Distributed Apriori in Hadoop MapReduce Framework

By Shulei Zhao (sz2352) and Rongxin Du (rd2537)

Individual Contribution:
Shulei Zhao: Implements centralized Apriori algorithm and input preprocessing code in Java
Rongxin Du: Configures distributed environment and implements distributed Apriori using Hadoop MapReduce APIs.

Cooperative Contribution:
Debugging and code cleaning

1. Introduction

This final report will focus on how we transform the centralized Apriori [1] algorithm to a distributed structure that could be applied in MapReduce framework and produces the same correct results as the centralized one. However, the MapReduce distributed framework is essentially a shared-nothing distributed structure, which means that each worker (slave) can only communicate with the master node and we prohibit communication among the workers (salves).

In our implementation, we use a different combination of loop and recursion algorithm to implement the apriori_gen() and subset() function in Apriori, which will be discussed later. Let’s first briefly review these two important functions in the traditional Apriori.

The apriori_gen() function takes the set of all \((k-1)\) large itemset as arguments and returns a superset of the \(k\) large itemset as the candidate set. The outer for loop keep iterating \(k\) to further generate the candidate set for all levels of large itemset. Noticing that the \((k-1)\) itemset argument is observed as global large, thus we can’t make a meaningful distributed form of this function (for sure, you can replicate the \((k-1)\) large itemsets among all the sites, but this is trivial and makes no sense).

Suppose for the \(k\)-th iteration in the outer for loop, the candidate set for \(k\) large itemset is generated by apriori_gen(), \(C_k = \text{apriori-gen}(L_{k-1})\). For this \(C_k\), we further search the whole transaction database (the inner for loop), using each line (a transaction of items) to refine (prune) \(C_k\) to a truncated subset \(C_t\), which should also be the superset of \(L_k\). This process is possible to be distributed in MapReduce, as long as each site \(i\) could get access to the global candidate set \(C_k\). For each worker site \(i\), simply scanning local data partition \(D_i\) and pruning \(C_k\) to \(C_{k(i)}\). For the master node, collect the \(C_{k(i)}\) from workers to calculate \(C_k = \bigcup C_{k(i)}\), supposing there are \(N\) partitions on \(N\) worker sites.

The occurrences counting of itemsets in the refined candidate set \(C_t\) could be done on-the-fly along with the condition filter that \(\text{count} > \text{min_sup}\) when the master node is doing the above reduce work.
2. Distributed Apriori in MapReduce

The following diagram illustrates the components and data flow of our distributed Apriori in MapReduce framework. The green downward arrow represents the source data we are going to process from a transaction centralized database. All the input and output files are formatted in HDFS. We take 3 map-reduce steps to accomplish the Apriori. A MapReduce first maps the input dataset to $N$ partitions (where $N$ is equal to the number of slave machines or daemons) and meanwhile emits a key-value pair $<item, 1>$ for each item in a line. Then the reduce phase would take the intermediate key-value pairs emitted in the map phase and send them to the master node to collect the count per item.

The first job component in Figure 1 is to generate frequent 1 items in the form of $<item, count>$ key-value pair (where the count indicates the number of occurrences), which are also added to the global HashMap item list. $O_1$ is the output file of the frequent items job.

The second job component is to generate candidate sets. This candidate generate job also takes the source data file. In the map phase, it firstly prunes those items that occur less than the support threshold by looking up the global HashMap item list, and then recursively calls functionCandidatesGenRecursion() to generate candidates. The reduce phase groups the same candidate itemset and collect the count as $<candidate-itemset, count>$ key-value pairs, which are written as the output and also added to the global HashMap comb.

The last job component called Association Rule takes the output of the candidate generate job as the input file and mining association rules on the candidate itemsets. Note that we prune the left hand of an association rule by looking up its count in the HashMap comb and...
calculate the confidence: \( \text{Association Rule } \{X \rightarrow XY\}.\text{conf} = \frac{XY.\text{count}}{X.\text{count}} \)

3. New Way for Candidate Generation

We design the following candidate generation algorithm which is a loop and recursion fashion.

INPUT: a list of frequent 1 large item set
OUTPUT: All possible candidate itemsets (superset of the large itemsets)

In the i-th iteration of for-loop

- Generate different choices of item-pairs(2 items as a pair )
  \[ [i, i+1], [i, i+2], [i, i+3], \ldots, [i, \text{INPUT.length}] \]

In the j-th level of recursion

- generate different sizes of candidate for a given choice
  \[ [i, i+1] \rightarrow [i, i+1, i+2] \rightarrow [i, i+1, i+2, i+3], \ldots \]
  \[ [i, i+2] \rightarrow [i, i+2, i+3] \rightarrow [i, i+2, i+3, i+4], \ldots \]
  \[ \ldots \]
  \[ [i, \text{INPUT.length-1}] \rightarrow [i, \text{INPUT.length-1, INPUT.length}] \]

4. Association Rule Mining

We did an interesting trick here to mine the association rules. We reuse the loop + recursion idea, basically the candidate generation algorithm illustrated above, except that we substitute INPUT with the k-large itemset. Then the output will be possible left hand body of the association rule. For example, if the 3-large itemset is \( \{1,2,3\} \), we will use the loop + recursion algorithm to generate \( \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\} \) as the left-hand body, and the trivial one \( \{1,2,3\} \rightarrow \{1,2,3\} \) will be discarded. Accordingly, we might mine out the following association rules (we need further prune, prune techniques will be discussed later)

\( \{1,2\} \rightarrow \{1,2,3\} \) (equivalent to \( \{1,2\} \rightarrow \{3\} \))
\( \{1,3\} \rightarrow \{1,2,3\} \) (equivalent to \( \{1,3\} \rightarrow \{2\} \))
\( \{2,3\} \rightarrow \{1,2,3\} \) (equivalent to \( \{2,3\} \rightarrow \{1\} \))

5. Prune Techniques

We are using the following 4 prune techniques to reduce the candidate sets size and also to ensure that the output of the 2nd job (candidate generate in Figure 1) is also the large itemsets.

1. In the candidate generate job map phase, we only take those items with \( \text{count} > \text{support} \) as the INPUT of the loop + recursion algorithm described in above Candidate Generation section;
   In addition, here, we have to keep a global HashMap item list to look up the item count.
2. In the candidate generate job reduce phase, we only output those candidate itemsets with \( \text{count} > \text{support} \). These output candidate itemsets are exactly the large itemsets globally.
In addition, here, to keep track of the large itemsets, we have to keep a global HashMap comb to save these large itemsets for future lookup.

(3). In the association rule job map phase, since we take the output of the previous job as the right-hand body, the right-hand body will always be large itemsets. For every generated left-hand body by using loop + recursion algorithm on the right-hand body, we have to check whether it is globally large. To achieve this, we simply look up the HashMap comb we saved in the previous job and retrieve the count.

(4). Association rule prune on calculating the confidence, trivial.

6. Experiment and Results

Our data comes from [2], which consists of 1000 lines of market-basket transactions. There are basically 4 modes in Hadoop MapReduce:

(1). stand-alone mode (the same machine as the master and also the slave);
(2). Pseudo-Distributed mode (where each Hadoop daemon runs in a separate Java process);
(3). Cluster mode (need a cluster of real machines);
(4). Using Amazon Elastic MapReduce;

We test our code under mode 1 and mode 2, and it works giving the correct result.

Here is the first 3 records from the result when support = 10%, confidence = 80%, for example the first association rule can be read as customers who buy artichok and ham also tend to buy heineken, the left-hand occurs 115 times and the right-hand occurs 111 times, the confidence is 96.52%

\{artichok, ham\} => \{artichok, heineken, ham\} \{X\}:115 => \{X,Y\}:111
Confidence(percentage)= 96.521736

\{artichok, ham\} => \{avocado, artichok, heineken, ham\} \{X\}:115 => \{X,Y\}:111
Confidence(percentage)= 96.521736

\{avocado, artichok, ham\} => \{avocado, artichok, heineken, ham\} \{X\}:111 => \{X,Y\}:111
Confidence(percentage)= 100.0

Another interesting issue is that our distributed Apriori algorithm is sensitive to the order of items in a single transaction and also the duplicates. For example, if transaction is like “apple, orange, apple”, in our algorithm \{apple, orange\} and \{orange, apple\} will be treated as different candidate itemset, and also \{apple, apple\} could also be a candidate, which makes no sense. However, this problem is very common, think about when you check out at the cashier in a supermarket, of course you might buy 2 apples together. In order to solve this problem, we write a sorting program to sort the input file to ensure a transaction (a line in CSV) is arranged in increasing numeric or alphabetic order and eliminate all the duplicates.

7. Complexity Analysis
Apriori algorithm faces the following difficulties [3]: It spends so much time dealing with particularly large number of candidate sets since candidate \((k + 1)\) itemsets are constructed through the self-join of frequent \(k\)-itemsets. Its growth is exponential.

Here we didn’t use the self-join, instead, we use a recursion enclosed by the for-loop. The time complexity is about the same, \(O ((\text{INPUT. Length})^2)\), however, the subtle difference is that we choose the space as the tradeoff of time. As we know, recursive function calls occupies lots of run-time function frames in the stack. Suppose that we map into \(N\) partitions, the memory overhead compared to a single partition would be \(I/N\). To conclude we achieve the same goal of generating candidates without using the self-join operation.

8. Problems and Future Work

We got stuck in running in Amazon Elastic MapReduce, the problem is that the global variables are not visible on the cloud to other virtual machines, and it seems that Hadoop will not distribute these global variables to be shared by every partition due to its nature shared-nothing architecture. We try many methods to get it work, including writing the global variable as files to Amazon S3 (Simple Storage Service) file system by giving the path like s3n://folder/file. However even though we can write files on the cloud, the files seem not visible to Hadoop HDFS, but we don’t know how to reformat files on cloud to the HDFS.

Since testing MapReduce code in stand-alone and Pseudo-Distributed mode is trivial, and it will be even slower than running a centralized algorithm, because of the partitioning and collecting overheads. We are really eager to figure out how to solve the above problem and run our code on the cloud to process huge data like 100M or more, and also testing the speed-up and scalability of our distributed Apriori by opening up multiple slave machines in EMR [5]. Maybe we can also try to use the DistributedCache [6] to cache files and pass to a mapper by overwrite the \textit{setup()} function.

References

[2] AssociationsSP.txt, High-Performance Information Computing Center, Computer Information Systems Department of California State University, Los Angeles, April 2012