Frequent Itemset Mining Project

DATA ANALYTICS - I

28-10-20

Anishka Sachdeva (2018101112)

Satyam Viksit Pansari (2018101088)

Overview

Implementation of Apriori and FP Growth algorithms to generate frequent itemsets and do a comparative study about them.

About the Project

- Language used : C++
- Inputs required for both the Algorithms :
    - Value of Minimum Support (int)
    - Name of the dataset text file as "<FileName>.txt"

  Note : The text file should be present in the same directory as the codes.
- Apriori Code File Name : 2018101112_2018101088_apriori.cpp
- FP-Growth Code File Name : 2018101112_2018101088_fpg.cpp
- Expected Output : Frequent Itemsets of different sizes for different strategies used to implement a particular algorithm. Output also displays the time taken for the algorithm to find the frequent itemsets.

Note : Time calculations **does not** include the following :

- Formation of transactional dataset from the .txt file.
- Printing of frequent itemsets on the terminal.

Preprocessing of Dataset

1. The items in each transaction in a dataset were separated by -1.
2. Each transaction ended with -2.
3. Thus, the .txt file was preprocessed handling the above two cases to finally get the data in the form of a transactional database (in a particular data structure spending on the algorithms implemented) consisting of transactions only containing items (without -1 or -2).

Goals

Task 1 - Implementation of Apriori Algorithm

Original Apriori Algorithm

- ***About :***
  Apriori Algorithm is an iterative approach or level-wise search where k-frequent itemsets are used to find k+1 itemsets. This can be explained as : First, the set of frequent 1-itemsets is found by scanning the database to accumulate the count for each item and collecting those items that satisfy minimum support. The resulting set is denoted by L1. Next, L1 is used to find L2, the set of frequent 2-itemsets, which is used to find L3, and so on, until no more frequent k-itemsets can be found (by using the intermediate Candidate or Ck sets generated).

  To improve the efficiency of level-wise generation of frequent itemsets, an important property is used called *Apriori property* which helps by reducing the search space.

  Apriori Property –

  All non-empty subsets of frequent itemsets must be frequent. The key concept of Apriori algorithm is its anti-monotonicity of support measure. Apriori assumes that :

  All subsets of a frequent itemset must be frequent(Apriori property).

If an itemset is infrequent, all its supersets will be infrequent.

- ***Implementation :***
    1. We store the transactional database in the form of a vector of vectors where each vector is a transaction.

    2. We next call the “run_apriori” function which deals with the generation of Candidate Itemsets (Ck) and Frequent Itemsets (Lk). Both the Candidate Itemsets and Frequent Itemsets are a vector of vectors.

       Since the Apriori Algorithm is an iterative algorithm, the generation of Ck and Lk is done iteratively. This is shown in the below code snippet :

```
while(1)
{
    process_next_candidates(Candidates_k,L_k, dataset,items_size, type);
    items_size++;
    if(Candidates_k.size() == 0)
        break;

    generate_frequent_itemsets(L_k,dataset, Candidates_k, transaction_reduction_flag, minsup, type);

    if(L_k.size() > 0)
        frequent_itemsets_mined.push_back(L_k);

}
```

    3. The run_apriori function begins with generating the “next candidate sets” using the previous frequent itemsets i.e we generate C(k) using L(k-1) by merging elements of L(k-1) and then pruning .Then generate_frequent_itemsets generates L(k) fromC(k) by traversing the database once .

       Where L(k)=set of k-frequent itemsets and,C(k)=candidates for k-frequent item set

    4. Now depending upon the item_set size :
        a. If the item set size is equal to 1, then we simply traverse the dataset and store all the singletons in a set. Then the singletons are put into the Candidate Itemsets vector.
        b. If the itemset size >= 2, then, merging is applied to the itemsets present in Lk to generate Ck+1. Merging/joining of the itemsets

is done on the following rule : All those itemsets are merged who have the same k-1 length items in them.

Below code snippet does the **merging of itemsets** :

```
for(int i = 0; i < L_k.size(); i++)
{
    vector<int>v1 = L_k[i];
    int last_item_v1 = v1[v1.size()-1];
    v1.pop_back();
    for(int j = i + 1; j < L_k.size(); j++)
    {
        vector<int>v2 = L_k[j];
        int last_item_v2 = v2[v2.size()-1];
        v2.pop_back();
        if(v1 == v2)
        {
            int small = min(last_item_v1, last_item_v2);
            int large = max(last_item_v1, last_item_v2);
            v2.push_back(small);
            v2.push_back(large);
            Candidates_k.push_back(v2);
        }
        else
            break;
    }
}
```

Now, if an immediate subset from a merged itemset is not in Lk, then it cannot be in Ck+1. And thus it is pruned/removed.

Below code snippet does the **pruning of the itemsets** :

```
set<vector<int> >L_k_set;
set<vector<int> >::iterator itr;
for(int i = 0; i < L_k.size(); i++)
{
    L_k_set.insert(L_k[i]);
}

vector<vector<int> >pruned;
bool is_present = false;
for(int i = 0; i < Candidates_k.size(); i++)
{
    is_present = false;
    for(int j = 0; j < Candidates_k[i].size(); j++)
    {
        vector<int>merged_item =Candidates_k[i];
        merged_item.erase(merged_item.begin() + j);
        if(L_k_set.find(merged_item) == L_k_set.end())
        {
            is_present = true;
            break;
        }
    }
    if(is_present == false)
        pruned.push_back(Candidates_k[i]);
}
Candidates_k = pruned;
```

5. Once, the Ck+1 set is generated, we generate the Lk+1. We traverse the database and calculate the frequency of itemsets in the transactions of the dataset. The itemsets having frequency >= minsup become Lk+1.

6. The above iterative algorithm keeps running until the size of Candidates vector becomes zero which essentially means no candidate itemsets are left to find frequent itemsets of further lengths.

The following optimizations were implemented :

1. Hash Based Technique

- ***About :***

  As we know that apriori algorithm has some weakness so to reduce the span of the hopeful k-item sets, C_k hashing technique is used.

Our hash based Apriori execution, utilizes the data structure that specifically speaks to a hash table. Specifically the 2-itemsets, since that is the way to enhancing execution. It simply deals with hashing itemsets into corresponding buckets. A hash-based technique can be used to reduce the size of the candidate k-itemsets, C_k, for k > 1. For example, when scanning each transaction in the database to generate the frequent 1-itemsets, L_1 ,, we can generate all the 2-itemsets for each transaction, hash (i.e., map) them into the different buckets of a hash table structure, and increase the corresponding bucket counts. A 2-itemset with a corresponding bucket count in the hash table that is below the support threshold cannot be frequent and thus should be removed from the candidate set. Such a hash-based technique may substantially reduce the number of candidate k-itemsets examined (especially when k = 2).

- ***Implementation :***
  - **Data Structure** : We used an ordered map to construct the hash table where :

```
map<pair<int,int>,int>count_c2
```

    - Key - Pair consisting of the 2-items (Example : (I2,I3))
    - Value - Stores the frequency of a particular pair occurring as a combination in different transactions in the database.
  - **Hashing strategy** :
    - The normal apriori is modified by making a Hash for all the sets of 2 sized Itemsets possible.
    - The hashing was done using a standard library ordered map of <pair<int,int>,int>.
    - Hence there was no step to generate 2-itemset candidates from frequent 1-itemsets .
    - In the first pass of the database itself we counted all the 1-items and for every transaction generated its all possible pairs and increment the count of each pair in the hash table.

2. Transaction Reduction

- ***About :***

A transaction that does not contain any frequent k-itemsets cannot contain any frequent (k + 1)-itemsets. Therefore, such a transaction can be marked or removed from further consideration because subsequent database scans for j-itemsets, where j > k, will not need to consider such a transaction.

- ***Implementation :***

  We tried implementing this optimization in 2 ways :

  1. **Boolean Vector**
     - **Data Structure** : A vector of the size equal to the number of transactions where indices basically represent transaction IDs.

```
vector<bool>transaction_reduction_flag(dataset.size(), true);
```

       Steps : The normal apriori algorithm is implemented along with the following additions/modifications :

       1. A boolean vector is initialised with all the values as “true” in the starting depicting that all the transactions are potential transactions of generating frequent itemsets. The values true or false act as flag values for a transaction.
       2. On generating the candidates set, if none of the itemsets present in the candidates occur in a particular transaction, then that particular transaction won’t be useful in the further computations (as stated by the Transaction Reduction Optimization). And thus, we mark that transaction id as false (depicting that it is not useful any further).
       3. Now, whenever we have to check for frequent itemsets in the entire database, we traverse only those transactions which have true as their flag value.

     - **Time Complexity :** O(1) for checking whether a transaction id is true or false.
     - **Space Complexity :** O(N) where N = number of transactions in the entire database.

```
for(int t_id = 0; t_id < dataset.size(); t_id++)
{
    bool is_transaction_useful = false;
    if(transaction_reduction_flag[t_id] == true)
    {
        if(dataset[t_id].size() < Candidates_k[0].size())
            continue;

        for(int c_id = 0; c_id < Candidates_k.size(); c_id++)
        {
            int count = 0;
            for(int c_element = 0; c_element < Candidates_k[0].size(); c_element++)
            {
                if(dataset[t_id].find(Candidates_k[c_id][c_element]) != dataset[t_id].end())
                {
                    count++;
                }
                else
                {
                    break;
                }
            }
            if(count == Candidates_k[0].size())
            {
                is_transaction_useful = true;
                support_count[c_id] += 1;
            }
        }
        if(is_transaction_useful == false)
        {
            transaction_reduction_flag[t_id] = false;
        }
    }
}
```

2. **Multiset of sets**
    - **Data Structure :** Multiset of sets

      Steps : The normal apriori algorithm is implemented along with the following additions/modifications :

      1. In the original Apriori algorithm implementation, the database was in the form of a vector of sets where each set contains items in a transaction. For this implementation, we made the database as a multiset of sets where each set contains items in a transaction.
      2. On generating the candidates set, if none of the itemsets present in the candidates occur in a particular transaction, then that particular transaction won't be useful in the further computations (as stated by the Transaction

Reduction Optimization). And thus, we simply remove that particular transaction set from the multiset.

3. Now whenever we have to check for frequent itemsets in the database, we will not have useless transactions at all and thus the complexity of scanning the database decreases (because length of the database decreases as transactions get removed).

There was not much difference in time taken to generate the frequent itemsets in fact the multiset method was never faster . And hence we finally, we went on with the first method.

Task 2 - Implementation of FP Growth Algorithm

Original FP-Growth Algorithm

1. ***About :***

   The two primary drawbacks of the Apriori Algorithm are:-

   1. At each step, candidate sets have to be built.
   2. To build the candidate sets, the algorithm has to repeatedly scan the database.

   These two properties inevitably make the algorithm slower. To overcome these redundant steps, a new association-rule mining algorithm was developed named Frequent Pattern Growth Algorithm. It overcomes the disadvantages of the Apriori algorithm by storing all the transactions in a Trie Data Structure. FP growth algorithm represents the database in the form of a tree called a frequent pattern tree or FP tree.

   This tree structure will maintain the association between the itemsets. The database is fragmented using one frequent item. This fragmented part is called “pattern fragment”. The itemsets of these fragmented patterns are analyzed. Thus with this method, the search for frequent itemsets is reduced comparatively. The root node represents null while the lower nodes represent the itemsets. The association of the nodes with the lower nodes

that is the itemsets with the other itemsets are maintained while forming the tree.

2. ***Implementation :***

Steps :

1. We have a struct representing each node of the FP_TREE :

```
struct FP_TreeNode {

    int count;
    int node_value;
    FP_TreeNode* parent;
    map<int, FP_TreeNode*>children;
    bool marked;

};
```

- **Count** : Count of an item in the FP_Tree
- **Node_Value** : Stores the item's numeric value
- **FP_TreeNode* parent** : Stores the address of the parent of a node.
- **map<int, FP_TreeNode*>children** : Hashmap to store all children of a node (points to all the children of the given node). We allocate memory for items in use.
- **bool marked** : will be used for merging strategy optimization as a marker to find out whether a node is being visited for the first time while generating a conditional pattern base.

  Space used here with every node here is proportional to number of children.

2. For building the dataset, the transactions are stored in the form of vector of pairs where :

```
vector<pair<set<int>,int> >dataset;
```

- First element of the pair is a set of integers that stores the elements in a transaction.
- Second element of pair is an int value that stores the count of the transactions as in case of conditional pattern bases for the original data case we just provide every transaction count=1.

3. We create an empty vector Alpha (which denotes NULL value of Alpha in the beginning of the algorithm).

4. Now we scan the initial database for the first time and store the frequencies of singleton items in a map.

```
map<int,int>order_map;
```

   - Key of the map - Singleton
   - Hash Value - Frequency of Singleton

5. Now we made a vector of pair to store the frequent items where :

```
vector<pair<int,int> >order;
```

   - First element of a pair is an integer representing the frequency of the frequent item.

   - Second element of a pair is an integer representing the item itself.

6. We sort the order vector in **Descending order** (Descending order is our chosen order for inserting the transactions in the FP Tree).

7. We again have a map for storing the position of items in the sorted order to be used efficiently later in the implementation.

```
map<int,int>elementPosition_in_order;
```

8. Now we create the FP_Tree which has a Trie type of structure. We traverse the dataset transaction by transaction. Now we traverse the order vector and check if any element in the order vector is present in the particular transaction, then that is inserted in the FP_TREE else skipped.

   The following code snippet creates the required FP Tree.

```
FP_TreeNode* root = nullptr;
if(root == nullptr)
    root = create_TreeNode(nullptr);

FP_TreeNode* crawler =  root;

set<int>::iterator itr;
for(int t_id = 0; t_id < dataset.size(); t_id++)
{

    set<int>transaction = dataset[t_id].first;
    int cnt = dataset[t_id].second;

    crawler = root;
    for(int i = 0; i < order.size(); i++)
    {
        if(transaction.find(order[i].second) != transaction.end())
        {

            if(crawler->children.find(order[i].second) == crawler->children.end())
            {
                FP_TreeNode *temp = create_TreeNode(crawler);
                temp->node_value = order[i].second;
                crawler->children[order[i].second] = temp;
                header_table[elementPosition_in_order[(order[i].second)]].insert(temp);
            }

            crawler->children[order[i].second]->count += cnt;
            crawler = crawler->children[order[i].second];
        }
    }
}
```

The below snippet creates a particular node of the FP_Tree :

```
FP_TreeNode* create_TreeNode(FP_TreeNode* parent_pointer)
{
    FP_TreeNode* newNode = new FP_TreeNode;
    newNode->count = 0;
    newNode->node_value = -1;
    newNode->parent = parent_pointer;
    newNode->marked=false;
    return newNode;
}
```

9. While creating the FP_Tree, we store the frequency of a particular item, parent pointer and value of the particular node.

10. Once the FP_Tree is built, the process of FP_Tree mining and frequent itemset generation begins. If FP_Tree is a null tree we can simply

return from the function. Otherwise we check whether the tree is a multi branch tree or a single branch tree.

a. If the tree is a single branch tree, then we create a vector to store all the node values (item values) occurring in the single branch. We make all possible subsets of the items present in the single branch factor and append the alpha vector to each of them . The resultants are frequent itemsets. And then return from the recursive function.

   All possible subsets are calculated by the following method :

```
string findBinary(int n)
{
    string final = "";
    while(n > 0)
    {
        int t = n%2;
        if(t == 0)
            final += "0";
        else
            final += "1";

        n = n>>2;
    }
    return final;
}
```

   A “1” in the string represents the presence of that item in the subset.

b. If the tree has multiple branches, then we have to create a conditional pattern-base for each item present in the header table. For constructing the Conditional pattern base, the lowest node(last item in the order determined while constructing the tree) is examined first along with the links of the lowest nodes (stored in the header table). The lowest node represents the frequency pattern length 1. From this, traverse the path in the FP Tree. This path or paths are called a conditional pattern base.

Conditional pattern base is a sub-database consisting of prefix paths in the FP tree occurring with the lowest node (suffix).

11. We create a Beta Vector which is formed by :

    Alpha Vector **union** Header_Table elements

Beta vector becomes a frequent itemset.

```
vector<int>beta;

for(int j = 0; j < alpha.size(); j++)
{
    beta.push_back(alpha[j]);
}
beta.push_back(order[i].second);
frequents_global.push_back(beta);
starter2(dataset_arr[i],minsup,beta); //RECURSION
```

12. Then the function is recursively called on new dataset and beta values (where beta vector will become alpha for the next call).

```
starter2(dataset_arr[i],minsup,beta); //RECURSION
```

The following optimizations were implemented :

1. Merging Strategy Technique

- ***About*** **:**

  It is time and space consuming to generate numerous conditional pattern bases during pattern-growth mining. An interesting alternative is to push right the branches that have been mined for a particular item p, that is, to push them to the remaining branch(es) of the FP-tree. This is done so that fewer conditional pattern bases have to be generated and additional sharing can be explored when mining the remaining FP-tree branches.

- ***Implementation*** **:**

  Every path in the tree was being iterated multiple times to generate the conditional transactions for conditional pattern bases for all the elements along the path. For instance if we consider a path root->a->b->c->d->e it will be traversed 5 times (for e, then d, then c, then b, then a)

  The basic idea we used was to stop this redundant traversal. So when the path is first traversed for say ‘e’, we generate the conditional transaction for all the other nodes along the path here itself and add them in their respective conditional pattern bases.

  Steps :

1. Check if the node is marked. If it is marked then we don't need to traverse its path to the root. If unmarked get traverse its path till root along with the address of the nodes.
2. For all the nodes in this path that are unmarked, the conditional transaction will be a subset of the path which we extract and then mark the node and add this subset in the conditional pattern base of the respective element.

This takes care of the redundancy which is clearly reflected in the results as the time taken to generate frequent itemsets reduces but not by a great extent as the tradeoff is not that huge .

We tried deleting the nodes instead of marking them but it turned out to be costlier as the time taken increased so we stood with the marking the nodes version .

Task 3 - Comparisons of different strategies

Purple : Apriori

Yellow : FP-Growth

Green : Total Itemsets generated

We have taken 3 values of minimum support as : **0.5%, 0.75% and 1%** of the entire dataset size.

Dataset Link	**Number of Transactions**	**Minimum support count**	**Time taken by original algorithm** **Apriori**	**Time taken by Transaction Reduction** **Apriori**	**Time taken by Hash Based Technique** **Apriori**	**Time taken by optimized algorithm** **Apriori**	**Time taken by original algorithm** **FP-G**	**Time taken by optimized algorithm (Merging)** **FP-G**	**Total Itemsets**
http://www.	59601	298	37.005	36.201	1.159	1.162	1.485	1.433	201

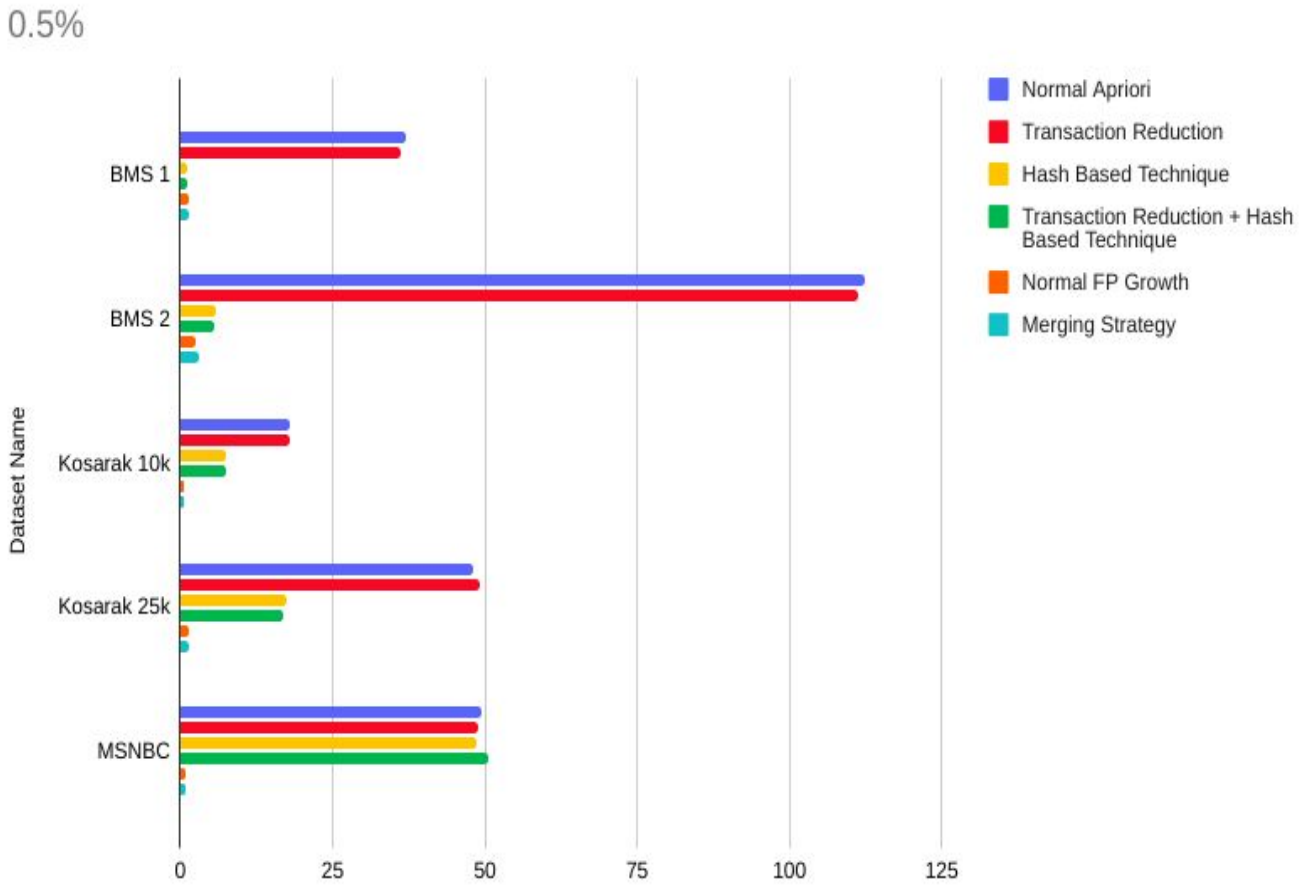
philippe-fournier-viger.com/spmf/datasets/BMS1_spmf		447 596	17.079 7.342	17.284 7.322	1.093 1.073	1.091 1.074	1.160 0.757	1.084 0.746	123 78
http://www.philippe-fournier-viger.com/spmf/datasets/BMS2.txt	77512	387 581 775	112.469 31.262 11.022	111.288 31.638 11.426	5.815 4.309 4.468	5.416 4.988 4.579	2.605 1.711 1.117	2.921 1.745 1.111	411 160 82
http://www.philippe-fournier-viger.com/spmf/datasets/kosarak10k.txt	10000	50 75 100	17.840 5.409 2.329	17.975 5.168 2.264	7.454 6.108 5.270	7.482 6.118 5.202	0.582 0.359 0.311	0.544 0.346 0.275	1716 699 392
http://www.philippe-fournier-viger.com/spmf/datasets/kosarak25k.txt	25000	125 187 250	47.960 13.492 5.709	49.020 12.812 5.390	17.469 15.166 13.027	16.925 13.741 12.726	1.373 0.823 0.621	1.306 0.805 0.608	1668 701 399
http://www.philippe-fournier-viger.com/spmf/datasets/BIBLE.txt	36969	185 277 370	Took a lot of time Took a lot of time 421.260	Took a lot of time Took a lot of time	Took a lot of time Took a lot of time	Took a lot of time Took a lot of time	18.666 13.5141 10.6722	20.361 14.714 11.352	43363 19919 11448
http://www.philippe-fou	31790	158	49.365	48.969	48.515	50.670	0.720	0.778	9754

rnier-viger.com/spmf/datasets/MSNBC.txt		238 317	39.167 32.724	45.261 36.559	44.589 33.477	42.516 33.125	0.679 0.629	0.720 0.648	7092 5669

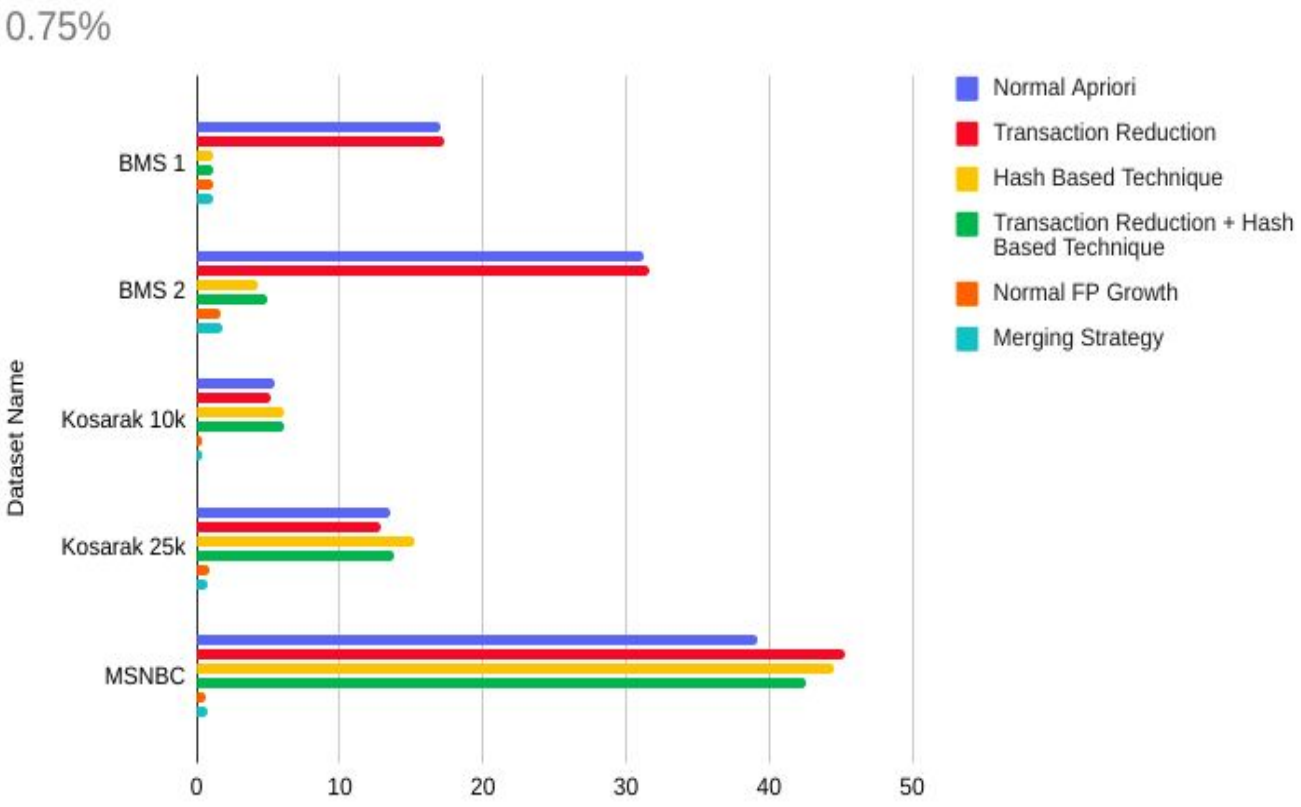
Graphical Comparative Study for all the different strategies:

1. Bar Charts :

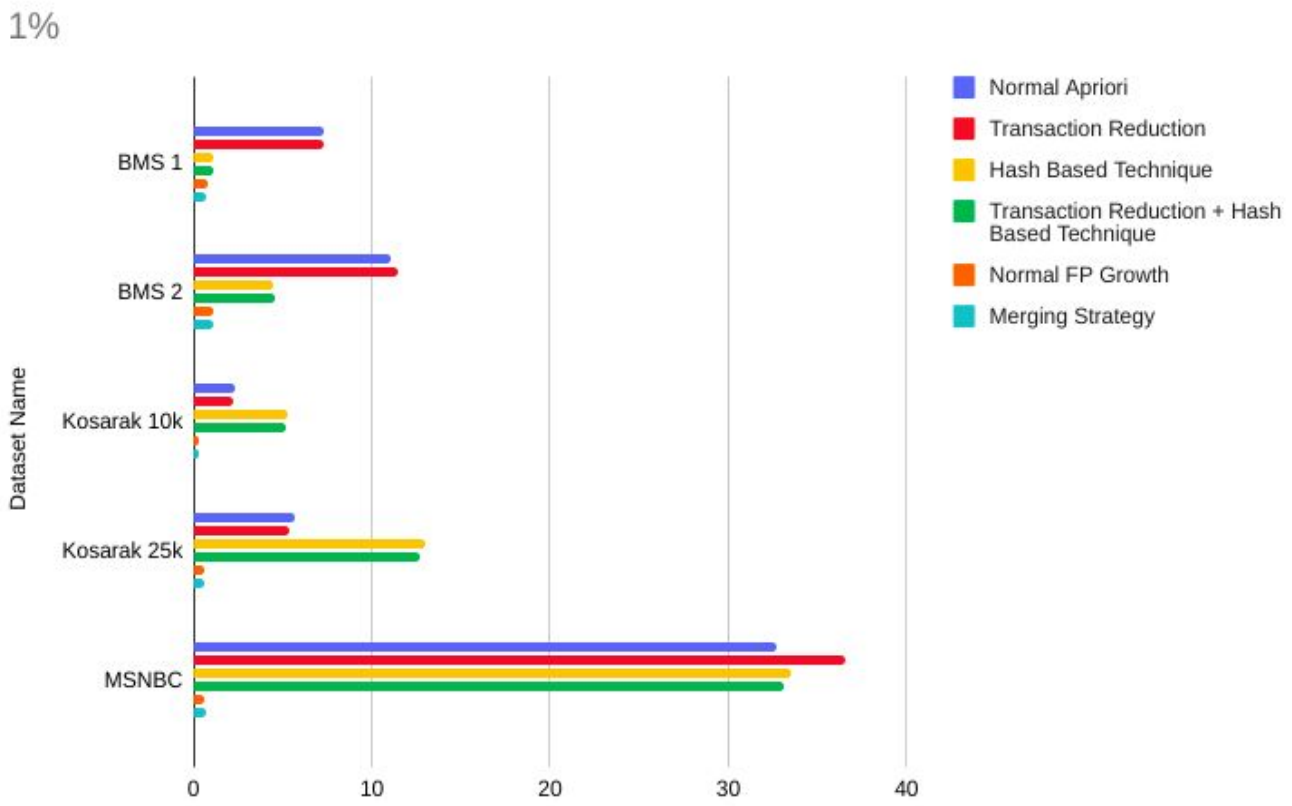
- **For 0.5% as minimum support :**



- **For 0.75% as minimum support :**

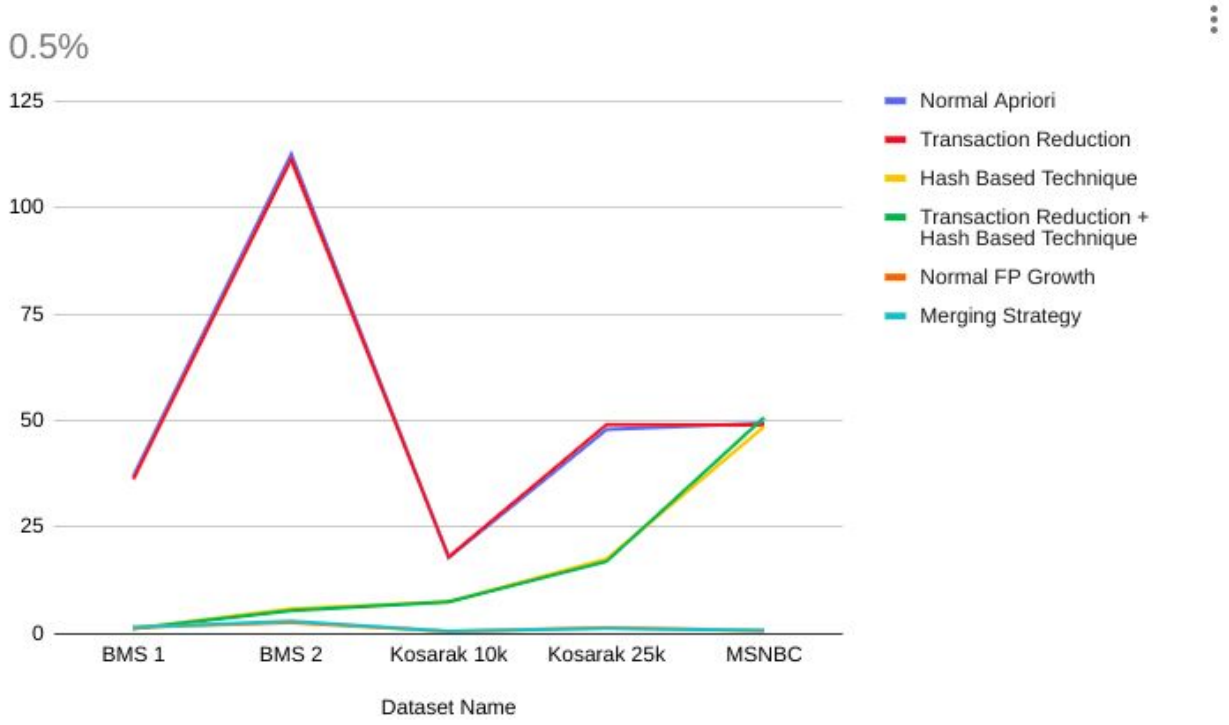


- **For 1% as minimum support :**

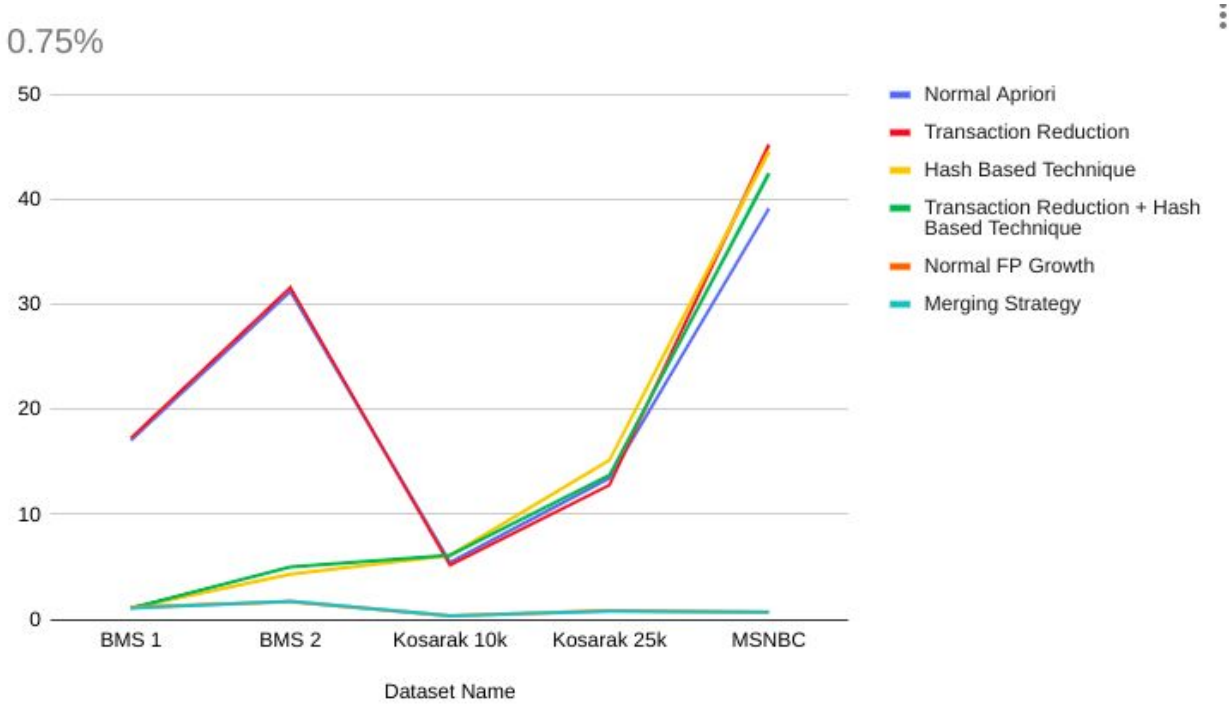


2. Line Charts :

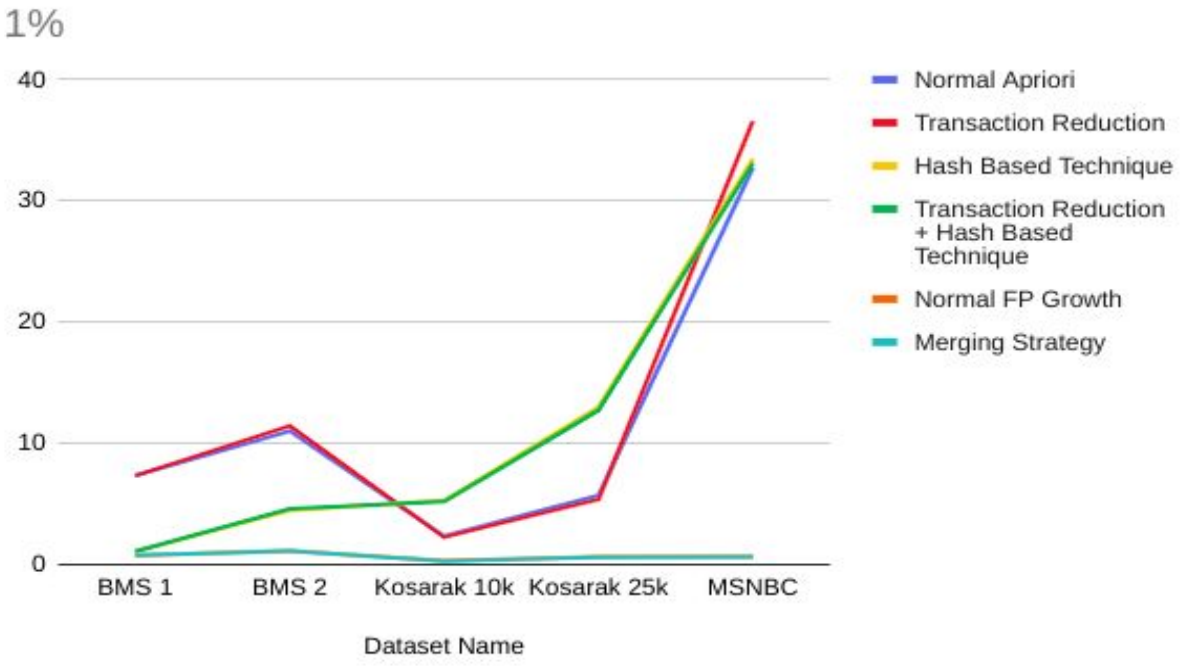
- **For 0.5% as minimum support :**



- **For 0.75% as minimum support :**



- **For 1 % as minimum support :**



We can see how 2-2 lines (**Normal Apriori--Transaction Reduction**; **Hash Based Technique--Transaction Reduction + Hash Based Technique; Normal FPG -- Merging Strategy**) have merged i.e. 3 almost identical pairs of algorithms can be seen. We can see that the shape of the line chart changed as we changed minsup that implies that things depend much more on the database than just minsup. Further analysis gave the following observations .

Observations :

- All the data points to the fact that FP Growth is significantly better than Apriori and the only times when Apriori came close to FP Growth is when the output only had a maximum of frequent ⅔-itemsets and in that also Apriori with hashmap was comparable and not the normal Apriori or Apriori with transaction reduction . Basically when the dataset was iterated once or twice in Apriori. So we concluded that it is the fewer database scans that make FP Growth so much better than Apriori.
- As min support count is increased, execution time decreases greatly. It is obvious as the number of iterations reduces due to fewer itemsets in the final answer.

- In one particular dataset (Bible) Apriori was taking a lot of time so we had to stop it and even FP growth took much greater time w.r.t all the other datasets . The reason for this we think is that it had a much greater item count and output compared to any other dataset around 43000 for minsup=0.5%. Which was even more than the number of transactions in the input which was not observed in any other dataset. So we understand how important item count is .
- Transaction reduction reduced time mostly in cases where the final answer contained very few itemsets . The reason for this we think is that transaction reduction only makes sense when a significant number of transactions are discarded else it is just over head to check for every transaction if only a few end up being discarded .
- In any scenario among the apriori algorithms the hash based technique always gave a significant reduction in time . The reason for this is that in apriori the most expensive part is creating candidates and checking for the 2-itemsets as all combinations of 1-itemsets are eligible candidates for 2-itemsets and thus no pruning happens at this stage . Hash based technique not only reduced one entire traversal of transactions (which is the most expensive part as the dataset has to be loaded from disk) but also gave both 1 and 2-frequent itemsets in a single traversal os transactions.
- Similarly in cases where the database (with given minsup) lacked frequent 1 or 2-Itemsets Apriori with hash based technique and Apriori without it took almost the same time .
- In cases with low itemcount and lower number of frequent itemsets in the output , both the algorithms did really well .Hence we understood that time required not only depends on how many transactions are there in the input dataset in fact it depends much more on the item count and expected output and that is why a dataset containing 77512 transactions took much less time than the one with 36969.
- As we increased the min support in Apriori algorithms the normal Apriori took lesser and lesser time while the difference in case of Hash based was not that steep and eventually at a high enough minsup the time almost became comparable . The reason for this is that hash bases only optimises

the algo to find 1&2-frequent itemsets faster after that it's all the same . Also when minsup is high then 1-frequent and 2-frequent itemsets will also decrease and thus Hash map technique starts to lose its advantage over Apriori without it.

- FP Growth without merging and with merging didn't show much difference though FP Growth with merging was almost always slightly better. The reason for this is that we can expect a significant improvement only in cases where dense and deep trees are formed and the overhead can be dearly compensated and there were no such cases . Hence in cases where the a lot of shallow trees with a lot of branches are formed will result in merging taking more time than the original for instance in case of Bible with minsup 0.5% out of 43363 frequent itemset generated around 39000 were till 5 itemsets only (item count for bible was the highest) and Bible itself had 36969 transactions hence dense and shallow trees were generated and time taken by FP Growth with merging increased by around 2 sec.
- The runtime of the various algorithms heavily depend upon the factor on how frequently the same dataset text file is being used to run the algorithms i.e if the dataset text file is being used which had been used in the recent past to run the algorithm, we observe a decrease in the runtime. This happens because when the text file is used for the first time, it takes maximum time because it has to be brought from the hard disk to the RAM, thus increasing the I/O operation. Whereas when the same text file is run after getting executed previously, the same file is present in the Cache, thus decreasing the I/O operation.

Conclusion

According to us, **FP - Growth with Merging Strategy** works the best and thus, can be the best choice for an unseen dataset.
