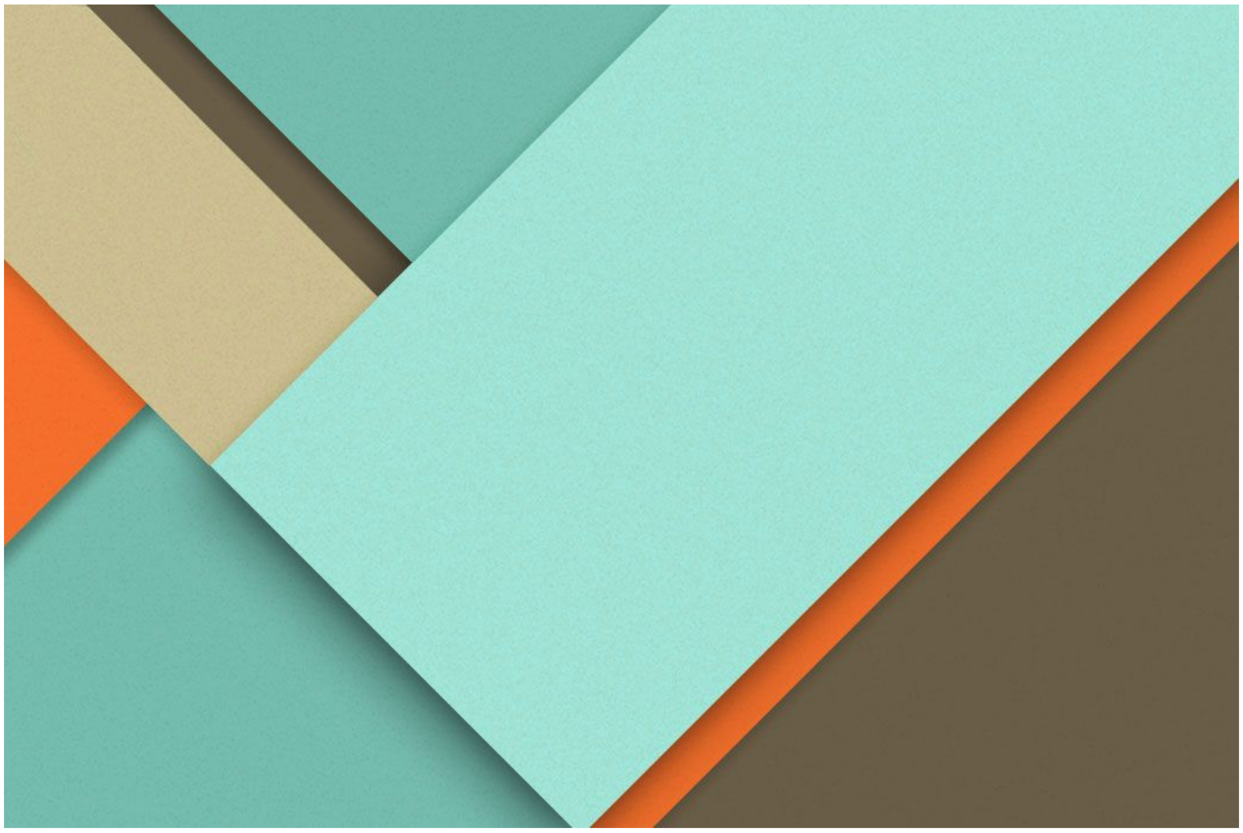


Implementation of Parallel algorithm for Association rule mining

20161063 (S.Preetham)

20161145 (P.Pradeep)

Introduction

Finding frequent itemsets is one of the most important field of research in data mining . Several research has been done on this topic since long time . Of all the algorithms that are present, the Apriori algorithm and the FP growth algorithms have been improvised, implemented and evaluated several times .Both these algorithms were well defined and widely accepted in the association mining tasks. Several research papers were published improvising the time complexity of these algorithms over the decade since the first time these algorithms were written .Among many techniques that were used to improvise these algorithms parallelization is one of the important technique used to improve them . Parallelization of these algorithms has reduced the time complexity of these algorithms by a significant amount . In this paper we implemented , one of these algorithm which is parallel in nature and provided the comparison between the parallelized approach and the normal one .

Tools and Source:

We implemented the PFP-growth algorithm from the following paper:
Pfp: parallel fp-growth for query recommendation
Python spark library was used to implement parallelization . There are two slave nodes and one master node in the network . The parallelization is due these two slave nodes which act parallelly over the course of the algorithm resulting in improving the time taken .

The code is attached in the respective directory source_code .
To run the algorithm for a specific input data file

```
 spark-submit --conf PYSPARK_PYTHON=/usr/bin/python3 <input path>
<output path> minSupport .
```

DATASETS:

We used four datasets to validate our algorithm from the following link :

Frequent Itemset Mining Dataset Repository

From the above data we used the following datasets:

Chess: originally from the ML Repository, by R. Bayardo.n=3196, k=75

Mushroom:originally from the ML Repository, prepared by R. Bayardo

T10I4D100K: artificially generated market basket data

T40I10D100K :artificially generated market basket data n=100 000, k=1000

Algorithm:

-> Sharding: Dividing DB into successive parts and storing the parts on P different computers. Such division and distribution of data is called sharding, and each part is called a shard.

->Parallel Counting : Doing a MapReduce pass to count the support values of all items that appear in DB. Each mapper inputs one shard of DB. The result is stored in F-list.

The mapper is fed with shards of DB. For each item, say aj ∈ Ti, the mapper outputs a key-value pair. After all mapper instances have finished, for each key' generated by the mappers, the MapReduce infrastructure collects the set of corresponding values

-> Grouping Items: Dividing all the items on FList into Q groups. The list of groups is called group list (G-list), where each group is given a unique groupid (gid).

-> Parallel FP-Growth: The key step of the whole algorithm. This step takes one MapReduce pass, which contains map stage and reduce stage

Mapper – Generating group-dependent transactions: Each mapper instance is fed with a shard of DB generated. Before it processes transactions in the shard one by one, it reads the G-list. With the mapper algorithm , it outputs one or more key-value pairs, where each key is a group-id and its corresponding value is a generated group-dependent transaction.

Reducer – FP-Growth on group-dependent shards: When all mapper instances have finished their work, for each group-id, the MapReduce infrastructure automatically groups all corresponding group-dependent transactions into a shard of group-dependent transactions.
Each reducer instance is assigned to process one or more group-dependent shard one by one. For each shard, the reducer instance builds a local FP-tree and growth its conditional FP-trees recursively. During the recursive process, it may output discovered patterns.

->Aggregating the results generated as our final result.

The aggregating step reads from the output from above step .For each item, it outputs corresponding top-K mostly supported patterns. In particular, the mapper is fed with pairs.Because of the automatic collection function of the MapReduce infrastructure, the reducer is fed with pairs in the form of hkey' = aj, value' = V(aj)i, where V(aj) denotes the set of transitions including item aj. The reducer just selects from S(aj) the top-K mostly supported patterns and outputs them.

Experiments

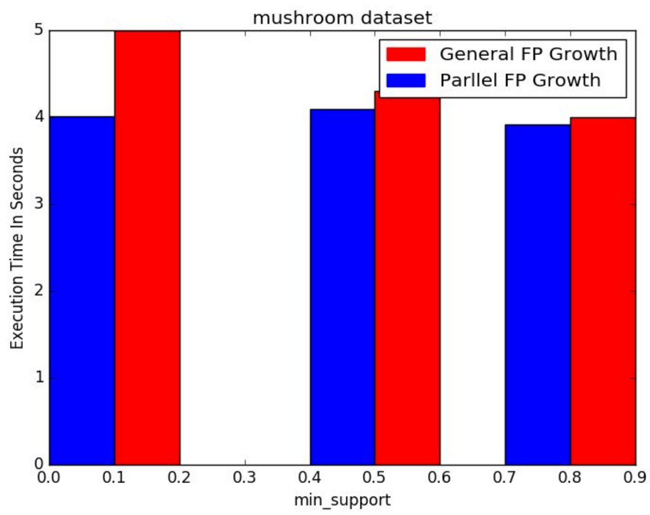
We have implemented the above mentioned parallel algorithm of FP-Growth . The code is attached . These are the results we have obtained with the serial implementation across various support thresholds .

Values

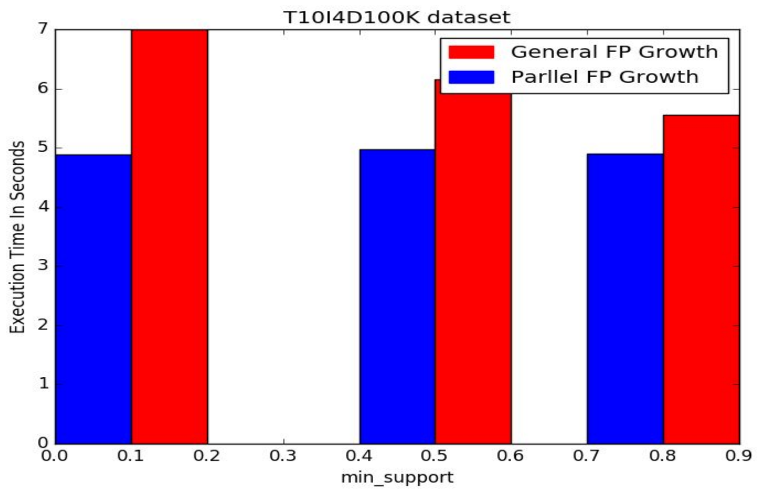
DATASET	MINSUPPORT	TIME TAKEN(Sec)	ALGORITHM
Mushroom	0.8	3.92	parallel FP
	0.5	4.09	
	0.1	4.008	
	0.8	4	Serial FP
	0.5	4.3	
	0.1	5	
T10I4D100K	0.8	4.90	Parallel FP
	0.5	4.97	
	0.1	4.88	
	0.8	5.56	Serial Fp
	0.5	6.15	
	0.1	7	
T40I10D100K	0.8	6.97	Parallel FP
	0.5	7.2	
	0.1	7.894	
	0.8	9.65	Serial FP
	0.5	12.5	

	0.1	16	
Chess	0.8	3.83	Parallel FP
	0.5	3.86	
	0.1	3.91	
	0.8	3.8	Serial FP
	0.5	3.7	
	0.1	5	

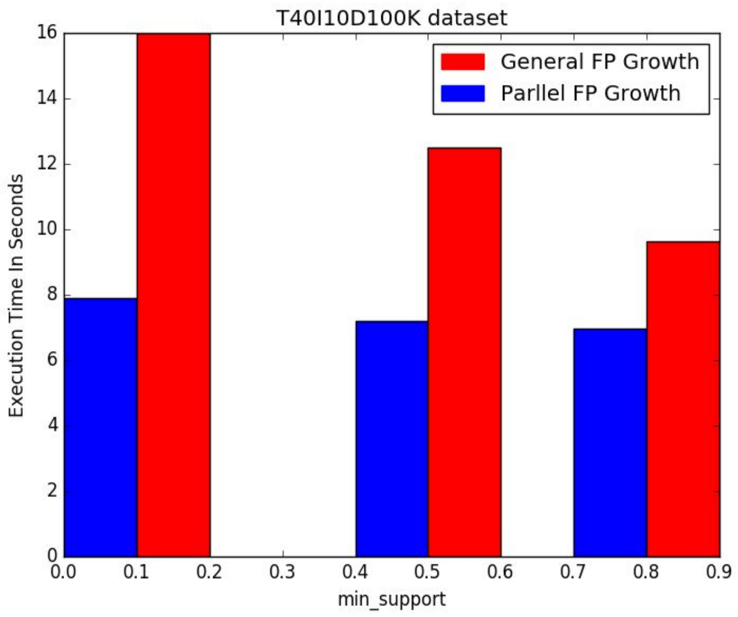
Pictorial representation for Mushroom dataset



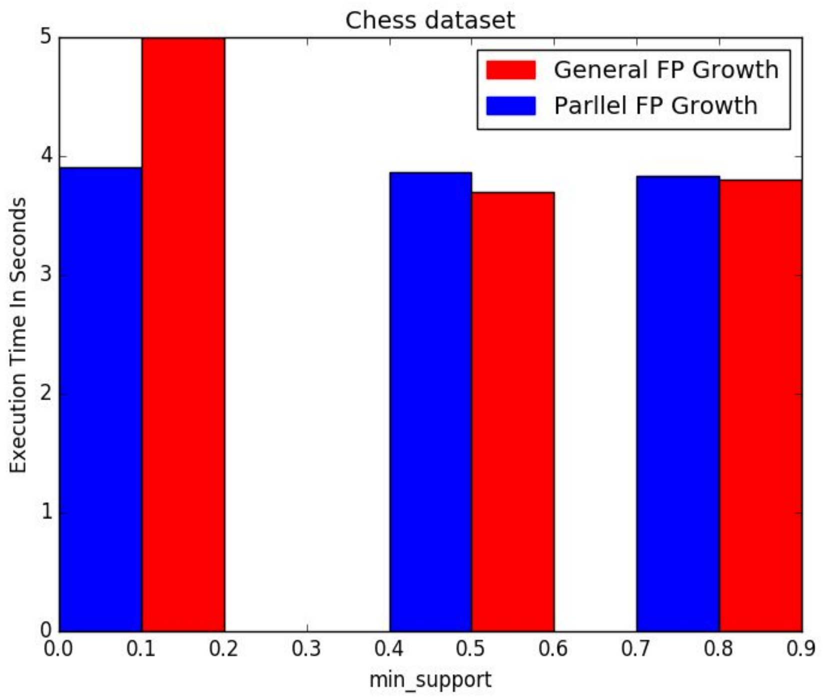
Pictorial representation for T10I4D100K dataset



Pictorial representation for T40I10D100K dataset



Pictorial representation for Chess dataset



Conclusion:

It is clearly evident from above that the time taken for mining the frequent itemsets in case of parallel and normal implementation doesn't change that much when the data is low (Either number of transactions or the total number of different items were low) . This is the case for the mushroom, chess and T10I4D100K datasets .But in the case of T40I10D100K the parallel implementation definitely outperformed the naive implementation due to large number of items in the dataset and also the number of transactions .