Frequent Itemset Mining Report

Submitted by -

Nikunj Nawal (2018111011)
Manas Kabre (2018111014)

Chosen Datasets -

- BMSWebView1 (BMS1_spmf)
- BMSWebView2 (BMS2)
- Bible

	Transaction Count	Item Count	Avg Seq Length
BMS - 1	59,601	497	2.42
BMS - 2	77,512	3,340	4.62
BIBLE	36,369	13,905	21.6

BIBLE -> This dataset is a dense dataset as it has large sequences and hence these can correspond to larger order frequent itemsets -> expected to take longer time
BMS - 1 -> This dataset has small Seq length and hence will not generate higher order Frequent itemsets and hence is expected to take lesser time
BMS -2 -> This dataset has medium Seq length and has a high number of entries. No prediction for this dataset.

Note - Only those dataset were chosen which have item representation in Integral form.

Pre-Processing Of Data -

- From the Data it was observed that -1 was used as the separation delimiter between two items and -2 was used for depicting the end of a transaction.
- Some transactions were having the same item entry multiple times hence set which were removed (used set for storing a transaction).

Input Format in File -

- Input the File path as a single string on which you want to run the algorithm.
- Input the percent of number of transactions of the File on which you will be running the codes for the Min Support Count Percentage.

Apriori -

Standard Notations:

L_k - Set of all Frequent Itemset of Size k which satisfy minimum support.
C_k - To find L_k , a set of candidate k-itemsets is generated by joining L_{k-1} with itself. This set of candidates is denoted C_k

About The Algorithm:

- Apriori employs an iterative approach known as a level-wise search, where k-itemsets are used to explore (k + 1)-itemsets.
- The Apriori algorithm is based on *The Apriori Property* which states that Any (k−1)-itemset that is not frequent cannot be a subset of a frequent k-itemset.(here k is the size of the itemset).
- Two steps are involved in Apriori Algorithm Implementation
    1. *The Join Step* - The join, $L_{k-1} \bowtie L_{k-1}$, is performed, where members of L_{k-1} are joinable if their first (k − 2) items are in common. The resulting itemset formed by joining l_1 and l_2 is $\{l_1[1], l_1[2], \ldots, l_1[k-2], l_1[k-1], l_2[k-1]\}$ where l_1 and l_2 are in L_{k-1} and are joinable and $l_1[k-1]$ is not the same as $l_2[k-1]$.
    2. *The Prune Step* - if any (k − 1)-subset of a candidate k-itemset is not in L_{k-1} , then the candidate cannot be frequent either and so can be removed from C_k .

Implementations:

- File name - 2018111011_2018111014_apriori.cpp
- ll is for long long int.

- vector < set<ll> > v is the vector containing all the transactions where each transaction is in the form of a set.
- sup is the Minimum Support Count required to qualify for Frequent Itemset.
- All functions mentioned below will return a vector of all possible frequent itemsets of all sizes.

1. **Basic Apriori** -
    - The algorithm is implemented as the function `apriori(vector <set<ll> > v, ll sup)` in the code.
    - This function was implemented by referring to the given pseudo-code in the Textbook.
    - Some Basic Tweaks were-
        1. Items in Itemsets in L_k and C_k are sorted Lexicographically which helped in fastening the join step.
        2. Not Checking for the Transactions with size lesser than that of the Itemset while counting for the support count.
        3. Set from standard libraries in cpp was used to store each Transactions for faster item search in a transaction. Set was also used for fastening the check in of immediate subsets in the pruning step to make it faster.
        4. Map from standard libraries in cpp was for generating the all the frequent itemsets of size 1(L_1). (Map was of {ll,ll} format)

2. **Hashed Apriori** -
    - The algorithm is implemented as the function `hashed_apriori(vector <set<ll> > v, ll sup)` in the code.
    - The function of apriori(basic) mentioned above is modified by making a Hash for all the sets of 2 sized Itemsets possible. The hashing was done using a standard library map of {pair,ll}.
    - Again the transactions having only 1 item in it were not used for hashing of 2 sized itemsets for making the algorithm more faster.
    - The formation hash of all 2 sized itemsets was made in the same data set iteration while generation of L_1.

3. **Transaction Reduction** -

- The algorithm is implemented as the function
  `transaction_reduction(vector <set<ll> > v, ll sup)`
  in the code.
- The function of apriori(basic) mentioned above is modified by removing those transactions which do not help in increasing the support count of any itemset in pruned C_k for the rest of the algorithm. This was done by checking if a transaction helped in increasing the support count of the transaction while counting for the support count of all itemsets remaining in pruned C_k.
- If some transaction helped in increasing the support count of any itemsets of size k and the transaction's size itself is also equal to k then after checking for all the itemsets of C_k for that transaction , that transaction was also removed for the rest of the algorithm for making it more faster.

4. **Both Hashed and Transaction Reduction** -
   - The algorithm is implemented as the function
     `reduction_hashed_apriori(vector <set<ll> > v, ll sup)`
     in the code.
   - The function of apriori(basic) mentioned above is modified by applying both hashed and transaction reduction optimization with all the small tweaks mentioned in their respective segments as well.
   - All transactions of size 1 and 2 were removed after hashing all the itemsets of size 1 and 2.

FP - Growth

File Name - 2018111011_2018111014_fpg.cpp

Struct node :

- Represents each node of the tree
- Has the following elements :

```
struct node
{
    unordered_map<int,node*> pointer;
    int freq;
    int item;
    node* parent;
};
```

- Pointer is a hash-map which points to all the children of the given node
- Freq maintains the count of the node traversed as required in FP Growth
- Item stores the item number/name of the item
- Parent points to the parent of the given node.

FP_Growth :

- Function to build the FP-Tree

```
FP_Growth(vector<vector<int>> transactions, int min_sup, vector<int>
alpha);
```

- Transactions is the set of transactions on which the FP tree is built, with support count value as min_sup and alpha as the vector alpha.
- The function has 4 parts :
    - Sorting the transaction based on their occurrence and removing the transaction having that value below min_sup
    - Build Tree - Building the FP Tree using the transactions which is received as a parameter

- Check for single path - If the tree has a single path, then the function returns after pushing all the combinations of the nodes of this single path into the final answer
- Call FP_Mine : If the tree does not have a single path, then the FP_Mine function is called by passing in the required parameters

FP_Mine :

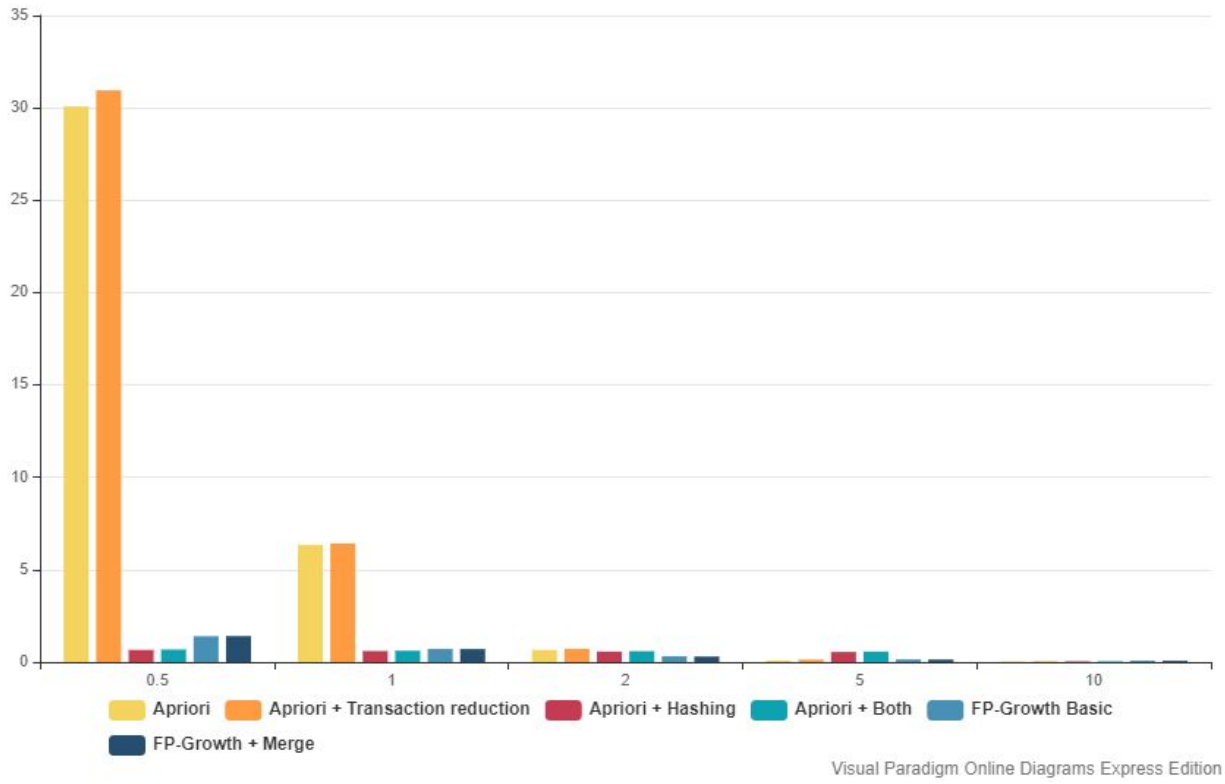
- Function to mine the frequent item sets.
- Generates a new set of sub-transactions based on the algorithm and then calls FP_Growth function recursively with the new subset of transactions.

Optimisations :

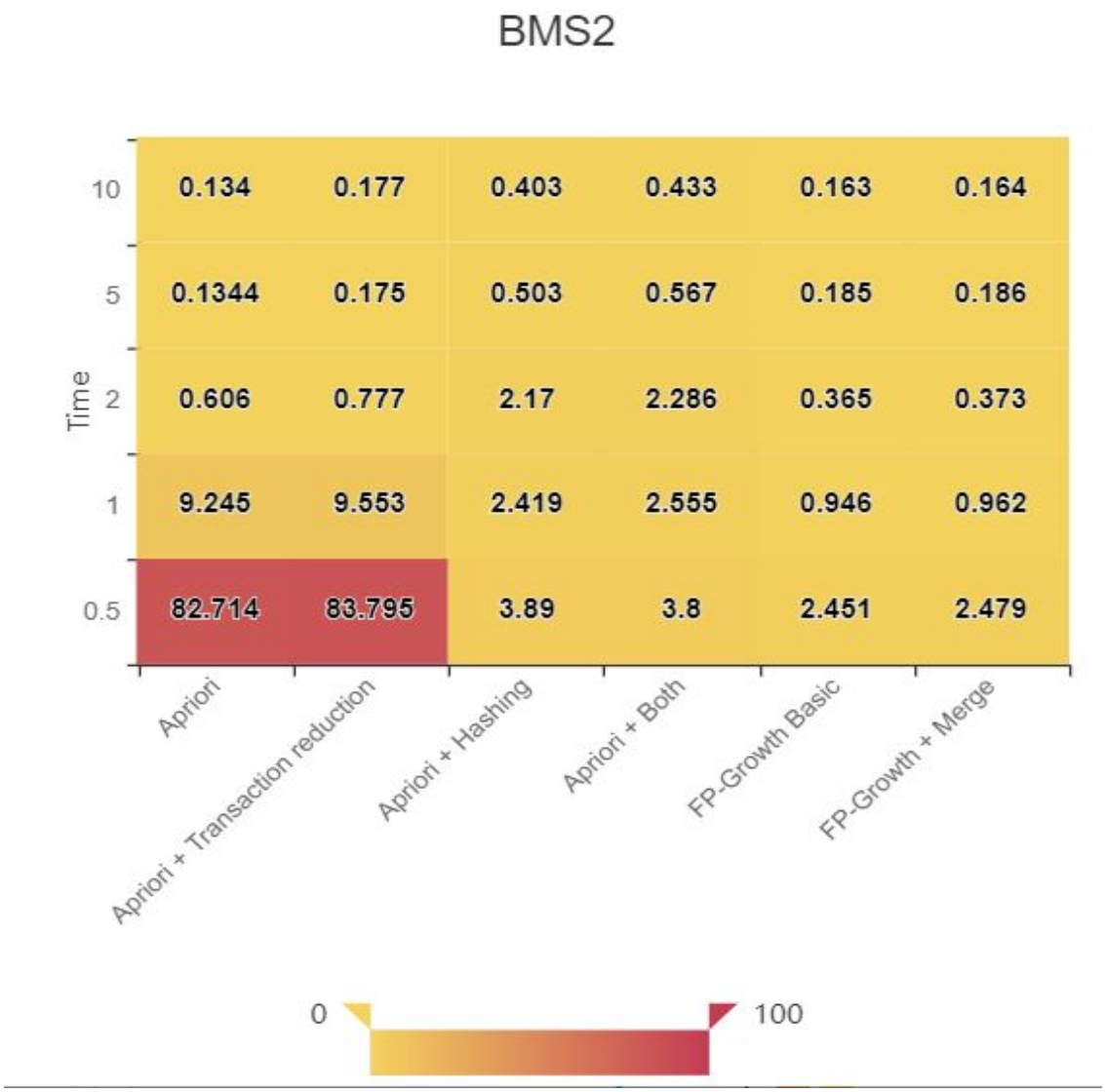
- We have done Merging strategy optimization.
- Two similar functions FP_Growth_Merge and FP_Mine_Merge are used. FP_Growth_Merge is an exact copy of the FP_Growth function and FP_Merge has the Merging strategy optimisation.
- An unordered_map is used to identify if a node was already visited or not and accordingly the transactions are saved in an array so that the path of a visited node need not be visited again if one of its children has been visited already.
- This follows a bottom up approach from every leaf node of the tree and as we move up, every visited node is marked and is not checked and iterated upon again, as the transaction corresponding to that has already been saved in a transaction array.
- This optimisation follows a dynamic programming approach which saves visited paths of the tree and hence these are not visited again.

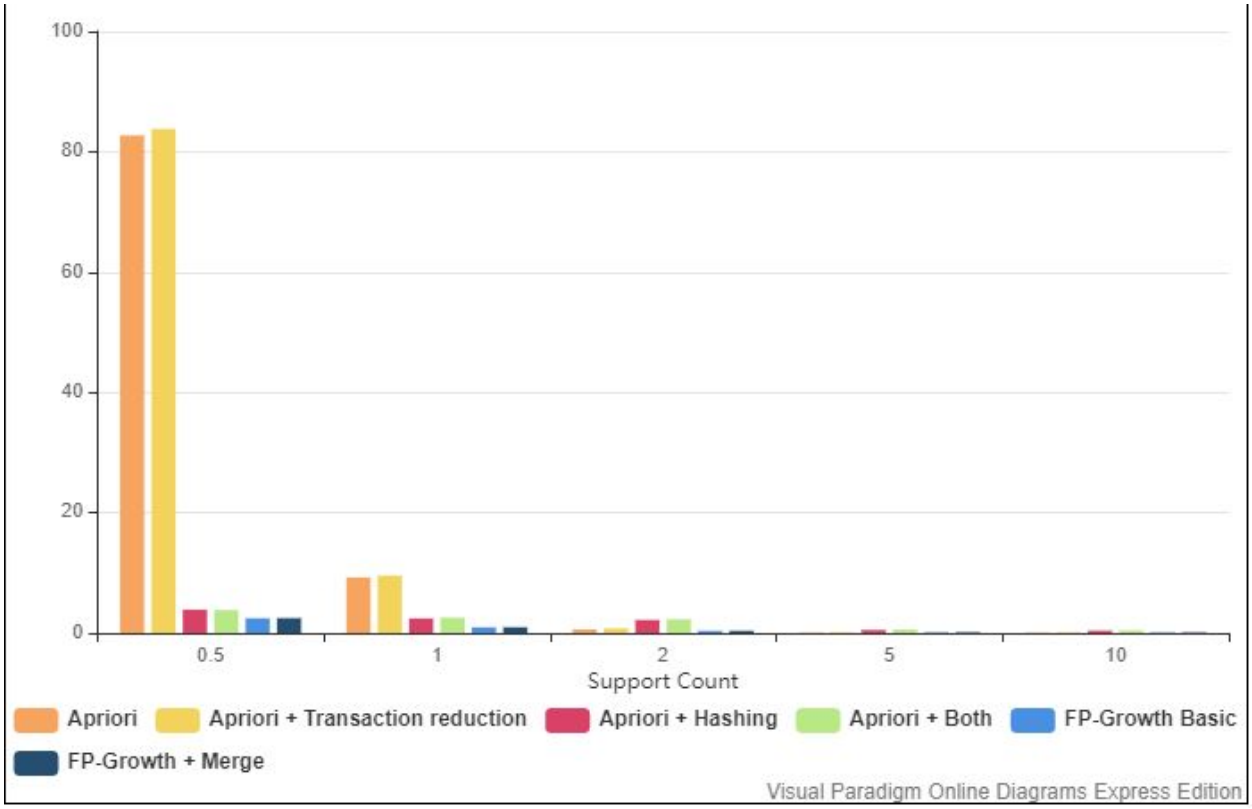
Comparison -

BMS1_spmf :

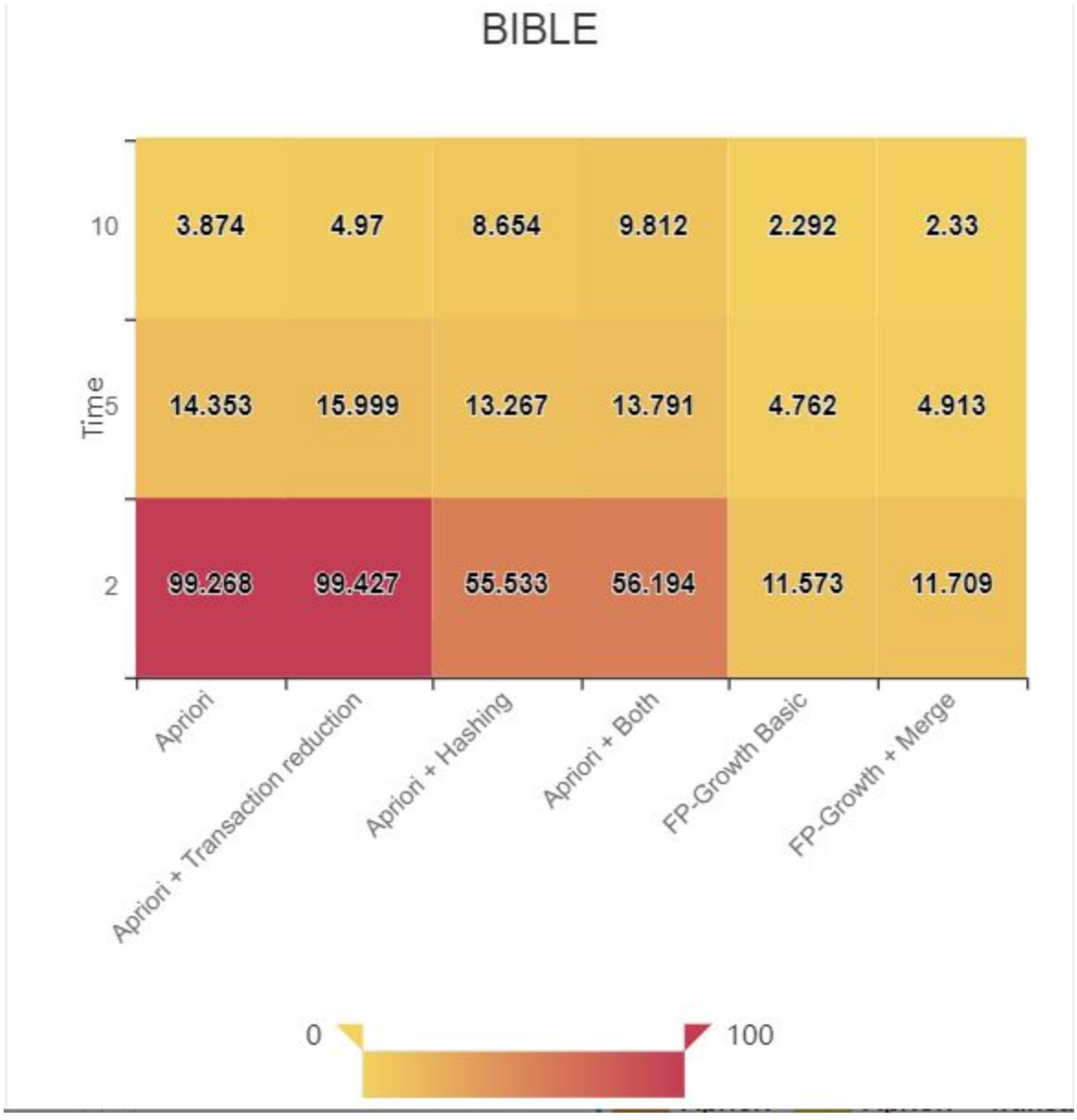

35
30
25
20
15
10
5
0
0.5
1
2
5
10
Apriori
Apriori + Transaction reduction
Apriori + Hashing
Apriori + Both
FP-Growth Basic
FP-Growth + Merge
Visual Paradigm Online Diagrams Express Edition

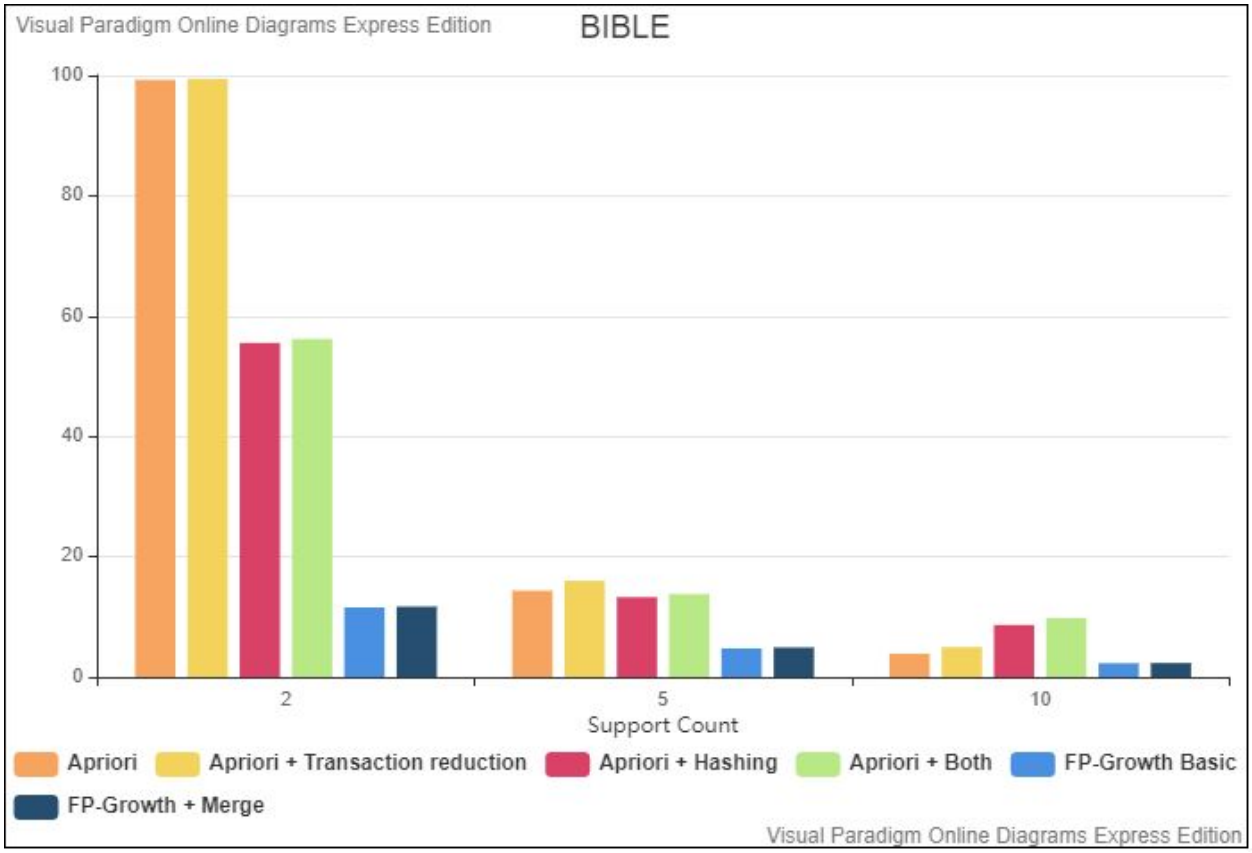
BMS2 :


BMS2
Time
10
5
2
1
0.5
0.134
0.177
0.403
0.433
0.163
0.164
0.1344
0.175
0.503
0.567
0.185
0.186
0.606
0.777
2.17
2.286
0.365
0.373
9.245
9.553
2.419
2.555
0.946
0.962
82.714
83.795
3.89
3.8
2.451
2.479
Apriori
Apriori + Transaction reduction
Apriori + Hashing
Apriori + Both
FP-Growth Basic
FP-Growth + Merge
0
100


100
80
60
40
20
0
0.5
1
2
5
10
Support Count
Apriori
Apriori + Transaction reduction
Apriori + Hashing
Apriori + Both
FP-Growth Basic
FP-Growth + Merge
Visual Paradigm Online Diagrams Express Edition

BIBLE :


BIBLE
Time
10
5
2
3.874
4.97
8.654
9.812
2.292
2.33
14.353
15.999
13.267
13.791
4.762
4.913
99.268
99.427
55.533
56.194
11.573
11.709
Apriori
Apriori + Transaction reduction
Apriori + Hashing
Apriori + Both
FP-Growth Basic
FP-Growth + Merge
0
100


Visual Paradigm Online Diagrams Express Edition
BIBLE
100
80
60
40
20
0
2
5
10
Support Count
Apriori
Apriori + Transaction reduction
Apriori + Hashing
Apriori + Both
FP-Growth Basic
FP-Growth + Merge
Visual Paradigm Online Diagrams Express Edition

Observations and Conclusions :

- Hash based Apriori Works best for lower minimum support value threshold however after that Normal implementation is preferred because for Hashed it generates all possible itemsets of size 2 which are present in the transactions whereas in Normal it uses L_1 for producing the possible frequent itemsets of size 2 which is very much less than those generated by hash for higher values minimum support value threshold.

- Hashed is very very fast as compared to any other type of Apriori when transaction size is also moderate. If transaction size will increase then hash will take a bit more time.

- Transaction Reduction is faster when we have a large number of transactions, when the minimum support count threshold is large and sometimes when the average size of transactions are large.

- On larger datasets Transaction Reduction Apriori performs better but on Shorter ones Normal Apriori perform better. This is because the time required for optimisation + execution in case of lower sized datasets becomes greater than the execution time of normal apriori.

- Both hashing and transaction reduction applied apriori is best among all aprioris for the average cases (data-set with significant number of items and transactions). This performs better than hashing only when the optimisation time + execution time is less than the execution time in Hashed Apriori.

- FP_Growths Merge optimisation works similar to the Normal implementation, this is because the dataset is loaded on to the main memory and hence does not make much of a difference.

- In case of FP-Growth both work similar and any one of them can be chosen. FP-Growth with the Merging optimisation would be preferred as for

bigger datasets which generate bigger trees, the merging optimisation will have significant improvement.

- As the data distribution gets dense, the algorithms take longer to run. By dense distribution we mean a dataset which has multiple reparations, thereby having higher order Frequent Itemsets - Upto Order 6 in case of BIBLE dataset. In the BIBLE data set the algorithms were run at 2% as the minimum as 0.5% and1% were taking too long to run.

- FP_Growth performs way faster than Apriori. This is because the FP-Growth algorithm focuses on only generating the frequent itemsets as compared to Apriori which generates certain sets which might not be frequent and then prunes them, which is an extra step.