90.8185

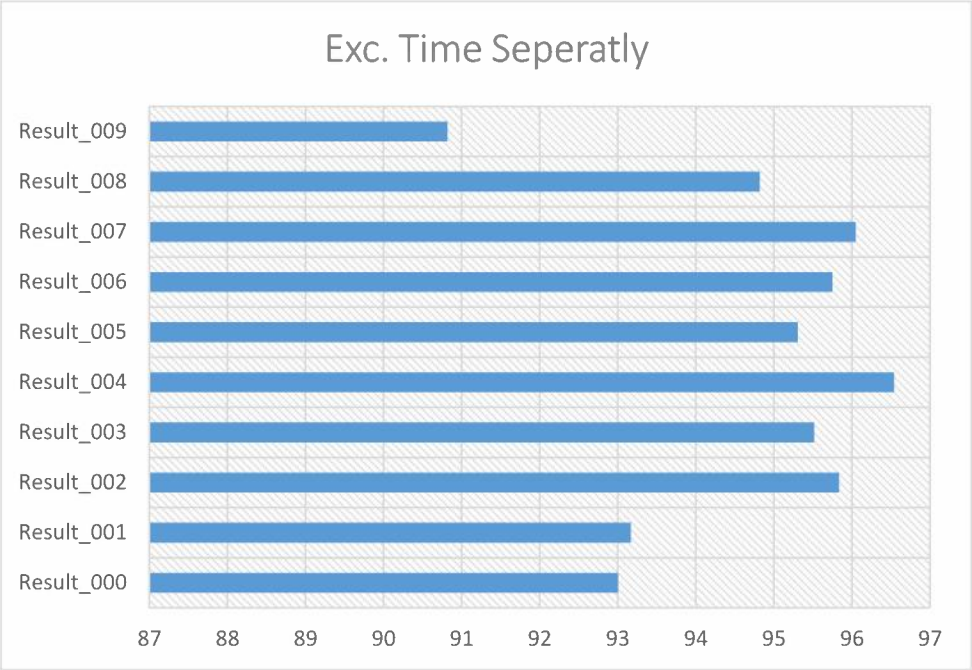


Figure (5.4): The execution time for each partition separately.

Now suppose that the system has two-nodes, which means that the partitions will be five partitions and first partition contains both datasets from previous experiment in section (5.4.1).

The first and the second partitions contain 6600000 rows and when pass this file to HDFS it will be split the file in to two blocks, each block belongs to one of the two datanodes.

Each node run the MapReduce function on its own dataset block at the same time, if one node finished the Map function it sends the results to the reduce function , but the reduce function will wait until the other datanode finished before writing the results to the HDFS again.

From that, it concluded that the execution time would be the highest execution time and that is 93.1667 seconds.

As the previous assumption, we can generate execution time for a two-node system as shown in table (5.8).

Table (5.8): Two-node execution time

File	ExcTS	Merged Exc. Time
Result_000	93.166	0
Result_001	95.83	188.996
Result_002	96.5416	285.5376
Result_003	96.0528	381.5904
Result_004	94.8186	476.409
Result	583.7603	

In addition, to reach the 50% of data (which means reach the full identical rows with the Traditional Apriori), we must apply the proposal prototype in section (5.4.2) at least until the third result file.

The third result file execution time according to table (5.8) is 285.5376 seconds, and it is about 48.91% comparing to Traditional Apriori execution time. Figure (5.5) shows two-node assumption execution time.

Figure (5.5): Two-node assumption execution time

5.5.5. Rules comparison between Traditional Apriori and proposed PMRA

Based on the results from various statistical and data mining techniques the findings that behavioral variables are better predictors of profitable customers were confirmed using the Apriori Algorithm we want to find the association rules that have minimum support (0.01) and minimum lift=1 in our large dataset.

Best rules found in applying Traditional Apriori algorithm was 208 rule and this is the top four rules according to descending sort measure lift:

1. Organic Strawberry Chia Lowfat 2% Cottage Cheese=YES 1163 ==> Organic Cottage Cheese Blueberry Acai Chia=YES 839 conf:(0.26 < lift:(9.44)>
2. Grain Free Turkey Formula Cat Food=YES 1809 ==> Grain Free Turkey Formula Cat Food=YES 879 conf:(0.175) < lift:(6.02)>
3. Organic Fruit Yogurt Smoothie Mixed Berry=YES 1518 ==> Apple Blueberry Fruit Yogurt Smoothie=YES 1249 conf:(0.22) < lift:(5.54 >
4. Nonfat Strawberry With Fruit On The Bottom Greek Yogurt=YES 1666 ==> 0% Greek, Blueberry on the Bottom Yogurt=YES 1391 conf:(0.24) < lift:(5.31)>

Best rules found in applying PMRA on 50% of the dataset was 441 rule and this is the top four rules according to descending sort measure lift:

1. Organic Strawberry Chia Lowfat 2% Cottage Cheese=YES 595 ==> Organic Cottage Cheese Blueberry Acai Chia=YES 432 conf:(0.28) < lift:(9.7)>
2. Grain Free Turkey Formula Cat Food=YES 945 ==> Grain Free Turkey Formula Cat Food=YES 444 conf:(0.175) < lift:(5.95)>
3. Organic Grapefruit Ginger Sparkling Yerba Mate=YES 1518 ==> Cranberry Pomegranate Sparkling Yerba Mate=YES 844 conf:(0.2 1) < lift:(5. 77)>
4. Organic Fruit Yogurt Smoothie Mixed Berry=YES 755 ==> Apple Blueberry Fruit Yogurt Smoothie =YES 649 conf:(0.2 3) < lift:(5.44)>

Comparing the two models, we found that all the 208 rules from Traditional Apriori algorithm were found in applying 50% of dataset on PMRA model on single node cluster.

5.6. Summary

This chapter presented and analyzed the experimental results. First, it presented the dataset structure, the experiments infrastructure, the Traditional Apriori algorithm experiment and the PMRA experiment. Second, it discussed the results from different perspective. Finally, it explained the improvement in the execution time and speedup for experiments in single node and two nodes cluster assumption.

Chapter Six

Conclusion and Future Works

Chapter Six

Conclusion and Future Works

This chapter concludes the thesis, clarifing its advantages and drawbacks , also it suggests some future works to improve the proposal implementation.

6.1 Conclusion

The past decade has been characterized by the continued growth of the technological industry. The results are seen as a significant increase in data generation. A phenomenon commonly referred to as large data or big data revolution. The widespread common interest in the ability to analyze this massive amount of data today is a key issue that has led to an increase in technologies geared towards, thereby local data as its main advantage.

Big data is a vast research area, and parallel-distributed is often one of the most effective techniques for solving problems and discovering hidden information patterns. Among these new technologies for parallel-distributed programming and computing, the Hadoop/MapReduce which widly accepted framework.

Processing big data in parallel-distributed systems bring many opportunities, but with these opportunities come many challenges. Challenges including but not limited to latency, security (security restrictions to process data on the cloud systems like Amazon Web Services (AWS) or alternative solutions), privacy and local system capacity (local system capacity can not handle processing data in reasonable time).

The PMRA proposed showing high promises solution to found the best rules in large datasets using parallel and distributed system.

This study, suggests promised alternative solution to these challenges by dividing data into parts that can fit into the local system. Using simple words, this study can be described as a partitioning before partitioning.

The basic idea behind this study, is that the size of the data is greater than the size of the system storage capacity to process in timely manner. The study suggests a new method depends on the HDFS system capacity.

Through this study and based on the conducted experiments and their results, it concluded that the implementation of the proposed approach to address 50% of the data, these results can help to make fair decisions. On other hand, better results could be obtained if the proposed approach is implemented in real Hadoop system, so we can overcome the delay results from adoption of virtual environment.

6.2 Future work

To compensate the drawbacks in this study, the auther suggest applying the same implementation using another factor instead of lift. Using confidence measure instead of lift as a factor to solve the percentage of false data.

In addition, this study focused on HDFS, and using of MapReduce function in a simple architecture, it suggested more focus on MapReduce function design.

There is another technology in Hadoop eco system called spark, applying the proposal through it would improve the performance. Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.

References

[1] A. Bifet, "Mining Big Data in Real Time," *Informatica (Slovenia),* vol. 37, no. 1, pp. 15-20, 2013.

[2] D. Reinsel, J. Gantz, and J. Rydning, "Data age 2025: the digitization of the world from edge to core," *Seagate https://www. seagate. com/files/www-content/our-story/trends/files/idc-seagate-dataage-whitepaper. pdf,* 2018.

[3] J. Fan, F. Han, and H. Liu, "Challenges of big data analysis," *National science review,* vol. 1, no. 2, pp. 293-314, 2014.

[4] V. Parmar and I. Gupta, "Big data analytics vs Data Mining analytics," *International Journal in IT & Engineering,* vol. 3, no. 3, pp. 258-263, 2015.

[5] D. Laney, "3D data management: Controlling data volume, velocity and variety," *META group research note,* vol. 6, no. 70, p. 1, 2001.

[6] A. De Mauro, M. Greco, M. Grimaldi, G. Giannakopoulos, D. P. Sakas, and D. Kyriaki-Manessi, "What is big data? A consensual definition and a review of key research topics," in *AIP conference proceedings*, 2015, vol. 1644, no. 1, pp. 97-104: AIP.

[7] A. McAfee, E. Brynjolfsson, and T. H. Davenport, "Big data: the management revolution," *Harvard business review,* vol. 90, no. 10, pp. 60-68, 2012.

[8] F. SARTORE, "Big Data: Privacy and Intellectual Property in a Comparative Perspective.," *Trento Law and Technology Research Group,* 2016.

[9] M. Hilbert, "Big data for development: A review of promises and challenges," *Development Policy Review,* vol. 34, no. 1, pp. 135-174, 2016.

[10] M. Schroeck, R. Shockley, J. Smart, D. Romero-Morales, and P. Tufano, "Analytics: the real-world use of big data: How innovative enterprises extract value from uncertain data, Executive Report," *IBM Institute for Business Value and Said Business School at the University of Oxford,* 2012.

[11] S. Finlay, *Predictive analytics, data mining and big data: Myths, misconceptions and methods*. Springer, 2014.

[12] H. A. Edelstein, *Introduction to data mining and knowledge discovery*. Two Crows, 1998.

[13] A. Ansari and A. Parab, "Apriori-A Big Data Analysis in Education," *IJISET - International Journal of Innovative Science, Engineering & Technology,* vol. 1, no. 10, 2014.

[14] O. Yahya, O. Hegazy, and E. Ezat, "An Efficient Implementation of A-Priori algorithm based on Hadoop-MapReduce model," *International journal of Reviews in Computing,* vol. 12, 2012.

[15] H. Map and R. Tutorial, "ApacheTM," *Retrieved November,* vol. 12, p. 2012, 2010.

[16] A. M. Middleton, "Data-intensive technologies for cloud computing," in *Handbook of cloud computing*: Springer, 2010, pp. 83-136.

[17] M. T. Özsu and P. Valduriez, *Principles of distributed database systems*. Springer Science & Business Media, 2011.

[18] D. Mango, "Top 5 considerations when evaluating NoSQL Databases," *White Paper*.

[19] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *ACM sigmod record*, 1993, vol. 22, no. 2, pp. 207-216: ACM.

[20] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, 1994, vol. 1215, pp. 487-499.

[21] J. S. Park, M.-S. Chen, and P. S. Yu, *An effective hash-based algorithm for mining association rules* (no. 2). ACM, 1995.

[22] R. E. Thevar and R. Krishnamoorthy, "A new approach of modified transaction reduction algorithm for mining frequent itemset," in *Computer and Information Technology, 2008. ICCIT 2008. 11th International Conference on*, 2008, pp. 1-6: IEEE.

[23] M. J. Zaki, "Parallel and distributed association mining: A survey," *IEEE concurrency,* vol. 7, no. 4, pp. 14-25, 1999.

[24] H. Toivonen, "Sampling large databases for association rules," in *VLDB*, 1996, vol. 96, pp. 134-145.

[25] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," in *ACM SIGMOD Record*, 1997, vol. 26, no. 2, pp. 255-264: ACM.

[26] M. J. Zaki and K. Gouda, "Fast vertical mining using diffsets," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, 2003, pp. 326-335: ACM.

[27] N. Li, L. Zeng, Q. He, and Z. Shi, "Parallel implementation of apriori algorithm based on mapreduce," in *Software Engineering, Artificial Intelligence, Networking and Parallel & Distributed Computing (SNPD), 2012 13th ACIS International Conference on*, 2012, pp. 236-241: IEEE.

[28] H. Yu, J. Wen, H. Wang, and L. Jun, "An improved Apriori algorithm based on the Boolean matrix and Hadoop," *Procedia Engineering,* vol. 15, pp. 1827-1831, 2011.

[29] A. Ezhilvathani and K. Raja, "Implementation of parallel apriori algorithm on hadoop cluster," *International Journal of Computer Science and Mobile Computing,* vol. 2, no. 4, pp. 513-516, 2013.

[30] Z. Qureshi and S. Bansal, "Improving Apriori algorithm to get better performance with cloud computing," *Int. J. Softw. Hardware Res. Eng,* vol. 2, no. 2, 2014.

[31] S. Singh, R. Garg, and P. Mishra, "Review of apriori based algorithms on mapreduce framework," *arXiv preprint arXiv:1702.06284,* 2017.

[32] F. Kovacs and J. Illés, "Frequent itemset mining on hadoop," in *(ICCC), 2013 IEEE 9th International Conference on Computational Cybernetics*, 2013, pp. 241-245: IEEE.

Appendix A

Traditional Apriori algorithm implementation code

This code in python language represents the Traditional Apriori algorithm implementation for two frequent itemset. The code was written in python programming language and can be executed on any operating system. The code calculates the two frequent itemset, the support, the confidence, the lift and exports all the results to the screen and to a CSV file. In addition, it calculates the execution time for the whole procedure.

```
# Python code for Traditional Apriori algorithm implementation
import pandas as pd
import numpy as np
import sys
from itertools import combinations, groupby
from collections import Counter
from IPython.display import display
import time
import random
#Time before the operations start
then = time.time()
# Function that returns the size of an object in MB
def size(obj):
    return "{0:.2f} MB".format(sys.getsizeof(obj) / (1000 * 1000))
```

```
orders = pd.read_csv('order_products__train.csv')
print('orders -- dimensions: {0};   size: {1}'.format(orders.shape, size(orders)))
display(orders.head())
```

 # Convert from DataFrame to a Series, with order_id as index and item_id as value

```
orders = orders.set_index('order_id')['product_id'].rename('item_id')
display(orders.head(10))
type(orders)
```

```
print('dimensions: {0};   size: {1};   unique_orders: {2};   unique_items: {3}'
     .format(orders.shape, size(orders), len(orders.index.unique()),
len(orders.value_counts())))
```

 # Returns frequency counts for items and item pairs

```
def freq(iterable):
   if type(iterable) == pd.core.series.Series:
      return iterable.value_counts().rename("freq")
   else:
```

```
    return pd.Series(Counter(iterable)).rename("freq")


  # Returns number of unique orders
def order_count(orders_item):
   return len(set(orders_item.index))
# Returns generator that yields item pairs, one at a time
def get_items_pairs(orders_item):
   orders_item = orders_item.reset_index().values
   for order_id, order_object in groupby(orders_item, lambda x: x[0]):
      item_list = [item[1] for item in order_object]


      for item_pair in combinations(item_list, 2):
         yield item_pair


# Returns frequency and support associated with item
def merge_items_stats(items_pairs, items_stats):
   return (items_pairs
            .merge(items_stats.rename(columns={'freq': 'freqA', 'support': 'suptA'}),
left_on='item_A', right_index=True)
```

```
            .merge(items_stats.rename(columns={'freq': 'freqB', 'support': 'suptB'}),
left_on='item_B', right_index=True))


# Returns name associated with item
def merge_item_name(rules, item_name):
    columns = ['itemA','itemB','freqAB','suptAB','freqA','suptA','freqB','suptB',
            'confAtoB',"confBtoA",'lift']
    rules = (rules
            .merge(item_name.rename(columns={'item_name': 'itemA'}),
left_on='item_A', right_on='item_id')
            .merge(item_name.rename(columns={'item_name': 'itemB'}),
left_on='item_B', right_on='item_id'))
    return rules[columns]


def association_rules(orders_item, min_sup):


    print("Starting orders_item: {:22d}".format(len(orders_item)))
    # Calculate item frequency and support
    items_stats            = freq(orders_item).to_frame("freq")
    items_stats['support']  = items_stats['freq'] / order_count(orders_item) * 100
```

```
# Filter from orders_item items below min support

qualifying_items       = items_stats[items_stats['support'] >= min_sup].index

orders_item            = orders_item[orders_item.isin(qualifying_items)]


print("Items with support >= {}: {:15d}".format(min_sup, len(qualifying_items)))

print("Remaining orders_item: {:21d}".format(len(orders_item)))


# Filter from orders_item orders with less than 2 items

order_size             = freq(orders_item.index)

qualifying_orders      = order_size[order_size >= 2].index

orders_item            = orders_item[orders_item.index.isin(qualifying_orders)]

print("Remaining orders with 2+ items: {:11d}".format(len(qualifying_orders)))

print("Remaining orders_item: {:21d}".format(len(orders_item)))


# Recalculate item frequency and support

items_stats            = freq(orders_item).to_frame("freq")

items_stats['support']  = items_stats['freq'] / order_count(orders_item) * 100

# Get item pairs generator

item_pair_gen          = get_items_pairs(orders_item)
```

```
    # Calculate item pair frequency and support
    items_pairs             = freq(item_pair_gen).to_frame("freqAB")
    items_pairs['suptAB'] = items_pairs['freqAB'] / len(qualifying_orders) * 100
    print("Item pairs: {:31d}".format(len(items_pairs)))


    # Filter from items_pairs those below min support
    items_pairs             = items_pairs[items_pairs['suptAB'] >= min_sup]
    print("Item pairs with support >= {}: {:10d}\n".format(min_sup, len(items_pairs)))


    # Create table of association rules and compute relevant metrics
    items_pairs = items_pairs.reset_index().rename(columns={'level_0': 'item_A',
'level_1': 'item_B'})
    items_pairs = merge_items_stats(items_pairs, items_stats)
    items_pairs['confAtoB'] = items_pairs['suptAB'] / items_pairs['suptA']
    items_pairs["confBtoA"] = items_pairs['suptAB'] / items_pairs['suptB']
    items_pairs['lift']         = items_pairs['suptAB'] / (items_pairs['suptA'] *
items_pairs['suptB'])
```

 # Return association rules sorted by lift in descending order

```
    return items_pairs.sort_values('lift', ascending=False)
```

 # Calling the association_rules function and pass the orders variable and the minimum support threshold 0.01

```
rules = association_rules(orders, 0.01)
```

 # Replace item ID with item name and display association rules

```
item_name   = pd.read_csv('products.csv')

item_name   = item_name.rename(columns={'product_id':'item_id',
'product_name':'item_name'})

rules_final = merge_item_name(rules, item_name).sort_values('lift', ascending=False)

display(rules_final)
```

 # Time after it finished

```
now = time.time()
```

 # Calculate the execution time and print the results to the screen

```
print("It took: ", now-then, " seconds")
```

 # Print the Results to CSV file

```
rules_final.to_csv('resulttest.csv', sep=';')
```

Appendix B

PMRA implementation code

This code in python language represents the PMRA implementation for two frequent itemset. The code was written to be executed Hadoop environment as a mapper function. The code calculates the two frequent itemset, the support, the confidence, the lift and export all the results to the HDFS in CSV file. Unlike the previous Traditional Apriori algorithm implementation code in Appendix A, the coding has some rules must be obtained when written to be run in Hadoop MapReduce:

1. There is no printing on the screen while the program is run; all the results will be written in HDFS when the whole procedure is completed.
2. The input and the output must be in standard input/output format.
3. Replacing item ID with item name must be done outside the MapReduce function because it needs to call external file.

 # Python code for Traditional Apriori algorithm implementation

```
! /usr/bin/env python

# For Python 2.7

import pandas as pd

import numpy as np

import sys

from itertools import combinations, groupby

from collections import Counter
```

```
from IPython.display import display


# Function that returns the size of an object in MB
def size(obj):
    return "{0:.2f} MB".format(sys.getsizeof(obj) / (1000 * 1000))
orders = pd.read_csv('sys.stdin')


# Convert from DataFrame to a Series, with order_id as index and item_id as value
orders = orders.set_index('order_id')['product_id'].rename('item_id')
# Returns frequency counts for items and item pairs
def freq(iterable):
    if type(iterable) == pd.core.series.Series:
        return iterable.value_counts().rename("freq")
    else:
        return pd.Series(Counter(iterable)).rename("freq")


# Returns number of unique orders
def order_count(order_item):
    return len(set(order_item.index))
```

```
# Returns generator that yields item pairs, one at a time
def get_item_pairs(order_item):
    order_item = order_item.reset_index().values
    for order_id, order_object in groupby(order_item, lambda x: x[0]):
        item_list = [item[1] for item in order_object]
        for item_pair in combinations(item_list, 2):
            yield item_pair


# Returns frequency and support associated with item
def merge_item_stats(item_pairs, item_stats):
    return (item_pairs
                .merge(item_stats.rename(columns={'freq': 'freqA', 'support': 'supportA'}),
left_on='item_A', right_index=True)
                .merge(item_stats.rename(columns={'freq': 'freqB', 'support': 'supportB'}),
left_on='item_B', right_index=True))


# Returns name associated with item
def merge_item_name(rules, item_name):
```

```
    columns = ['itemA','itemB','freqAB','supportAB','freqA','supportA','freqB','supportB',
            'confidenceAtoB','confidenceBtoA','lift']
    rules = (rules
            .merge(item_name.rename(columns={'item_name': 'itemA'}),
left_on='item_A', right_on='item_id')
            .merge(item_name.rename(columns={'item_name': 'itemB'}),
left_on='item_B', right_on='item_id'))
    return rules[columns]



def association_rules(order_item, min_support):
    # Calculate item frequency and support
    item_stats             = freq(order_item).to_frame("freq")
    item_stats['support']  = item_stats['freq'] / order_count(order_item) * 100
    # Filter from order_item items below min support
    qualifying_items       = item_stats[item_stats['support'] >= min_support].index
    order_item             = order_item[order_item.isin(qualifying_items)]
```

```
# Filter from order_item orders with less than 2 items
order_size          = freq(order_item.index)
qualifying_orders   = order_size[order_size >= 2].index
order_item          = order_item[order_item.index.isin(qualifying_orders)]

# Recalculate item frequency and support
item_stats          = freq(order_item).to_frame("freq")
item_stats['support']  = item_stats['freq'] / order_count(order_item) * 100

# Get item pairs generator
item_pair_gen       = get_item_pairs(order_item)

# Calculate item pair frequency and support
item_pairs          = freq(item_pair_gen).to_frame("freqAB")
item_pairs['supportAB'] = item_pairs['freqAB'] / len(qualifying_orders) * 100

# Filter from item_pairs those below min support
item_pairs          = item_pairs[item_pairs['supportAB'] >= min_support]
```

```
    # Create table of association rules and compute relevant metrics
    item_pairs = item_pairs.reset_index().rename(columns={'level_0': 'item_A', 'level_1':
'item_B'})
    item_pairs = merge_item_stats(item_pairs, item_stats)

    item_pairs['confidenceAtoB'] = item_pairs['supportAB'] / item_pairs['supportA']
    item_pairs['confidenceBtoA'] = item_pairs['supportAB'] / item_pairs['supportB']
    item_pairs['lift']          = item_pairs['supportAB'] / (item_pairs['supportA'] *
item_pairs['supportB'])

    # Return association rules sorted by lift in descending order
    return item_pairs.sort_values('lift', ascending=False)

rules = association_rules(orders, 0.01)
display(rules)
```

#Script to execute PMRA Code

```
echo -e "Starting Example for test01 \n"

echo -e "Removing old output directories \n"

hadoop fs -rm -r test01-streaming

hadoop fs -rm -r test01

echo -e "\nCreating input directory and copying input data\n"

hadoop fs -mkdir test01

hadoop fs -copyFromLocal order_products__train.csv test01

echo -e "\nRunning Map Reduce\n"
```

Streaming command

```
hadoop jar /usr/lib/hadoop-0.20-mapreduce/contrib/streaming/hadoop-streaming-2.6.0-mr1-cdh5.13.0.jar \

    -input dataset \

    -output result-streaming \

    -mapper mapper.py \

    -combiner org.apache.hadoop.mapred.lib.IdentityReducer \

    -reducer org.apache.hadoop.mapred.lib.IdentityReducer \

    -jobconf stream.num.map.output.key.fields=2 \

    -jobconf stream.num.reduce.output.key.fields=2 \

    -jobconf mapred.reduce.tasks=1 \
```

```
    -file mapper.py


echo -e "\nMapReduce completed. Printing output\n"

hadoop fs -cat result-streaming/*

echo -e "\nExample completed."
```

Appendix C

The merge and comparison operation

The merge and comparison code was implemented using python programming language in Anaconda Jupyter Notebook environment, which allow us to execute the code systematically to view the results of this certain part of code.

A. The compare operation code.

1. Declare and import the python language libraries which necessary to the comparison operation. Reading the result file from the Traditional Apriori algorithm implementation to DataFrame variable and print to the screen the first five lines in the file.

```
In [2]: %matplotlib inline
        import matplotlib.pyplot as plt
        import pandas as pd

        data = pd.read_csv('result.csv', sep=';', index_col=None)
        #data=data.head(100)
        data.head(5)
```

Out[2]:

	T	itemA	itemB	freqAB	supportAB	freqA	supportA	freqB	supportB	confidenceAtoB	confidenceBtoA	lift
0	0	Organic Strawberry Chia Lowfat 2% Cottage Cheese	Organic Cottage Cheese Blueberry Acai Chia	306	0.010155	1163	0.038595	839	0.027843	0.263113	0.364720	9.449868
1	1	Grain Free Chicken Formula Cat Food	Grain Free Turkey Formula Cat Food	318	0.010553	1809	0.060033	879	0.029170	0.175788	0.361775	6.026229
2	3	Organic Fruit Yogurt Smoothie Mixed Berry	Apple Blueberry Fruit Yogurt Smoothie	349	0.011582	1518	0.050376	1249	0.041449	0.229908	0.279424	5.546732
3	9	Nonfat Strawberry With Fruit On The Bottom Gre...	0% Greek, Blueberry on the Bottom Yogurt	409	0.013573	1666	0.055288	1391	0.046162	0.245498	0.294033	5.318230
4	10	Organic Grapefruit Ginger Sparkling Yerba Mate	Cranberry Pomegranate Sparkling Yerba Mate	351	0.011648	1731	0.057445	1149	0.038131	0.202773	0.305483	5.317849

2. Count how many lines Traditional Apriori algorithm result file contain which mean the amount of the frequent item set it contain.

```
In [4]: len(data.index)
Out[4]: 48751
```

3. Reading the first result file from the PMRA implemented on first partition to DataFrame variable and print to the screen the first five lines in the file.

```
In [6]: result000 = pd.read_csv('result_000.csv', sep=';', index_col=None)
        #result000=result000.head(100)
        result000.head(5)
```

Out[6]:

	Unnamed: 0	itemA	itemB	freqAB	supportAB	freqA	supportA	freqB	supportB	confidenceAtoB	confidenceBtoA	lift
0	0	Organic Raspberry Yogurt	Organic Wildberry Yogurt	31	0.010102	109	0.035521	112	0.036498	0.284404	0.276786	7.792254
1	1	Organic Grapefruit Ginger Sparkling Yerba Mate	Cranberry Pomegranate Sparkling Yerba Mate	45	0.014664	177	0.057680	118	0.038454	0.254237	0.381356	6.611548
2	2	Yerba Mate Sparkling Classic Gold	Cranberry Pomegranate Sparkling Yerba Mate	36	0.011732	142	0.046275	118	0.038454	0.253521	0.305085	6.592924
3	4	Baby Food Pouch - Roasted Carrot Spinach & Beans	Baby Food Pouch - Spinach Pumpkin & Chickpea	37	0.012057	182	0.059310	97	0.031610	0.203297	0.381443	6.431386
4	5	Baby Food Pouch - Roasted Carrot Spinach & Beans	Baby Food Pouch - Butternut Squash, Carrot & C...	51	0.016620	182	0.059310	147	0.047904	0.280220	0.346939	5.849617

4. Count how many lines PMRA implemented on first partition result file contain which mean the amount of the frequent item set it contain.

```
In [7]: len(result000.index)
```

Out[7]: 50172

5. Remove the frequent items that have lift smaller than one (lift < 1) from the Traditional Apriori algorithm result file and print to the screen the first five lines in the file.

In data mining and association rule learning, lift is a measure of the performance of a targeting model

- lift = 1 implies no relationship between A and B. (ie: A and B occur together only by chance)
- lift > 1 implies that there is a positive relationship between A and B. (ie: A and B occur together more often than random)
- lift < 1 implies that there is a negative relationship between A and B. (ie: A and B occur together less often than random)

The value of lift is that it considers both the confidence of the rule and the overall data set.

```
In [8]: # Remove all the items that have lift less than 1 form the main result file
        data = data.drop(data[data.lift < 1].index)
        data.head(5)
```

Out[8]:

	T	itemA	itemB	freqAB	supportAB	freqA	supportA	freqB	supportB	confidenceAtoB	confidenceBtoA	lift
0	0	Organic Strawberry Chia Lowfat 2% Cottage Cheese	Organic Cottage Cheese Blueberry Acai Chia	306	0.010155	1163	0.038595	839	0.027843	0.263113	0.364720	9.449868
1	1	Grain Free Chicken Formula Cat Food	Grain Free Turkey Formula Cat Food	318	0.010553	1809	0.060033	879	0.029170	0.175788	0.361775	6.026229
2	3	Organic Fruit Yogurt Smoothie Mixed Berry	Apple Blueberry Fruit Yogurt Smoothie	349	0.011582	1518	0.050376	1249	0.041449	0.229908	0.279424	5.546732
3	9	Nonfat Strawberry With Fruit On The Bottom Gre...	0% Greek, Blueberry on the Bottom Yogurt	409	0.013573	1666	0.055288	1391	0.046162	0.245498	0.294033	5.318230
4	10	Organic Grapefruit Ginger Sparkling Yerba Mate	Cranberry Pomegranate Sparkling Yerba Mate	351	0.011648	1731	0.057445	1149	0.038131	0.202773	0.305483	5.317849

6. Remove the frequent items that have lift smaller than one (lift < 1) from the PMRA implemented on first partition result file and print to the screen the first five lines in the file.

```
In [9]: # Remove all the items that have lift less than 1 form the first sub result
result000 = result000.drop(result000[result000.lift < 1].index)
result000.head(5)
```

Out[9]:

	Unnamed: 0	itemA	itemB	freqAB	supportAB	freqA	supportA	freqB	supportB	confidenceAtoB	confidenceBtoA	lift
0	0	Organic Raspberry Yogurt	Organic Wildberry Yogurt	31	0.010102	109	0.035521	112	0.036498	0.284404	0.276786	7.792254
1	1	Organic Grapefruit Ginger Sparkling Yerba Mate	Cranberry Pomegranate Sparkling Yerba Mate	45	0.014664	177	0.057680	118	0.038454	0.254237	0.381356	6.611548
2	2	Yerba Mate Sparkling Classic Gold	Cranberry Pomegranate Sparkling Yerba Mate	36	0.011732	142	0.046275	118	0.038454	0.253521	0.305085	6.592924
3	4	Baby Food Pouch - Roasted Carrot Spinach & Beans	Baby Food Pouch - Spinach Pumpkin & Chickpea	37	0.012057	182	0.059310	97	0.031610	0.203297	0.381443	6.431386
4	5	Baby Food Pouch - Roasted Carrot Spinach & Beans	Baby Food Pouch - Butternut Squash, Carrot & C...	51	0.016620	182	0.059310	147	0.047904	0.280220	0.346939	5.849617

7. Compare the main result file and the first PMRA result file and find the compatibility between them.

```
In [10]: # Compare between the main result file and the first sub result file Find
# the combtiblity between the main result file and the first sub result file
dataLenth = len(data.index)
result000Lenth = len(result000.index)
c = 0
for i in range(0,dataLenth):
    for j in range (0, result000Lenth):
        if ((data.loc[i,'itemA'] == result000.loc[j,'itemA']) and (data.loc[i,'itemB'] == result000.loc[j,'itemB'])
            and (data.loc[i,'lift'] > 1 and result000.loc[j,'lift']) > 1):
            c = c + 1
            #print(data.loc[i,'itemA'],'--',data.loc[i,'itemB'],'\n')
            #print(result000.loc[j,'itemA'],'--',result000.loc[j,'itemB'],'\n')
print('The identical rows =  ',c)
```

```
The identical rows =  173
```

8. Calculate the average of compatibility between the Traditional Apriori algorithm result file and identical rows in PMRA implemented on first partition result file.

```
In [11]: # Calculate the average of compatibality between the the main result file and identical rows in sub result file
averageToReasult = (c/208)*100
averageToReasult
```

```
Out[11]: 83.17307692307693
```

9. Count the length of the PMRA implemented on first partition result file after removing the lift below 1, which means count the number of frequent items in this file.

```
In [14]: result000Lenth
Out[14]: 227
```

10. Calculate the average of incompatibility between the Traditional Apriori algorithm result file and identical rows in PMRA implemented on first partition result file.

```
In [12]: # Calculate the average of compatibality between the the sub result file and identical rows in sub result file
         averageToReasult000 = (c/227)*100
         averageToReasult000
Out[12]: 76.2114537444934
```

B. The merge and compare operation code.

The only different between this operation and the operation that we need to merge the result files from PMRA implementation one by one before comparing with the Traditional Apriori algorithm result file.

1. Repeat the first and second steps in section A and then reading the first result file from the PMRA implemented on second partition to DataFrame variable and print to the screen the first five lines in the file.

```
In [3]: #read second result file and load it to dataframe
        result001 = pd.read_csv('result_001.csv', sep=';', index_col=None)
        #result000=result000.head(100)
        result001.head(5)
```

Out[3]:

	Unnamed: 0	itemA	itemB	freqAB	supportAB	freqA	supportA	freqB	supportB	confidenceAtoB	confidenceBtoA	lift
0	0	Organic Strawberry Chia Lowfat 2% Cottage Cheese	Organic Cottage Cheese Blueberry Acai Chia	37	0.012063	128	0.041732	87	0.028365	0.289062	0.425287	10.190982
1	1	Yogurt, Sheep Milk, Strawberry	Yogurt, Sheep Milk, Blackberry	31	0.010107	110	0.035863	106	0.034559	0.281818	0.292453	8.154675
2	2	Nonfat Strawberry With Fruit On The Bottom Gre...	0% Greek, Blueberry on the Bottom Yogurt	45	0.014671	168	0.054773	130	0.042384	0.267857	0.346154	6.319801
3	3	Strawberry and Banana Fruit Puree	Peter Rabbit Organics Mango, Banana and Orange...	34	0.011085	229	0.074661	73	0.023800	0.148472	0.465753	6.238269
4	4	Organic Fruit Yogurt Smoothie Mixed Berry	Apple Blueberry Fruit Yogurt Smoothie	32	0.010433	127	0.041406	131	0.042710	0.251969	0.244275	5.899544

2. Count how many lines PMRA implemented on second partition result file contain which means the amount of the frequent item set it contain.

```
In [4]: #lenth of the second dataframe
        len(result001)
Out[4]: 50397
```

3. Merge the two result files from the PMRA implementation by merge the two DataFrames belong to them.

```
In [5]: #merge the first and socend result file
        mergeResult = pd.concat([result000, result001])
        mergeResult.head(5)
```

Out[5]:

	Unnamed: 0	itemA	itemB	freqAB	supportAB	freqA	supportA	freqB	supportB	confidenceAtoB	confidenceBtoA	lift
0	0	Organic Raspberry Yogurt	Organic Wildberry Yogurt	31	0.010102	109	0.035521	112	0.036498	0.284404	0.276786	7.792254
1	1	Organic Grapefruit Ginger Sparkling Yerba Mate	Cranberry Pomegranate Sparkling Yerba Mate	45	0.014664	177	0.057680	118	0.038454	0.254237	0.381356	6.611548
2	2	Yerba Mate Sparkling Classic Gold	Cranberry Pomegranate Sparkling Yerba Mate	36	0.011732	142	0.046275	118	0.038454	0.253521	0.305085	6.592924
3	4	Baby Food Pouch - Roasted Carrot Spinach & Beans	Baby Food Pouch - Spinach Pumpkin & Chickpea	37	0.012057	182	0.059310	97	0.031610	0.203297	0.381443	6.431386
4	5	Baby Food Pouch - Roasted Carrot Spinach & Beans	Baby Food Pouch - Butternut Squash, Carrot & C...	51	0.016620	182	0.059310	147	0.047904	0.280220	0.346939	5.849617

4. After merge, the result files to one DataFrame, we notes that the new merged DataFrame have duplicated indexes.

```
In [6]: # after merge the result files to one dateframe we notes that the
        # new merged dataframe have duplicatied indexes

In [7]: mergeResult.loc[0,'itemA']
Out[7]: 0                          Organic Raspberry Yogurt
        0    Organic Strawberry Chia Lowfat 2% Cottage Cheese
        Name: itemA, dtype: object
```

5. Sort the merged DataFrame by higher lift.

```
In [8]: # Sorting the merge dataframe by higher lift

In [9]: SortedMergeResult = mergeResult.sort_values(by=['lift'],ascending=False)
        SortedMergeResult.head(5)
```

Out[9]:

	Unnamed: 0	itemA	itemB	freqAB	supportAB	freqA	supportA	freqB	supportB	confidenceAtoB	confidenceBtoA	lift
0	0	Organic Strawberry Chia Lowfat 2% Cottage Cheese	Organic Cottage Cheese Blueberry Acai Chia	37	0.012063	128	0.041732	87	0.028365	0.289062	0.425287	10.190982
1	1	Yogurt, Sheep Milk, Strawberry	Yogurt, Sheep Milk, Blackberry	31	0.010107	110	0.035863	106	0.034559	0.281818	0.292453	8.154675
0	0	Organic Raspberry Yogurt	Organic Wildberry Yogurt	31	0.010102	109	0.035521	112	0.036498	0.284404	0.276786	7.792254
1	1	Organic Grapefruit Ginger Sparkling Yerba Mate	Cranberry Pomegranate Sparkling Yerba Mate	45	0.014664	177	0.057680	118	0.038454	0.254237	0.381356	6.611548
2	2	Yerba Mate Sparkling Classic Gold	Cranberry Pomegranate Sparkling Yerba Mate	36	0.011732	142	0.046275	118	0.038454	0.253521	0.305085	6.592924

```
In [10]: # Still have the same problem in duplicated idexes

In [11]: SortedMergeResult.loc[0,'itemA']
```

```
Out[11]: 0    Organic Strawberry Chia Lowfat 2% Cottage Cheese
         0                          Organic Raspberry Yogurt
         Name: itemA, dtype: object
```

6. Re-index the sorted merged DataFrame to solve the duplicated indexes.

```
In [12]: # Reindex the sorted merged dataframe

In [13]: SortedMergeResult.index = pd.RangeIndex(len(SortedMergeResult.index))

         SortedMergeResult.index = range(len(SortedMergeResult.index))

In [14]: SortedMergeResult.head(5)
```

Out[14]:

	Unnamed: 0	itemA	itemB	freqAB	supportAB	freqA	supportA	freqB	supportB	confidenceAtoB	confidenceBtoA	lift
0	0	Organic Strawberry Chia Lowfat 2% Cottage Cheese	Organic Cottage Cheese Blueberry Acai Chia	37	0.012063	128	0.041732	87	0.028365	0.289062	0.425287	10.190982
1	1	Yogurt, Sheep Milk, Strawberry	Yogurt, Sheep Milk, Blackberry	31	0.010107	110	0.035863	106	0.034559	0.281818	0.292453	8.154675
2	0	Organic Raspberry Yogurt	Organic Wildberry Yogurt	31	0.010102	109	0.035521	112	0.036498	0.284404	0.276786	7.792254
3	1	Organic Grapefruit Ginger Sparkling Yerba Mate	Cranberry Pomegranate Sparkling Yerba Mate	45	0.014664	177	0.057680	118	0.038454	0.254237	0.381356	6.611548
4	2	Yerba Mate Sparkling Classic Gold	Cranberry Pomegranate Sparkling Yerba Mate	36	0.011732	142	0.046275	118	0.038454	0.253521	0.305085	6.592924

7. Count how many lines PMRA implemented on first and second partition result file contain which mean the amount of the frequent item set contained in the merged DataFrame.

```
In [15]: # Lenth of the sorted merged dataframe after reindexing
         len(SortedMergeResult)
```

```
Out[15]: 100569
```

8. After merging there are duplicated rows depending in the itemA and itemB columns and keep the first frequent two items and drop the other because after sorting the first will be the higher so we will keep the higher lift value.

```
In [18]: # Now we search for duplcated rows depending in the itemA and itemB columns and keep
         # the first and drop the other because after sorting the first will be the higer
         # so we will keep the higher lift value
         dropDuplicteRow = SortedMergeResult.drop_duplicates(['itemA','itemB'], keep='first', inplace=False)
         dropDuplicteRow.head(5)
```

Out[18]:

	Unnamed: 0	itemA	itemB	freqAB	supportAB	freqA	supportA	freqB	supportB	confidenceAtoB	confidenceBtoA	lift
0	0	Organic Strawberry Chia Lowfat 2% Cottage Cheese	Organic Cottage Cheese Blueberry Acai Chia	37	0.012063	128	0.041732	87	0.028365	0.289062	0.425287	10.190982
1	1	Yogurt, Sheep Milk, Strawberry	Yogurt, Sheep Milk, Blackberry	31	0.010107	110	0.035863	106	0.034559	0.281818	0.292453	8.154675
2	0	Organic Raspberry Yogurt	Organic Wildberry Yogurt	31	0.010102	109	0.035521	112	0.036498	0.284404	0.276786	7.792254
3	1	Organic Grapefruit Ginger Sparkling Yerba Mate	Cranberry Pomegranate Sparkling Yerba Mate	45	0.014664	177	0.057680	118	0.038454	0.254237	0.381356	6.611548
4	2	Yerba Mate Sparkling Classic Gold	Cranberry Pomegranate Sparkling Yerba Mate	36	0.011732	142	0.046275	118	0.038454	0.253521	0.305085	6.592924

9. Count how many frequent items in the merged DataFrame.

```
In [19]: # Lenth of the new dataframe after dropping the duplicated rows
         len(dropDuplicteRow)
```

Out[19]: 57391

Now the merge operation for the first and second partitions from the PMRA implementation completed and it is ready to be compare with Traditional Apriori algorithm result file by repeating the same operations from section A. Repeat the same procedures for the rest PMRA results files.

Appendix D

The dataset splitter

This code in python language represent the dataset splitter. It is very simple, and designed to read a ***CSV*** file and load it to a variable. Then using ***for loop*** to read the data line by line. When the lines reach a specific number, which here in our case is (3300000) it creates a new file, gives it the same name of the original dataset file, and add a number for this file. Next, the loop starts again from the line coming after the last line in the first split file with repeating the same steps before.

```
import sys
fil=sys.argv[1]
csvfilename = open(fil, 'r').readlines()
file = 1
for j in range(len(csvfilename)):
   if j % 33000000 == 0:
      open(str(fil)+ str(file) + '.csv', 'w+').writelines(csvfilename[j:j+33000000])
      file += 1
```