Frequent itemset mining using parallel Apriori Algorithm
Overview

What is Frequent Itemset Mining?

Association rule learning is a rule-based machine learning method for discovering interesting relations between variables in large databases. It is intended to identify strong rules discovered in databases using some measures of interestingness.

Association rules are usually required to satisfy a user-specified minimum support and a user-specified minimum confidence at the same time.

Association rule generation is usually split up into two separate steps:

1. A minimum support threshold is applied to find all frequent itemsets in a database.

2. A minimum confidence constraint is applied to these frequent itemsets in order to form rules.

Description of the computational problem

Apriori is an algorithm for frequent item set mining and association rule learning over transactional databases. It proceeds by identifying the frequent individual items in the database and extending them to larger and larger item sets as long as those item sets appear sufficiently often in the database. The frequent item sets determined by Apriori can be used to determine association rules which highlight general trends in the database: this has applications in domains such as market basket analysis.
Example application of Apriori Algorithm

Challenges for Apriori Algorithm

- Lower Support = Greater Time
- To discover a frequent pattern of size N, 2N candidates need to be generated
- Multiple Database Scan
Analysis of Research Papers

For further understanding how this problem is approached and solved, we went through 3 research papers.

**Research Paper 1 - Apriori-based Frequent Itemset Mining Algorithms on MapReduce:**

- Scalability of mining algorithm is prone to errors; thus it becomes necessary to use a MapReduce based framework in order to make an efficient reliable parallel Frequent Itemset Mining algorithm.

- Converting a serial Apriori-like mining algorithms into a distributed algorithm on the MapReduce framework is not difficult but the mining performance is unsatisfactory. Multiple-pass on the algorithm necessitates multiple map-reduce phases, thus a good scheduling scheme is needed such that it reduces the number of scheduling invocations, maximize the utilization of each node during a single MapReduce phase and perform smart Load Balancing.

- The paper compares three possible approaches to schedule the nodes in the cluster so that we can perform MapReduce task efficiently for each phase.

- SPC algorithm, where it iterates over the dataset once in every phase of the MapReduce.

- FPC algorithm, where it iterates over the dataset k times when the maximum length of the frequent itemsets is k.

- In order to avoid algorithms from suffering from overloading candidates for a mapper if the number of candidates after merging is too large, we use DPC, which dynamically schedules the nodes for each phase.

- The paper concludes DPC can be used to strikes a balance between reducing the number of map-reduce phases and increasing the number of prunes candidates. Thus, is outperforms the other two methods and can be used for efficient Frequent Itemset Mining with MapReduce
Map(key, value = itemset in transaction $t_j$):
Input: a database partition $D_i$ and $L_{k-1}$ (k > 2)
1. read $L_{k-1}$ from DistributedCache;
2. Candidate threshold $ct = \alpha \cdot |L_{k-1}|$;
3. $C_{set} = C_k = \text{apriori-gen}(L_{k-1})$;
4. for (counter = 0; $|C_{set}| \leq ct$; counter++)
5. $C_{k+1+\text{counter}} = \text{apriori-gen}(C_{k+\text{counter}})$;
6. $C_{set} = C_{set} \cup \text{apriori-gen}(C_{k+1+\text{counter}})$;
end
8. construct a prefix-tree for $C_{set}$;
9. foreach transaction $t_i \in D_i$ do
10. $C_i = \text{subset}(C_{set}, t_i)$;
11. foreach candidate $c \in C_i$ do
12. output $<c, 1>$;
end
end

Reduce (key=itemset, value=count):
1. foreach key y do /* Initial y.count = 0 */
2. foreach value v in y's value list do
3. y.count += v;
4. end
5. if y.count $\geq$ minimum support count
6. output $<y, y\text{.count}>$; /* collected in $L_k \ldots L_{k+\text{counter}}$ */
7. end
8.end

Figure 8. Phase-k (k>2) of DPC algorithms.

Algorithm DPC
1. Phase-1: find $L_1$ /* Figure 1 */
2. Phase-2: find $L_2$ /* Figure 2 */
3. for (k = 3; $L_{k-1} \neq \phi$) /* each phase k */
4. Map function /* Figure 8 */
5. Reduce function /* Figure 8 */
6. $k += (\text{counter}+1)$;
7. end

Figure 9. Algorithm DPC.
Problem:

- Generally, it takes long to find the association rules between datasets when a database contains a large number of transactions.
- By applying parallel-distributed data mining techniques, the mining process can be effectively speeded up.
- With parallel-distributed data mining the calculation is done in a distributed environment but most of the time, irregular and imbalanced computation loads are allocated between processors and thus the overall performance is degraded.

Solution:

In this paper the Weighted Distributed Parallel Apriori algorithm (WDPA) is presented as a solution for this problem. In the proposed method, a database has only to be scanned once because metadata are stored in TID tables. This approach also takes the TID count into consideration. Therefore, WDPA improves load-balancing as well as reduces idle time of processors

\[
\text{Cnt\_Lattice}(I_i) = [(\text{len}(\text{freq}_{k-1}) - 1) - i]
\]

\[
\text{TotalCnt\_Lattice} = \sum_{i=0}^{\text{len}(\text{freq}_{k-1})-1} \text{Cnt\_Lattice}(I_i)
\]

By only calculating the Lattice number and ignoring the length of the itemset TID, an uneven distribution of workload occurs. Therefore, this algorithm also takes TID length into consideration and regards it as a weight value, which makes the distribution of itemsets more accurate and more even.
Paper 3 - Efficient Mining Using Enhanced Apriori with Hash Tree and Fuzzy:

- Their design tool allows experimenting with the concepts of fuzzy modification of association rules.
- They then analyzed the crisp boundary problem in the algorithm and how it can be overcome by a modified association Apriori hash tree fuzzy algorithm.
- Goal is to increase efficiency of overall system by compromising with the accuracy.

\[
Value_{-WeightTid}(I_i) = \sum_{j=1}^{\text{len}(\text{freq}_{k-1})-1} \text{len}(I_{i_{TID}}) \times \text{len}(I_{j_{TID}})
\]

\[
TotalValue_{-WeightTid} = \sum_{i=0}^{\text{len}(\text{freq}_{k-1})-1} \sum_{j=1}^{\text{len}(\text{freq}_{k-1})-1} \text{len}(I_{i_{TID}}) \times \text{len}(I_{j_{TID}})
\]

<table>
<thead>
<tr>
<th>Items</th>
<th>Count</th>
<th>TID</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>4</td>
<td>1,3,4,5</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>1,2,5</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
<td>3,4,5</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
<td>2,3</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
<td>2,4</td>
</tr>
<tr>
<td>M</td>
<td>2</td>
<td>2,3</td>
</tr>
<tr>
<td>O</td>
<td>2</td>
<td>2,5</td>
</tr>
<tr>
<td>P</td>
<td>2</td>
<td>1,5</td>
</tr>
</tbody>
</table>

Total WeightTid = 173
Average WeightTid = 86
Ways to improve efficiency

- Prune without checking all k-1 itemsets
- Join without looping over entire set L_{k-1}
- Reduce the number of transactions
- Reduce the number of candidates
- Reduce the number of subsets to be considered per transaction

Sequential Design and Operation of the Program

In the sequential design of the Apriori algorithm. We have to take the straight forward approach of iteratively finding the itemset which are above the minimum support criteria.

In order to do so, we have to first get the 1-itemset count for each transaction in the data source. From this, we will generation the candidates over which we have to find the minimum support.

From the candidate items, we make a frequent set for each of the candidate items and make a map of each candidate items and their corresponding frequencies.

After this step, we eliminate all the candidates which don’t pass the criteria given by the minimum support.

Over the candidates which pass the criteria, we have to make a new map which contains the subset of size n+1, where n is the size of the candidate in the current iteration of the Apriori Algorithm.

After this step, we re-iterate and start the next phase of the apriori algorithm.
Input: database of transactions $D$; the minimum support count threshold ($min\ sup$)
Output: frequent itemsets in $D$ ($L$)

1. $L_1 = \text{find\ frequent\_1\_itemsets}(D)$;
2. \textbf{for} (k = 2; $L_{k-1} \neq \emptyset$; k++) {
3. \hspace{1em} $C_k = \text{apriori\_gen}\ (L_{k-1})$;
4. \hspace{1em} \textbf{for each} transaction $t \in D$ { // scan $D$ for counts
5. \hspace{2em} $C_t = \text{subset}\ (C_k, t)$; // get the subsets of $t$ that are candidates
6. \hspace{2em} \textbf{for each} candidate $c \in C_t$
7. \hspace{3em} $c$.count++;
8. \hspace{1em} }
9. \hspace{1em} $L_k = \{c \in C_k | c$.count $\geq min\_sup\}$
10. \}
11. \textbf{return} $L = \cup_k L_k$;
Function apriori gen($L_{k-1}$: 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 ($l_1[1]=l_2[1]$) $\wedge$ ($l_1[2]=l_2[2]$) $\wedge$ ... $\wedge$ ($l_1[k-2]=l_2[k-2]$) $\wedge$ ($l_1[k-1] < l_2[k-1]$) then {
      (4)          $c = l_1 \triangledown l_2$; // join step: generate candidates
      (5)          If has_infrequent_subset($c$, $L_{k-1}$) then
          (6)               delete $c$; // prune step: remove infrequent candidate
      (7)          else add $c$ to $C_k$;
  (8)      }
(9) return $C_k$;

Procedure has infrequent_subset($c$: candidate k-itemset; $L_{k-1}$: frequent (k-1)-itemsets); // use prior knowledge
(1) for each (k-1)-subset $s$ of $c$
(2)   if $s \not\in L_{k-1}$ then
(3) return TRUE;
Parallel Design and Operation of the Program

In our approach for parallel computing, we develop a map-reduce based approach with same number of minimum support for the itemsets for frequent itemset mining. We begin by finding the amount of time each individual itemset is occurring the data and then we filter the itemsets based on the minimum support. After this set, we create the subsets of the remaining itemsets which are beyond the minimum support criteria and then start the next phase of the Apriori Algorithm.

In the mapper the and reducer task, the following things happen respectively.

First all the itemset’s occurrence is counted across each of the transaction in the data source file. This is done in parallel across many of the parallel mappers running together. The combiner then combines the total count of the number of itemsets occurring in each of the mapper and this is sent to the reducer.

In the reducer, we first check whether each pair of itemset and it’s value that we get through the combiner is above the minimum support requirement or not. If it passes the criteria, then we add to a new map of itemset and it’s value. After all the itemset’s item are checked and a new map is created, we start a new phase of Apriori algorithm with these itemsets to check. Before starting a new phase of the Apriori algorithm, we also to create a subset of size n+1, based on the size of the itemset in the current iteration. This is done in a sequential code, but if there are more than one thread for reducer, this can occur in parallel, thus reducer the bottleneck of the entire program.
This entire process is continued until we reach a point, where the total number of itemsets possible for the next iteration is zero. At this point, we stop our program.

Another possible stoppage point for the program is when we can want the size of our itemset set to be of a particular size. If it reaches that size, we stop the process.

**Flow of main program**

![Flowchart of main program]

**Flow of parallel program**

![Flowchart of parallel program]
Parallel Apriori Pseudo-Code
User’s Manual:

1. Using the CS Tardis machine, we have to keep in mind to use the JDK version 1.7 and the class path to include the PJ2, more details can be found at:
   http://www.cs.rit.edu/~ark/runningpj2.html
2. We have to place the required files in the same directory along with the modified version of the PJ2 library. The modified version should be included in the same directory as well.

3. For parallel program we made slight modifications to some source files of pj2 library. So, in order to compile our program, you need to use our jar file instead of original pj2 jar file. Include our version of pj2 library in the machine’s classpath with the command,
   ```
   export CLASSPATH=./current_directory/pj2.jar
   ```

4. Now, compile the program with the command, javac *.java

5. Now we have to generate the jar file of the build directory, in order to do that, simple enter the following command on the console:
   ```
   jar cf pf.jar *.class
   ```
Developer’s Manual

Database file:
Here’s an example of a sample database file we used for this project.

As you can see each line represents a single transaction and each column represents a different item.

Sequential Program:
In order to run the sequential program, the following files are required:

AprioriSeq.java

Parallel Program:
In order to run the parallel program, the following files are required:

AprioriMR.java
AprioriMR2.java
startProject.java
pj2.jar
Partitioned transaction database files

How to run the programs:

**Steps to compile and run the parallel program:**

1. To run the program, use following command:

   ```
   java startProject
   ```

   In order to change the minimum support in the parallel program, the user has to change the input string in the startProject file.

2. For the Sequential program, run using following command:

   ```
   java pj2 AprioriSeq cores=<k> <input file> <min_support>
   ```

   where `<k>` = Number of cores
   
   `<input file>` = transaction database file
   
   `<min_support>` = support to generate frequent itemsets
   
   Must be 0 to 1. Default value is 0.8

**Strong Scaling performance data:**

<table>
<thead>
<tr>
<th>Size</th>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>N = 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>87465</td>
<td>0.787</td>
<td>0.787</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>51834</td>
<td>1.328</td>
<td>0.664</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>29069</td>
<td>2.368</td>
<td>0.592</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>18076</td>
<td>3.808</td>
<td>0.476</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>17695</td>
<td>3.890</td>
<td>0.389</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>19578</td>
<td>3.516</td>
<td>0.293</td>
</tr>
<tr>
<td>N = 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>169753</td>
<td>0.811</td>
<td>0.811</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>94683</td>
<td>1.454</td>
<td>0.727</td>
</tr>
<tr>
<td>N</td>
<td>Value1</td>
<td>Value2</td>
<td>Value3</td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>50763</td>
<td>2.712</td>
<td>0.678</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>28729</td>
<td>4.792</td>
<td>0.599</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>27047</td>
<td>5.090</td>
<td>0.509</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>31260</td>
<td>4.404</td>
<td>0.367</td>
<td></td>
</tr>
</tbody>
</table>

**N = 3**

<table>
<thead>
<tr>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>244095</td>
<td>0.846</td>
</tr>
<tr>
<td>2</td>
<td>134794</td>
<td>1.532</td>
</tr>
<tr>
<td>4</td>
<td>72508</td>
<td>2.848</td>
</tr>
<tr>
<td>8</td>
<td>44428</td>
<td>4.648</td>
</tr>
<tr>
<td>10</td>
<td>38312</td>
<td>5.390</td>
</tr>
<tr>
<td>12</td>
<td>40020</td>
<td>5.160</td>
</tr>
</tbody>
</table>

**N = 4**

<table>
<thead>
<tr>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>313242</td>
<td>0.879</td>
</tr>
<tr>
<td>2</td>
<td>176726</td>
<td>1.558</td>
</tr>
<tr>
<td>4</td>
<td>92025</td>
<td>2.992</td>
</tr>
<tr>
<td>8</td>
<td>52545</td>
<td>5.240</td>
</tr>
<tr>
<td>10</td>
<td>47228</td>
<td>5.830</td>
</tr>
<tr>
<td>12</td>
<td>46922</td>
<td>5.868</td>
</tr>
</tbody>
</table>

**N = 5**

<table>
<thead>
<tr>
<th>Value1</th>
<th>Value2</th>
<th>Value3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>377799</td>
<td>0.911</td>
</tr>
<tr>
<td>2</td>
<td>197532</td>
<td>1.740</td>
</tr>
<tr>
<td>4</td>
<td>106754</td>
<td>3.224</td>
</tr>
<tr>
<td>8</td>
<td>64116</td>
<td>5.368</td>
</tr>
<tr>
<td>10</td>
<td>59238</td>
<td>5.810</td>
</tr>
<tr>
<td>12</td>
<td>51308</td>
<td>6.707</td>
</tr>
</tbody>
</table>
Discussion of non-ideal Strong Scaling:

Reducer’s finish method is giving a huge overhead due to subset creation, therefore with an increase in number of cores, efficiency decreases significantly.

Weak Scaling performance data:

<table>
<thead>
<tr>
<th>Case</th>
<th>Size</th>
<th>Cores</th>
<th>Time</th>
<th>Speedup</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>N = 1</td>
<td>1</td>
<td>1</td>
<td>85403</td>
<td></td>
<td>0.806</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>83335</td>
<td></td>
<td>0.826</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>4</td>
<td>93653</td>
<td></td>
<td>0.735</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>8</td>
<td>89386</td>
<td></td>
<td>0.770</td>
</tr>
<tr>
<td>N = 2</td>
<td>10</td>
<td>10</td>
<td>86803</td>
<td>0.793</td>
<td></td>
</tr>
<tr>
<td>N = 2</td>
<td>12</td>
<td>12</td>
<td>82734</td>
<td>0.832</td>
<td></td>
</tr>
<tr>
<td>N = 3</td>
<td>2</td>
<td>1</td>
<td>166469</td>
<td>0.827</td>
<td></td>
</tr>
<tr>
<td>N = 3</td>
<td>4</td>
<td>2</td>
<td>173378</td>
<td>0.794</td>
<td></td>
</tr>
<tr>
<td>N = 3</td>
<td>8</td>
<td>4</td>
<td>172087</td>
<td>0.800</td>
<td></td>
</tr>
<tr>
<td>N = 3</td>
<td>16</td>
<td>8</td>
<td>165667</td>
<td>0.831</td>
<td></td>
</tr>
<tr>
<td>N = 3</td>
<td>20</td>
<td>10</td>
<td>185539</td>
<td>0.742</td>
<td></td>
</tr>
<tr>
<td>N = 3</td>
<td>24</td>
<td>12</td>
<td>176274</td>
<td>0.781</td>
<td></td>
</tr>
<tr>
<td>N = 4</td>
<td>3</td>
<td>1</td>
<td>254003</td>
<td>0.813</td>
<td></td>
</tr>
<tr>
<td>N = 4</td>
<td>6</td>
<td>2</td>
<td>238183</td>
<td>0.867</td>
<td></td>
</tr>
<tr>
<td>N = 4</td>
<td>12</td>
<td>4</td>
<td>256209</td>
<td>0.806</td>
<td></td>
</tr>
<tr>
<td>N = 4</td>
<td>24</td>
<td>8</td>
<td>263399</td>
<td>0.784</td>
<td></td>
</tr>
<tr>
<td>N = 4</td>
<td>30</td>
<td>10</td>
<td>258454</td>
<td>0.799</td>
<td></td>
</tr>
<tr>
<td>N = 4</td>
<td>36</td>
<td>12</td>
<td>254316</td>
<td>0.812</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>4</td>
<td>1</td>
<td>310417</td>
<td>0.887</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>8</td>
<td>2</td>
<td>346339</td>
<td>0.795</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>16</td>
<td>4</td>
<td>360865</td>
<td>0.763</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>32</td>
<td>8</td>
<td>348091</td>
<td>0.791</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>40</td>
<td>10</td>
<td>323169</td>
<td>0.852</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>48</td>
<td>12</td>
<td>314315</td>
<td>0.876</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>5</td>
<td>1</td>
<td>376559</td>
<td>0.914</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>10</td>
<td>2</td>
<td>382416</td>
<td>0.900</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>20</td>
<td>4</td>
<td>385846</td>
<td>0.892</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>40</td>
<td>8</td>
<td>372887</td>
<td>0.923</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>50</td>
<td>10</td>
<td>397890</td>
<td>0.865</td>
<td></td>
</tr>
<tr>
<td>N = 5</td>
<td>60</td>
<td>12</td>
<td>401604</td>
<td>0.857</td>
<td></td>
</tr>
</tbody>
</table>
Discussion of non-ideal Weak Scaling:

Weak scaling gives us better result than strong scaling because it depends on mapper’s task which is very efficient in our program. Still we get less than ideal results due to huge overhead in reducer class due to subset creation.

Possible Future Work

It might be possible to speedup subset generation process by further parallelizing the reduction class using parallelFor loops. We are also planning to implement this in GPU programming. By utilizing large number of GPU cores, it might possible to improve on huge bottleneck performance we are getting.

Also, although for this project we were restricted by use of pj2 library and java, it might be possible to build a better and more efficient tool using other programming languages such as Python and other libraries or frameworks such as Hadoop.

What We Learned from the Project

1. Association rule mining is a NP-hard problem therefore algorithm like Apriori although gives accurate solution, but still not very efficient as generation of subsets creates a major overhead.
2. Apriori is uses an exhaustive search to find all the possible matches and subsets, so it’s very difficult to use with limited computational capabilities. A parallel implementation reduces the time considerably, making it more feasible. However, with increasing size of data, time still increases exponentially.
3. For problem like Apriori its more effective to use mapper and reducer than other form of parallel implementations as it requires multiple database scan and mapper-reduce structure allows us to divide data into chunks and feed each chunk into different mapper.
4. Parallel pj2 library doesn’t directly allow us to run more than one job one after the other. Therefore, we had to make some changes to source files and compile our own pj2.jar in order to run our program where it is possible to restart the map reduce task and start the second phase of the Apriori algorithm.
Contribution of each team member

Every step of our project was a team effort. We sat and discussed every part of our project. For sequential part of the program we pretty much followed the pseudo-code of Apriori we found on a paper we investigated, and Arjun converted it into java program. For Parallel program Arjun and Vrishabh both worked together and using sequential program as reference divided it into parts by structuring it in a map-reduce format. Since our program logic required certain things that were not directly possible with pj2, Vrishabh rewrote some pj2 classes and Arjun compiled them together. Arjun then wrote and reduce function while Vrishabh worked on main program logic and iterating between different phases of map-reduce job which is not directly possible in pj2. For presentations, we divided contents and then later combined them before presentation. For scaling and performance measure again, we worked together and there were lots of discussions.

References


