REPORT

Submitted by:
2019201072(Sai Rohith Areti)
2019201010(P V Haridatta)

1)Apriori:
Optimizations Used:
We used Transaction Reduction in our implementation for pruning the transactions which do not have all the immediate subsets as frequent Item Sets.
We used Hash Based Technique as the other Optimization.

2)**FPTree:**
Optimization applied(Strategy used):merging optimization
procedure: Applied as mentioned in the doc.we just traversed the once per conditional fp tree instead of traversing from each node in fp tree to root .In our code the function merging strategy does the same thing it is given fp tree as one of the arguments we traverse the tree only once per conditional fp tree and we maintain a dictionary for each where each frequent item is key and the set of all conditional databases as value
When ever we want conditional pattern base of one of frequent item we just check the dictionary Conditional Databases and use it in our calculations
Function in code: mergingStategy
Node structure in code:

```
class TreeNode:
   def __init__(self, Node_name,counter):
      self.name = Node_name
      self.count = counter
      self.nodeLink = None
      self.children = {}
```

Here we are not storing parents as we dont do upward traversal from node to root.
count-->count of that item
nodeLink-->header node links from header table to nodes
children-->children to each node

BONUS PART:
We have done bonus part

Then what are closed and maximal frequent itemsets?By definition, An itemset is maximal frequent if none of its immediate supersets is frequent. An itemset is closed if none of its

immediate supersets has the same support as the itemset. Let's use an example and diagram representation to better understand the concept.

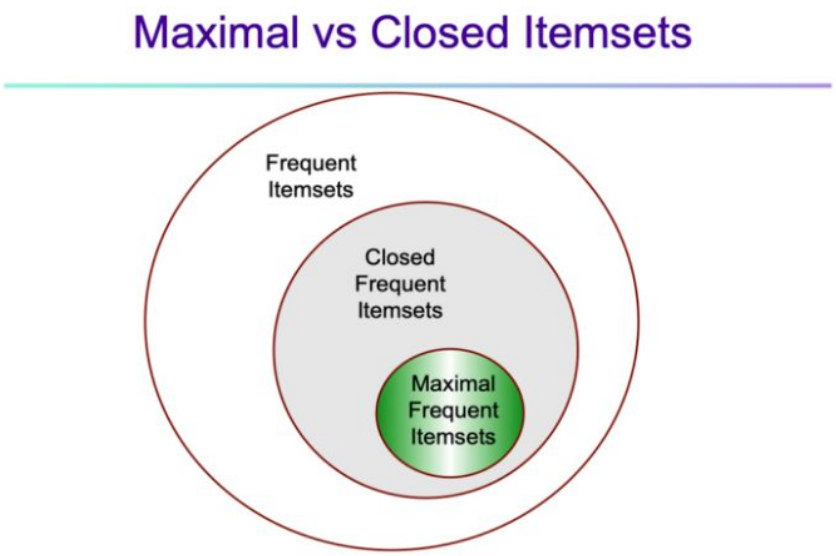
Code:(This can be found in our code in function treeMining)

```
#for finding closed item sets
   max_support_count=0
   for j in range(0,len(all_support_counts)):
     if max_support_count< all_support_counts[j] :
       max_support_count=all_support_counts[j]

   superset_found=False
for super_items,super_support in closed_maximum.items():
     if set(tuple(prefix)).issubset(super_items) and super_support==support_count_subset:
       # print(tuple(prefix),super_items)
       # print(super_support,support_count_subset)
       superset_found=True

   if superset_found==False and support_count_subset>max_support_count  :
     closed_maximum[tuple(prefix)]=support_count_subset

# for finding maximal
       if Conditional_header ==None:
         closed_maximum[tuple(new_frequentset)]=support_count[i]
```

Idea:

Idea behind maximal frequent itemsets if generated Conditional fp tree has header is NULL then the current frequent set before generating tree is maximal

Idea behind closed frequent itemsets storing all superset support counts and if the current support count is greater than superset support count and also checking if the superset of current itemset does not exist in closed frequent items then add it to our closed dictionary

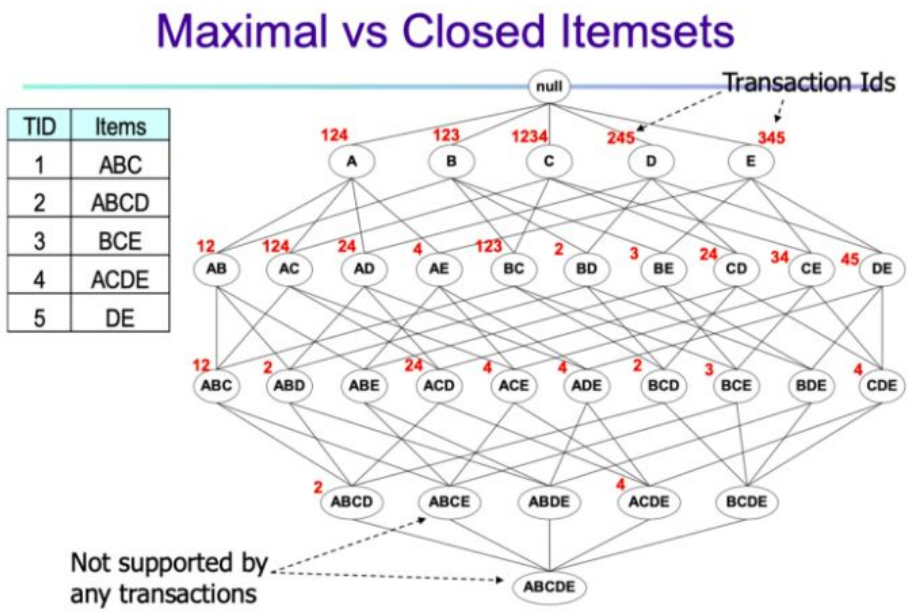


Maximal Frequent itemset is pure subset of closed frequent itemsets

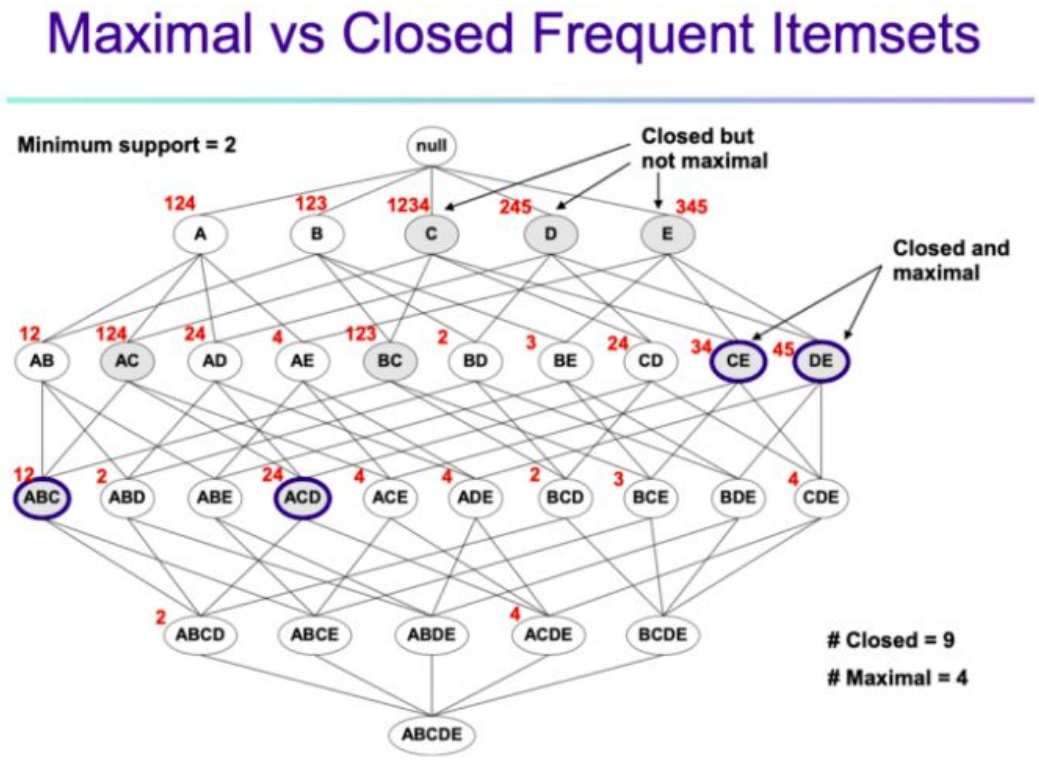
Running my code with this online example:

Enter the minimum support count:
2
Time Taken for operation is:
0.0005061626434326172
All frequent itemsets:
{('B',): 3, ('B', 'C'): 3, ('A',): 3, ('A', 'B'): 2, ('A', 'B', 'C'): 2, ('A', 'D'): 2, ('A', 'D', 'C'): 2, ('A', 'C'): 3, ('D',): 3, ('D', 'C'): 2, ('E',): 3, ('E', 'C'): 2, ('E', 'D'): 2, ('C',): 4}
Length of all frequent itemsets 14
closed item sets
{('B', 'C'): 3, ('A', 'B', 'C'): 2, ('A', 'D', 'C'): 2, ('A', 'C'): 3, ('D',): 3, ('E', 'C'): 2, ('E', 'D'): 2, ('E',): 3, ('C',): 4}
Length of closed item sets :9

Example screenshot:



Example demonstration.



Highlight all Closed and Maximal Frequent Itemsets

These figures are also show the same closed item set answers and the length also matches

Time Analysis of Apriori Implementation

1)File name: BMS1.txt

Support_count: 1192
Number of Frequent Item Sets: 22
Time taken: 3.385481834411621

Support_count: 2980
Number of Frequent Item Sets: 4
Time taken: 0.4075429439544678

Support_count: 3576
Number of Frequent Item Sets: 3
Time taken: 0.422482967376709

2)File name: BMS2.txt

Support_count: 450
Number of Frequent Item Sets: 9
Time taken: 0.5808792114257812

Support_count: 500
Number of Frequent Item Sets: 8
Time taken: 0.5265364646911621

Support_count: 750
Number of Frequent Item Sets: 2
Time taken: 0.20594024658203125

3)File name: leviathan.txt

Support_count: 500
Number of Frequent Item Sets: 853
Time taken: 20.616435766220093

Support_count: 600
Number of Frequent Item Sets: 20
Time taken: 13.61643576622009

Fp Growth analysis:

dataset Name:sign.txt

running for different support count

minimum support count:
500

our optimized code:
Time Taken for operation is:
0.0390625 sec
All frequent itemsets length:
60

python library pyfpgrowth:
Time Taken for operation is:
0.15897154808044434 sec
All frequent itemsets length:
59

minimum support count:
300

our optimized code:
Time Taken for operation is:
2.3756651878356934

All frequent itemsets length:
2500

python library pyfpgrowth:
Time Taken for operation is:
2.7466158866882324

All frequent itemsets length:
2400

Dataset name:fifa.txt

for support count:1800

our optimized code
Time Taken for operation is:
0.04449772834777832
All frequent itemsets length:
8

python library pyfpgrowth
Time Taken for operation is:
0.10732030868530273
All frequent itemsets length:
8

for support count:1700

our optimized code
Time Taken for operation is:
0.051842451095581055
All frequent itemsets length:
60

python library pyfpgrowth
Time Taken for operation is:
0.30566859245300293
All frequent itemsets length:52

Dataset:Levaitan.txt

minimum support count:
600

our optimized code
Time Taken for operation is:
0.022513151168823242
All frequent itemsets length:
20

python library pyfpgrowth
Time Taken for operation is:
0.30566859245300293
All frequent itemsets length:
17

Dataset:msnbc.txt

minimum support count:
2000

our optimized code
Time Taken for operation is:
0.228074312210083
All frequent itemsets length:
59

python library pyfpgrowth
Time Taken for operation is:
0.505689354345300293
All frequent itemsets length:
54

Dataset:bible.txt

minimum support count:
5000

our optimized code:
Time Taken for operation is:
0.4873771667480469
All frequent itemsets length:57
python library pyfpgrowth:
Time Taken for operation is:
1.5893771347234590
All frequent itemsets length:
50

We tried with various data size, data distribution, minimal support threshold setting, size of desired frequent itemsets
As per the above analysis we can come to a conclusion that normal apriori is slower than optimized apriori and fp growth python optimization (merging strategy) works faster than normal fp growth implementation.In between optimized apriori and optimized fp growth ,optimized fp growth is faster than optimized apriori

Reasoning For above conclusion:

Fp Growth(merging strategy):It is faster because instead of going from node to root for all the header frequent items always here we are repeating our upward traversal path always for all items in the .In optimized we just maintain a dictionary by just traversing whole Conditional fp tree once .So optimized Fp Growth is faster.