ASSOCIATION RULE MINING PROJECT

Kartik Agarwal (2018102017) **Dhruv Sharma (2018101095)**

PART 1:

FP Growth:

Implementation:

- We have done Merging strategy optimization.
- First, we implemented the normal FP Growth algorithm, and then we worked on the optimization using the merging strategy.
- This can be done by pushing the right branches that have been mined for a particular item as it can be time and space-consuming to generate numerous conditional pattern bases.
- 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 result of it has already been saved.
- This optimization follows a dynamic programming approach that saves visited paths of the tree and hence these are not visited again.

Results FP-Growth (Minimum Support - 70%)

Database name	**Simple (time taken in ms)**	**Merging based (time taken in ms)**
Sign (~800 records, ~267 items)	253ms	249ms
Kosarak (~10k records)	196ms (No frequent itemsets)	191ms (No frequent itemsets)
Bible (~36k)	1225ms (1 frequent itemset)	1195 (optimized)

Results FP-Growth (Minimum Support - 60%)

Database name	**Simple (time taken in ms)**	**Merging based (time taken in ms)**
Sign (~800 records, ~267 items)	708ms	768ms
Kosarak (~10k records)	200ms (only 1 frequent itemset)	192ms (only 1 frequent itemset)
Bible (~36k)	1345ms(only 2 frequent itemsets)	1254ms(optimised)

Results FP-Growth (Minimum Support - 50%)

Database name	**Simple (time taken in ms)**	**Merging based (time taken in ms)**
Sign (~800 records, ~267 items)	2784ms	2884ms
Kosarak (~10k records)	163ms (only 1 frequent itemset)	166ms (only 1 frequent itemset)
Bible (~36k)	1410ms	1401ms

Observations:

For fp-tree: We can see that for most of the datasets optimized code(merging) works better than the naive-implementation

PART 2:

Apriori:

Implementation:
As per the standard technique, in order to find Lk , the frequent itemsets of size k from Lk−1 , the frequent itemsets of size k − 1, a two-step process is followed, consisting of join and prune actions for the candidate generation, i.e., to find Ck and hence starting from 1−frequent itemsets, we get to reach the k-frequent itemsets. To improve the efficiency of the level-wise generation of frequent itemsets, an important property called the Apriori property is used to reduce the search space.

We have also calculated the closed frequent itemset after calculating the frequent itemsets.

- Simple:
    - All nonempty subsets of a frequent itemset must also be frequent. The Apriori property is based on the following observation. By definition, if an item- set I does not satisfy the minimum support threshold, min sup, then I is not frequent, that is, P(I) < min sup. If item A is added to itemset I, then the resulting itemset (i.e., I ∪ A) cannot occur more frequently than I. Therefore, I ∪ A is not frequent either, that is, P(I ∪ A) < min sup.
- Hash-Based:
    - A hash-based technique can be used to reduce the size of the candidate k-itemsets, Ck , for k > 1. For example, when scanning each transaction in the database to generate the frequent 1-itemsets, L1 , 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).
    - While taking the 1 frequent itemsets I made the hash_table of pair (two itemsets) maps to int (support). So then iterate over the hash_table then if its support is greater than min_support, then it is two frequent itemsets.
- Partitioning:
    - Taking input from the user for the partition size. We divide the whole database into some partitions and run the **simple** apriori algorithm on such partitions. Then we merge the results of the partitions and check it on the whole dataset again to see if any of them are frequent. We aim to select the size of the partition close to that of the main memory so that the system does not have to go to the disk again and again which will increase the I/O time.

Results of Apriori (Minimum Support - 70%)

Database Name	Simple (time in ms)	Hash-Based (up to 2 itemsets) (time in ms)	Partition based (time in ms)

Sign(~800 records, ~267 items)	848ms	1129ms (Performs poor atsparTask 1 - tabletop object findingse)	943ms (Optimum partition size = 800 since the database is small)
Kosarak (~10k records)	73ms (no frequent itemsets)	75ms (no frequent itemsets)	111ms (no frequent itemsets)
Bible (~36k)	462ms	10055ms(sparse graph, only 1 frequent itemset)	878ms (again, best at partition size = # of records)

Results of Apriori (Minimum Support - 60%)

Database Name	Simple (time in ms)	Hash-Based (time in ms)	Partition based (time in ms)
Sign(~800 records, ~267 items)	2846ms	2126ms (starts performing better at lower min support)	2870ms (Optimum at partition size = 800 = # of records, since database is small)
Kosarak (~10k records)	73ms	3650ms (have to do for all pairs since 1 frequent itemset is found)	127ms (optimum at partition size = 11k, however, the difference is not much for size = 5k)
Bible (~36k)	537ms	9940ms (sparse graph)	1013ms (optima at psize = # of records)

Results of Apriori (Minimum Support - 50%)

Database Name	Simple (time in ms)	Hash-Based (time in ms)	Partition based (time in ms)

Sign(~800 records, ~267 items)	11733ms	9207ms (performs considerably better than simple-apriori since the graph is denser)	13259ms (Optimum at partition size = # of records since the database is small)
Kosarak (~10k records)	74ms	3839ms	126ms (optimum at psize = # of records)
Bible (~36k)	1189ms	8763ms(sparse at bigger database)	1909ms (optima at psize = # of records)

Observations:

For apriori: Simple works better than hash-based if the graph is sparse (minimum support is high), otherwise results show that hash-tree apriori is better in dense graphs.
However, we could not utilize the performance of partition-based apriori. This might be due to the fact that the system we are using was able to load the whole database into its main memory in a single go. For us to see the true benefit (significant), we would have to take databases that contain > 100k records).

To verify!
To verify that hash-based apriori is indeed better for dense graphs we run the simple and hash-based apriori again for the bible.txt database at lower values of minimum support.

Results:

Results of Apriori (Minimum Support - 20%)

Database Name	**Simple (time in ms)**	**Hash-Based (time in ms)**
Bible (~36k)	6248ms	12383ms(still worse!)

On lowering the minimum support further

Results of Apriori (Minimum Support - 10%)

Database Name	**Simple (time in ms)**	**Hash-Based (time in ms)**
Bible (~36k)	55723ms	19224ms(sparse at bigger database)

Finally.! We can see that hash-based is definitely better on dense graphs :D

Comparing between apriori and fp-tree, we see that in general fp-tree performs much better. This might be due to the fact that fp-tree doesn't generate all the candidate sets and only iterates on the prefixes of the itemsets which are frequent, they will necessarily have higher frequencies (sorted in decreasing order of frequency), however, this does not guarantee that all the combinations of itemsets will be frequent.

We have also implemented closed frequent itemsets, refer to the code. However, for most of the cases, closed frequent itemsets = frequent itemsets.