AN EFFICIENT IMPLEMENTATION OF APRIORI ALGORITHM BASED ON HADOOP-MAPREDUCE MODEL

OTHMAN YAHYA, OSMAN HEGAZY, EHAB EZAT
Faculty of Computers and Information, Cairo University, Cairo, Egypt
E-mail: othman.cloudo@gmail.com , osman.hegazy@gmail.com , admin@drehab.net

ABSTRACT

Finding frequent itemsets is one of the most important fields of data mining. Apriori algorithm is the most established algorithm for finding frequent itemsets from a transactional dataset; however, it needs to scan the dataset many times and to generate many candidate itemsets. Unfortunately, when the dataset size is huge, both memory use and computational cost can still be very expensive. In addition, single processor’s memory and CPU resources are very limited, which make the algorithm performance inefficient. Parallel and distributed computing are effective strategies for accelerating algorithms performance. In this paper, we have implemented an efficient MapReduce Apriori algorithm (MRApriori) based on Hadoop-MapReduce model which needs only two phases (MapReduce Jobs) to find all frequent k-itemsets, and compared our proposed MRApriori algorithm with current two existed algorithms which need either one or k phases (k is maximum length of frequent itemsets) to find the same frequent k-itemsets. Experimental results showed that the proposed MRApriori algorithm outperforms the other two algorithms.

Keywords: Hadoop, MapReduce, Parallel Computing, Distributed Computing, Apriori Algorithm, Frequent Itemset, Data Mining, Association Rules.

1. INTRODUCTION

Data mining is the efficient discovery of previously unknown patterns in large datasets. It has attracted a lot of attention from both research and commercial communities for finding interesting information hidden in large datasets. One of the most important areas of data mining is association rule mining; its task is to find all subsets of items which frequently occur and the relationship between them by using two main steps: finding frequent itemsets (core step) and generating association rules [1].

Apriori [2] is the most established algorithm for finding frequent itemsets from a transactional dataset; however, it needs to scan the dataset many times and to generate many candidate itemsets. Unfortunately, when the dataset size is huge, both memory use and computational cost can still be very expensive. In addition, single processor’s memory and CPU resources are very limited, which make the algorithm performance inefficient. Furthermore; because of the exponential growth of worldwide information, enterprises (organizations) have to deal with an ever growing amount of data. As these data grow past hundreds of gigabytes towards a terabyte or more, it becomes nearly impossible to process (mine) them on a single sequential machine. The solution for the above problems is parallel and distributed computing.

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 which is a programming model for easily and efficiently writing applications that process vast amount of data in-parallel on large clusters of commodity hardware in a reliable, fault-tolerance manner [3]. More details about Hadoop-MapReduce will be illustrated in the next sections.

The rest of this paper is organized as follows: Section 2; presents background of Apriori algorithm, MapReduce model and Hadoop framework. Section 3, related work is discussed. Section 4, presents the proposed MRApriori algorithm. Section 5, performance evaluation. Conclusion is presented in section 6.
2. BACKGROUND

2.1. Apriori Algorithm

Apriori is the most classic and most widely used algorithm for mining frequent itemsets for Boolean association rules, proposed by R. Agrawal and R. Srikant in 1994 [2]. Figure 1 (at the end) shows the pseudo-code of Apriori algorithm.

2.2. MapReduce Model

MapReduce is one of the earliest and best known models in parallel and distributed (cluster) computing area, created by Google in 2004, based on C++ language. It is a programming model and associated implementation for processing and generating large data sets in a massively parallel and distributed manner [4]. One of the attractive qualities of MapReduce model is its simplicity: a MapReduce application (job) consists only of two functions, Map and Reduce functions. Developers write the map function that processes a key/value pair to generate a set of intermediate key/value pairs, and the reduce function that merges all intermediate values associated with the same intermediate key. Many data mining areas such as: association rule, clustering and classification algorithms have been implemented based on MapReduce model [5, 6, 7].

2.3. Hadoop Framework

Hadoop is an open source framework, created as an open source implementation of Google MapReduce architecture, based on Java language, sponsored by Apache Software Foundation. The Apache Hadoop software library is defined by Apache as “a framework that allows for the distributed processing of large data sets across clusters of computers using a simple programming model. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures”[8]. It originally consists of two models: MapReduce; the programming model, Hadoop Distributed File System (HDFS); the distributed storage model which designed after Google File System (GFS). Now it supports additional models and systems such as: HBase; a distributed column-oriented database, Hive; a data warehouse system, Avro; a data serialization system, Chukwa; a data collection system, ZooKeeper; a high-performance coordination service for distributed application, and Pig; a high level data-flow language [9].

Hadoop-MapReduce is a programming model for easily and efficiently writing applications which process vast amount of data (terabyte or more data sets) in-parallel on large clusters of commodity hardware in a reliable, fault-tolerance manner. A MapReduce program (job) partitions the input dataset into independent splits which are processed by the map tasks (map task per split) in a completely parallel manner. The Hadoop framework combines and stores the maps output as a set of intermediate key/value pairs which are then fetched as an input to the reduce tasks [3]. Figure 2 (at the end) shows the Hadoop-MapReduce architecture.

3. RELATED WORK

As we mentioned in the previous two sections, the performance of sequential Apriori algorithm is inefficient, especially when dealing with huge data sets. Thus, parallel Apriori algorithms were proposed [10, 11, 12, 13]. But in general, parallel and distributed computing are wide and varied fields and come with many problems that were not exists in sequential computing; so, a lot of time and effort are needed to handle and solve those problems. Such problems are: load balancing, data partition and distribution, jobs assignment and monitoring, parameters passing between nodes...etc. As mentioned above, parallel and distributed computing are wide and varied fields but the key distinctions of Hadoop are its simplicity, scalability, and reliability which solve most (if not all) of the challenges and problems listed above easily and efficiently.

Because of the benefits of MapReduce model, some parallel Apriori algorithms are implemented using Hadoop-MapReduce model [1, 14, 15, 16]. In general, we can classify those algorithms as two classes: one-phase and k-phases. In one-phase class; the algorithm needs only one phase (MapReduce job) to find all frequent k-itemsets [1], it sounds so good, but its execution time is very slow and its performance is inefficient, as we will see in section 5. Figure 3 shows the pseudo-code of this algorithm. In k-phases class (k is maximum
Figure 3. Pseudo-code of One-phase Algorithm

Map Task: // one for each split
Input: $S_i$ // Split i, line = transaction
Output: <key, 1> pairs, where key is an element of candidate itemsets.
1. For each transaction in $S_i$
2. Map(line offset,t) // Map function
3. Foreach itemset I in t /* I is all possible subsets of */
4. Out (I,1);
5. End foreach
6. End map
7. End foreach
8. End

Reduce Task:
Input: <key2, value2> pairs, minimum support_count, where key2 is an element of the candidate itemsets and value2 is its occurrence in each split.
Output: <key3, value3> pairs, key3 is an element of frequent itemsets and value3 is its occurrence in the whole dataset.
1. Reduce (key2, value2) // Reduce function.
2. Sum=0;
3. While (value2.hasNext())
4. Sum += value2.getNext();
5. End while
6. If (sum >= min_support_count)
7. Out (key2, sum);
8. End if
9. End reduce
10. End

Figure 4. Pseudo-code of k-phases Algorithm & k=1

Map Task: // one for each split
Input: $S_i$, $L_{k-1}$
Output: <key, 1> pairs, key is an element of candidate $k$-itemset
1. Read $L_{k-1}$ from DistributedCache.
2. $C_k$ = ap_gen($L_{k-1}$) // self-join
3. Foreach transaction in $S_i$
4. Map(line offset,t) // Map function
5. $C_t$ = subset ($C_k$, t);
6. Foreach candidate c in $C_t$
7. Out (c,1);
8. End foreach
9. End map
10. End foreach
11. End

Reduce Task:
The same reduce task as the previous phase
Figure 5. Pseudo-code of k-phases Algorithm & $k\geq2$

In this paper a new implementation of Apriori algorithm based on Hadoop-MapReduce model called MapReduce Apriori algorithm (MRApriori) is proposed. Subsequently, it is compared with the other two existed algorithms (one and k-phases). We conclude that the proposed MRApriori algorithm outperforms the others.

4. PROPOSED ALGORITHM

We have proposed a new implementation of Apriori algorithm based on Hadoop-MapReduce model, called MapReduce Apriori algorithm (MRApriori) algorithm. It only needs two MapReduce phases to find all frequent $k$-itemsets.

Map Task: // one for each split
Input: $S_i$ // Split i, line = transaction
Output: <key, 1> pairs, where key is an element of candidate $k$-itemset
1. For each transaction in $S_i$
2. Map(line offset,t) // Map function
3. Foreach itemset I in t // I is token
4. Out (I,1);
5. End foreach
6. End map
7. End foreach
8. End

Reduce Task:
Input: <key2, value2> pairs, minimum support_count, where key2 is an element of the candidate $k$-itemset and value2 is its occurrence in each split.
Output: <key3, value3> pairs, where key3 is an element of frequent $k$-itemset and value3 is its occurrence in the whole dataset.
1. Reduce (key2, value2) // Reduce function.
2. Sum=0;
3. While (value2.hasNext())
4. Sum += value2.getNext();
5. End while
6. If (sum >= min_support_count)
7. Out (key2, sum); // collected in $L_k$
8. End if
9. End reduce
10. End

Figure 6 (at the end) shows the data flow of our proposed MRApriori algorithm.
In phase one (solid arrows in figure 6), each input split is assigned a map task (executed by map worker) that calls a map function to process this split. What differ our map function from the previous related algorithms is the modification of its value parameter to take the whole split as an input, instead of one line (transaction) at a time, and then we apply the traditional Apriori algorithm on that split with partial minimum support count equal the number of transactions in the split multiply by the minimum support threshold. The map’s output is a list of intermediate key/value pairs: grouped by the key via combiner (optionally), and stored in the map worker; where the key is an element of partial frequent k-itemsets and the value is its partial count. When all map tasks are finished, the reduce task (executed by reduce worker) is started. The maps output are shuffled (fetched) to the reduce worker that calls a reduce function. The output of reduce function is a list (L_p) of key/value pairs, where the key is an element of partial frequent k-itemsets and the value equal one, stored in HDFS. Figure 7 shows the pseudo-code of this phase.

**Map Task:** // one for each split
**Input:** S_i, // Split i, line = transaction
**Output:** <key, value> pairs, where key is an element of partial frequent k-itemsets and value is its partial count
1. Map (object,S_i) // Map function
2. L = apply_Aprio_on(S_i); /* Partial_min_sup_count is used */
3. Foreach itemset I in L
4. Out (I, partial count);
5. End foreach
6. End map
7. End

**Reduce Task:**
**Input:** <key2, value2> pairs, where key2 is an element of the partial frequent k-itemsets and value2 is its occurrence in each split
**Output:** <key3, L_p> pairs, where key3 is an element of global candidate frequent k-itemsets
1. Reduce (key2, value2) // Reduce fun.
2. Out (key2, 1); // collected in L_p
3. End reduce
4. End

Figure 7 Pseudo-code of Phase-one of MRApriori Algorithm

In phase two (dashed arrows in figure 6), one extra input is added to the data flow of the previous phase, which is a file (copied from Hadoop DistributedCache, which in the stand-alone mode = local file system) that contains all partial frequent k-itemsets. The map function of this phase counts occurrence of each element of partial frequent k-itemset in the split and outputs a list of key/value pairs, where the key is an element of partial frequent k-itemset and the value is the total occurrence of this key in the split. The reduce function outputs a list (L_g) of key/value pairs, where the key is an element of global frequent k-itemsets (subset of partial frequent k-itemsets) and the value is its occurrence in the whole data set. Figure 8 shows the pseudo-code of this phase.

**Map Task:** // one for each split
**Input:** S_i, L_p
**Output:** <key, value> pairs, key is an element of L_p and value is its partial occurrence in the split
8. Read L_p from DistributedCache.
9. Foreach itemset I in L_p
10. Map (object, S_i) // Map function
11. count = Count_I in S_i(I, S_i);
12. Out (I, count);
13. End map
14. End foreach
15. End

**Reduce Task:**
**Input:** <key2, value2> pairs, where key2 is an element of the global candidate k-itemsets and value2 is its occurrence in each split
**Output:** <key3, value3> pairs, key3 is an element of global frequent k-itemsets and value3 is its global occurrence in the whole data set
11. Reduce (key2, value2) // Reduce fun.
12. Sum=0;
13. While(value2.hasNext())
14. Sum+= value2.getNext();
15. End while
16. If(sum>= min_sup_count)
17. Out (key3, sum); // collected in L_g
18. End if
19. End reduce
20. End

Figure 8. Pseudo-code of Phase-two of MRApriori Algorithm

5. PERFORMANCE EVALUATION

We have evaluated the performance of our proposed algorithm (MRApriori) by comparing its execution time with the execution time of the other two existed algorithms (one and k-phases).

In order to test the performance of the three algorithms, extensive experiments were done. All experiments were executed three times and the
average was taken. All experiments were performed on a single machine that contains: Windows 7 64-bit, Cygwin tool, Eclipse, Hadoop version 0.20.2 run on the stand-alone mode. All of the three algorithms were implemented in Java: JDK version is 1.6.31. The experimental dataset used is T10I4D100k which has been generated by IBM’s Quest Synthetic Data Generator. The total number of transactions is 100000, and each transaction contains 10 items in average, the total number of items is 1000, the average length of frequent itemsets is 4.

Figure 9 shows the performance of the three algorithms (one, two, and k-phases) with different data sets. The data sets (x-axis) are subsets of T10I4D100k, and the minimum support count =17. The results show that our proposed algorithm outperforms the other two algorithms and I think it will continue outperforms them as the data set size is increased.

**Note:** one-phase algorithm is inefficient and very time consuming; so, we will eliminate it from the next comparisons.

In figure 10, the performance with different minimum support (%) is shown. The used dataset is 2500 transactions. When minimum support = 0.52% the maximum length of frequent itemsets= 7, this means that k-phases algorithm executed only two MapReduce jobs, since k=2. However, our algorithm outperformed k-phases algorithm because it wrote more combine output records (pairs) than our. More illustration of combine output records is presented below.

In figure 10, the performance with different minimum support (%) is shown. The used dataset is 2500 transactions. When minimum support = 0.52% the maximum length of frequent itemsets= 7, this means that k-phases algorithm executed only two MapReduce jobs, since k=2. However, our algorithm outperformed k-phases algorithm because it wrote more combine output records (pairs) than our. More illustration of combine output records is presented below.

What essentially distinct our algorithm from k-phases algorithm is the number of MapReduce jobs (just two jobs) and the number of combine (map if no combine function) output records. Developers can optionally specify a combiner to perform local aggregation of the map function outputs, which helps to cut down the amount of records transferred from map worker to reduce worker. Many combine output records mean many output operations and more data transferring from map worker to reduce worker which affect the algorithm performance, especially on cluster environment. Figure 11 shows the number of combine output records related to data sets size and minimum support count =17.

**Note:** combine output records of one-phase algorithm is 134,870,641 records! That is when data set is 1000 transactions.
6. CONCLUSION

We have proposed a new implementation of the Apriori algorithm based on Hadoop-MapReduce model, called MRApriori algorithm. We have implemented three algorithms; MRApriori and the other two existed algorithms (one and k-phases) based on Hadoop-MapReduce programming model on hadoop platform running on the stand-alone mode and compared the performance of those algorithms. The results showed that: one-phase algorithm is inefficient and impractical; k-phases algorithm is effective and has execution time close to our proposed algorithm since, the experiments were done on a single machine and the combine output records did not move from map worker to reduce worker over the network; our proposed algorithm, MRApriori, is efficient and outperforms the other two algorithms in all experiments.

REFERENCES


Input:
- \( D \), a database of transactions;
- \( min\_sup \), the minimum support count threshold.

Output: \( L \), frequent itemsets in \( D \).

Method:
1. \( L_1 = \text{find\_frequent\_1\_itemsets}(D) \);
2. for \( (k = 2; L_{k-1} \neq \emptyset; k++) \) {
3. \( C_k = \text{apriori\_gen}(L_{k-1}) \);
4. for each transaction \( t \in D \) // scan \( D \) for counts
5. \( C_t = \text{subset}(C_k, t) \) // get the subsets of \( t \) that are candidates
6. for each candidate \( c \in C_t \)
7. \( c\text{.count}++ \);
8. \}
9. \( L_k = \{ c \in C_k | c\text{.count} \geq min\_sup \} \)
10. \}
11. return \( L = \cup_k L_k \);

procedure \text{apriori\_gen}(L_{k-1} \text{frequent \((k-1)\)-itemsets})
1. for each itemset \( l_1 \in L_{k-1} \)
2. for each itemset \( l_2 \in L_{k-1} \)
3. if \((l_1[1] = l_2[1]) \land (l_1[2] = l_2[2]) \land \ldots \land (l_1[k-2] = l_2[k-2]) \land (l_1[k-1] < l_2[k-1])\) then {
4. \( c = l_1 \setminus l_2 \) // join step: generate candidates
5. if \text{has\_infrequent\_subset}(c, L_{k-1}) \) then
6. \( \text{delete} \ c; \) // prune step: remove unfruitful candidate
7. else add \( c \) to \( C_k \);
8. \}
9. return \( C_k \);

procedure \text{has\_infrequent\_subset}(c \text{ candidate \( k \)-itemset};
\( \text{L}_{k-1} \) \text{ frequent \((k-1)\)-itemsets}); // use prior knowledge
1. for each \((k-1)\)-subset \( s \) of \( c \)
2. if \( s \notin L_{k-1} \) then
3. return TRUE;
4. return FALSE;

Figure 1. Pseudo-code of Apriori Algorithm. (Source: [2]).
Figure 2. Hadoop MapReduce Architecture. (Source: [9]).

Figure 6. Data Flow of MRApriori Algorithm.