YAFIM : Apriori Algorithm

Guided By: Prof. PM Jat

Mentored By: TA Vinay Sheth

Devarshi Joshi 202003007
Arpan Shingala 202003013
Mukund Ladani 202003039
Nikhil Vaghasiya 202003042

December 17, 2022

Objective

- To research Apriori Algorithm, its application, and implementations.
- To enhance upon the serialized Apriori Algorithms and research ideas, including parallelization and gear like SparkRDD, Map-Reduce, etc.
- The serial Apriori algorithm is both data-intensive and computing-intensive. This makes the serial Apriori algorithm very time-consuming while running on big data. Our objective is to resolve this hassle by using parallelization. We can parallelize it with the assistance of SparkRDD.

Algorithm

- First, we need to find unique items present in the dataset. We do so by using a map on the given data file and creating a dataset.
- Now, we created a function to generate a new candidate itemset (c_k) using previously generated frequent itemsets (f_{k-1}). We do so by "smartly" selecting two **itemset** (set data-structures) which have exactly 1 distinct element and taking their union.
- Using the candidate itemsets generated above, we will find the frequent itemsets. First, we will count in how many transactions the items come. Then we will remove items whose count is less than minimum support. Here we will use a spark mapper to parallelize the process and for faster execution.
- After getting frequent itemsets we will repeat the above process until we can't find itemsets whose count is greater than minimum support.

Implementation insight

- Let us store the file content in a new file. This can be done by **flatmap(data).distinct()**, where distinct() is an inbuilt function that removes the repeating elements.
- In k^{th} iteration, let $|it1| = k$ and $|it2| = k$. If $|it1 \cap it2|$ = k-1, then we create the new candidate itemset = it1 $\cup$ it2.
- Let get_sup be the function that gives items whose count is greater than minimum support. Here we use spark map to parallelize items to execute the get_sup function. So we get frequent itemset f_k at k^{th} iteration.
- If $|f_k| = 0$, then stop algorithm.

Datasets

- Chess: a data set listing chess end game positions for king vs. king and rook
- pumsb: The pumsb dataset contains census data for population and housing [1][**1**]

Dataset	Number of Items	Number of transactions
Chess	75	3,196
Pumsb	2,113	49,046

Table: Properties of Datasets

- Apart from the above datasets, we have also used chess11, chess12, chess21, and chess22 which are truncated datasets generated from the Chess dataset.

[1] https://archive.ics.uci.edu/ml/support/census+income

Results analysis

- Here, as we can see the parallel algorithms are faster than the linear algorithms
- In the case of the chess data set, the parallel algorithm is about 37.92% faster and in the case of pumsb, it is 44.06% faster.[a][2]

[a] Results

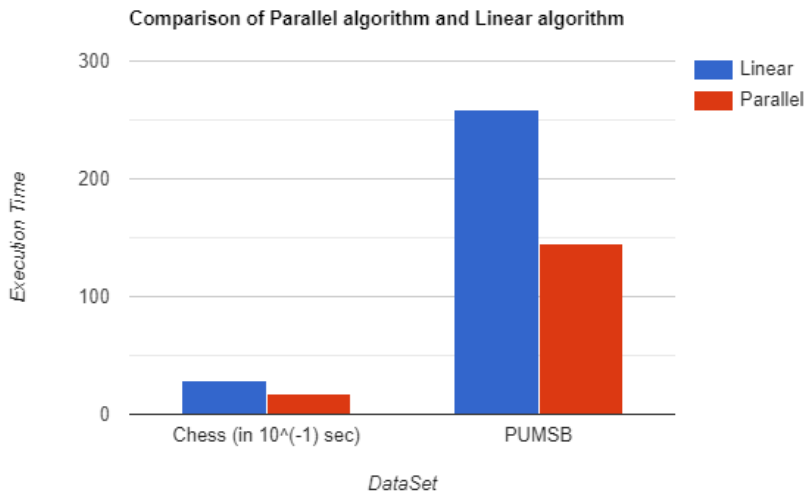

Figure: Comparison between Parallel and Linear algorithms

Results analysis

- chess11 and chess12 datasets have 700 transactions with 75 and 73 unique items respectively.
- chess21 and chess22 datasets have 1000 transactions with 75 and 73 unique items respectively.

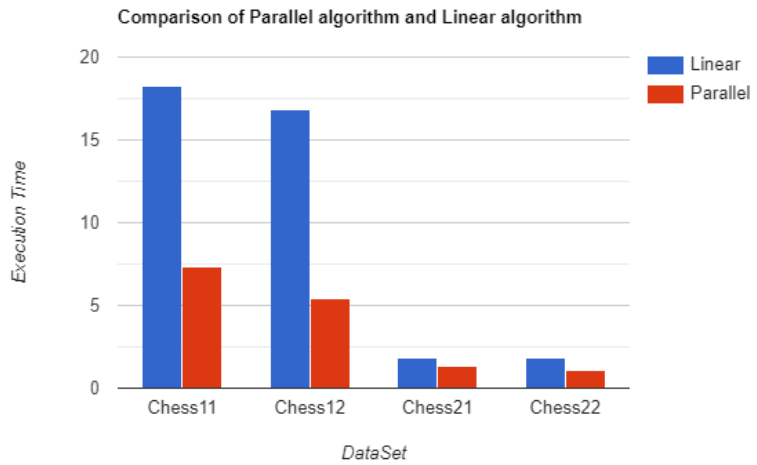

Figure: Chess dataset

- As we can see, the parallel algorithm is faster than the linear one. However, chess11 having less items than chess 21 is still faster(which is counter-intuitive). This is because we have taken min_support = 98% of the size of the dataset. This leads to items being considered "frequent" more in the first dataset than in the second. As a result, as the number of iterations increases, the candidate itemset in the first case will be a lot larger than the second one. Hence, taking more time.

Conclusions

Finally, using the execution time and graphs plotted from the modified chess dataset, we can conclude that the parallel apriori algorithm is much faster than the linear apriori algorithm.

Also, we can conclude that for smaller data sets, the difference between the execution time is more compared to that of larger data sets (in cases having fixed min_support in %age of dataset size).

Reference

[1] https://archive.ics.uci.edu/ml/support/census+income

[2] https://docs.google.com/spreadsheets/d/1Ygipnru1_fRrdRCKVv7-CtvU8Ak6kFgaqZjq5OWboms/edit?usp=sharing