Data Analytics
ARM Project Report

Abhishek – 2018101028
Vivek – 2018111032

Dataset:

In the project, we have used the Leviathan dataset from: SPMF: A Java Open-Source Data Mining Library (philippe-fournier-viger.com). This dataset is a conversion of the novel Leviathan by Thomas Hobbes (1651) as a sequence database (each word is an item).

```
Items: 9025

Sequences: 5834

Mean sequence length: 33.8
```

Apriori Algorithm:

Apriori is an algorithm for frequent item set mining and association rule learning over relational 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.

Output:

In this project, we considered the first 1000 sequences from the Leviathan dataset with minimum support as 50%. The output for the base implementation of the algorithm was:

```
{'17': 509,
 '71': 509,
 '18': 752,
 '14': 572,
 '8': 704,
 '18-8': 601}
```

The associated words are:

```
18=of

8=the

14=and
```

17=is

71=to

Partitioning:

By leveraging distributed/parallel processing, we can considerably speed up the process. We can divide the dataset D into n partitions s and frequent itemset mining is performed in each of them individually, any itemset that is potentially frequent with respect to D must occur as a frequent itemset in at least one of those n partitions. Now the algorithm can be on each of those partitions in parallel and all the local frequent itemsets generated can be checked for being globally frequent by a scan of the database again.

Hashing:

For very large databases, processing so many itemsets can be very time taking especially when generating combinations of items. In hashing optimization, we first calculate all the 2-itemsets for each transaction/row in the dataset and mapping them to different buckets while maintaining count of all the 2-itemsets in each bucket.

Now, we eliminate all the buckets that have the associated count/frequency lower than the minimum support threshold. This way, we considerably reduce the processing time of any subsequent operations in the Apriori algorithm.

Analysis:

Apriori: 2.69s

Partition: 1.56s

Hashing: 117s

Parition+Hashing: 39.2s

As we can see, the optimizations reduce the runtimes. However, hashing increases the time significantly. This can be explained by the fact that this example is for a smaller dataset. In a smaller dataset at k=n where n>2, the number of itemsets is smaller. The cost of pre-processing 2-itemsets does not compensate for the cost of the much lesser number of larger itemsets.

In a larger dataset, hashing might effectively reduce the overall runtime.

FP-Growth Algorithm:

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.

Output:

In this project, we considered the first 1000 sequences from the Leviathan dataset with minimum support as 50%. The output for the base implementation of the algorithm was:

```
[['17'], 509,
 ['71'], 509,
 ['14'], 572,
 ['8'], 704,
 ['8', '18'], 601,
 ['18'], 752]
```

The associated words are:

```
18=of
8=the
14=and
17=is
71=to
```

Here we can see that, the final output is exactly the same as the Apriori implementation (as it should be).

Merging optimization:

In the bottom-up projection technique of the FP-growth algorithm, we start with the item with the least support in the header row first. Therefore, in a given path in the tree as I2-I1-I3-I5, in the subsequent iterations this path will be traversed again which can be avoided. This is because keeping the nodes in memory is a waste of memory and time.

This can be avoided by deleting the part of the path that becomes redundant after the traversal and rather push into the next item's (in this example, I3) conditional pattern base.

Analysis:

FP normal: 2.41s

FP optimized: 2.13s

As we can see, using the merging strategy the time is reduced.

Comparing Apriori and FP-Growth

We notice that the FP-growth algorithm is faster than the Apriori algorithm. This confirms the previously stated fact.