Array 7 (2020) 100035

Contents lists available at ScienceDirect

Array

journal homepage: www.elsevier.com/journals/array/2590-0056/open-access-journal

A heuristic approach for load balancing the FP-growth algorithm on MapReduce

Sikha Bagui *, Keerthi Devulapalli, John Coffey

Department of Computer Science, University of West Florida, Pensacola, FL, USA

ARTICLE INFO

Keywords:
Association rule mining
Frequent pattern growth algorithm
Load balancing
MapReduce
Hadoop

ABSTRACT

Frequent itemset discovery is an important step in Association Rule Mining. The Frequent Pattern (FP) growth algorithm, often used for discovering frequent itemsets, cannot scale directly to today's Big Data, especially for large sparse datasets. Hence there is a need to distribute and parallelize the FP-growth algorithm. Parallel FP-growth (PFP) is a parallel implementation of the FP-growth algorithm on Hadoop's MapReduce execution framework. Though PFP scales to large datasets, it suffers from imbalanced load across processing units. In this paper we propose a heuristic based, lower order of complexity, load balancing strategy for the PFP algorithm, called Heuristic Based PFP (HBPFP). Our results show that HBPFP distributes the load more evenly across the Hadoop cluster nodes, runs faster than the PFP algorithm, and uses cluster resources more efficiently, especially for large sparse datasets.

1. Introduction

Association Rule Mining (ARM), most commonly used in transactional databases, is a process of discovering relationships among items or itemsets. Association rules are formed from frequent itemsets, hence the first step of ARM is frequent itemset discovery. A well-known algorithm for discovering frequent itemsets is the Apriori algorithm [2]. This algorithm requires multiple scans of a database, which is inefficient in today's Big Data era. The FP-growth [3] algorithm is another well-known algorithm for discovering frequent itemsets. It is based on the FP-Tree data structure, which stores frequency information of items in a concise form. Unlike the Apriori algorithm, the FP-growth algorithm only requires two scans of the database and it does not generate candidate itemsets. Hence the FP-growth algorithm is considered more efficient than the Apriori algorithm [3].

But, the FP growth algorithm has performance and scalability issues [3]. The FP-Tree, for very large data sets, will not fit in memory, making it difficult to process Big Data. A distributed and parallel implementation of the FP growth algorithm is needed to scale to large data sets. Hadoop [4] is a generic distributed framework that provides two features most important for Big Data solutions – a distributed file system, Hadoop's Distributed File System (HDFS), for efficiently storing large data sets in the size of petabytes, and an efficient distributed and parallel execution framework called MapReduce to process data stored in HDFS. Data stored in HDFS is divided into data blocks and each block is stored on a different cluster node, allowing for parallel processing on the nodes.

There have been several distributed implementations [9–15] of the FP-growth algorithm on different platforms. It can be observed from these implementations that apart from implementing this data mining algorithm on different platforms, there is an overhead of task distribution, task scheduling, effective resource utilization, task synchronization, etc. These tasks are related to maintaining a distributed system. Inspite of the advantages of using a parallel and distributed file system, [1] implemented a parallel version of the FP growth algorithm called Parallel FP (PFP) growth on Hadoop's MapReduce framework, and this algorithm did not produce optimized results due to imbalanced load in the nodes.

The main motivation of this paper is in the need for load balancing in the MapReduce environment in order to efficiently handle memory intensive algorithms like FP-Tree, especially for large sparse data sets which would be even more memory intensive. Dividing the data equally between the nodes in the cluster, which seems like the most logical thing to do, does not necessarily mean that the load is balanced for every algorithm that is being run on Hadoop's MapReduce environment. So, what is meant by *load* balancing? This can be explained by defining load. Load is the work to be performed by a node. Load influences the time taken by a node to complete its share of an overall task. When some nodes

* Corresponding author.
E-mail addresses: bagui@uwf.edu (S. Bagui), keerthi0589@gmail.com (K. Devulapalli), jcoffey@uwf.edu (J. Coffey).

https://doi.org/10.1016/j.array.2020.100035
Received 17 February 2019; Received in revised form 2 May 2020; Accepted 2 July 2020
Available online 8 August 2020

are assigned more work than others, the load is not balanced. This will lead to an increase in the overall run-time of a distributed algorithm since the distributed algorithm waits for the node taking the longest runtime (the node with the highest load) to complete.

In this paper, we propose a heuristic load balancing strategy, that will be particularly useful for sparse datasets, that distributes the load on all cluster nodes in a such way that the load on the nodes is more or less the same for the FP-growth's parallel implementation. We will refer to this as the Heuristic Based Balanced Parallel FP (HBPFP) growth algorithm. The heuristic behind this load balancing strategy is to identify what constitutes load or work in a given algorithm, and group items such that each group gets a fair share of items for which more work has to be performed and items for which less work has to be performed, hence balancing the overall load.

HBPFP is compared with two established load balancing algorithms, Mahout's PFP (MPFP) [7] and Balanced PFP (BPFP) [6], using a dataset that can be considered sparse, the transactional dataset from the Frequent Itemset Mining Dataset Repository [19]. Our proposed HBPFP outperforms both MPFP and BPFP in terms of run-time performance, load balancing, memory performance, and scalability, showing that it uses cluster resources more efficiently.

The rest of the paper is organized as follows: section two provides a background on FP-Growth on MapReduce and load balancing; section three discusses related work; section four provides the background for the current work; section five explains the heuristic based load balancing grouping algorithm along with a working example; section six provides implementation details; section seven describes the experimental setup; section eight presents the results and discussion; and finally, section nine presents the conclusion.

2. FP-growth on MapReduce and loadbalancing

Hadoop's MapReduce environment has two main phases, map and reduce. Data in read into the map phase, each mapper processes this data and emits key-value pairs. These key-value pairs then become input for the reduce phase. In the reduce phase, key-value pairs from different mappers are combined to produce the final result.

The FP-Growth algorithm has two phases – the tree construction phase and the growth phase. In the PFP [1] algorithm, the mapper generates a transaction tree containing the frequent items for each transaction and emits it as intermediate data for each group. In the reducer, for each group, all the emitted transaction trees are combined to form the FP-tree of that group. The FP-tree construction is an O(n*m) operation, where *n* is the number of transaction trees to be combined and *m* is the size of the transaction trees. The number of transaction trees generated for a group is equal to the number of transactions present in the database that contain at least one of the group items. The number of transaction trees is reduced by using a combiner. The transaction trees with least frequent items will be larger in size than the transaction trees with most frequent items. So a group containing least frequent items will have to combine larger transaction trees and hence will spend more time in the tree construction phase.

In the growth phase, the conditional pattern base is constructed for an item and mined to generate the frequent patterns for the item. Hence the time spent in the growth phase is directly proportional to the size of the conditional pattern base of the item. Items that occur deep in the FP-Tree, that is, towards the leaves, have long prefix paths and hence will have bigger conditional pattern bases. More time will be spent on constructing and mining the conditional FP-trees for such items. For the items that occur towards the root of the FP-tree, the prefix paths are shorter and their conditional pattern bases are smaller. Less time is spent on mining these small conditional pattern bases. Therefore, in the growth phase, more time is spent on less frequent items that occur deep in the FP-tree and less time is spent on more frequent items that occur towards the root of the FP-tree. Overall the FP-Growth algorithm spends more time on less frequent items than on more frequent items. So, overall, the FP-Growth algorithm will have more of a problem in sparse databases, which will have more less frequent items.

To distribute the FP growth algorithm across all reducer tasks, each division must be assigned to a reduce task to construct the FP tree and mine it for frequent itemsets. Let $I = (a_1; a_2; \ldots; a_m)$ be a set of items, and DB be the transaction database, denoted by $DB = (T_1; T_2; \ldots; T_n)$, where each $T_i \subseteq I$ $(1 \leq i \leq n)$ is a transaction. Let the set of items, I, be divided into different disjoint non-empty subsets, I_1, I_2, I_3, etc. Each subset I_i is assigned to a reducer to be processed. For each subset I_i, an FP tree is constructed from all the transactions in DB and the FP growth algorithm is used to obtain the frequent patterns for the items in the given subset. Load balancing occurs when the time taken for each subset I_i, in the FP tree construction and FP growth phases, are almost equivalent, that is, time taken for $I_1 \approx$ time taken for $I_2 \approx$ time taken for $I_3 \approx \ldots$ time taken for I_i.

3. Related work

Several works have focused on the performance issues on the serial version of the FP growth algorithm. These related works can be divided into two broad areas: (i) works that focused on distributing and parallelizing the FP growth algorithm on different execution platforms; and (ii) works that focused on tasks related to maintaining distributed systems, that is, algorithms that focused on the overhead due to the distribution of tasks, task scheduling, effective resource utilization and task synchronization, etc.

3.1. Works that focused on distributing and parallelizing the FP growth algorithm on different execution platforms

[9] presented a multi-threaded algorithm, Multiple Local Frequent Pattern Tree (MLFPT), that runs on SMP processors with shared memory. In this algorithm, multiple local FP-trees are constructed, one by each processor. These local FP-Trees are shared among all processors in the growth phase. The load on the processors is balanced using the average number of occurrences that need to be traversed by each processor, and this is considered the load factor. MLFPT scales up well for very large datasets.

[10] describes the FP growth algorithm underutilizing the modern processor's architectural features. Hence, the FP tree was implemented as a cache-conscious data structure which improved performance by taking advantage of hardware cache prefetching. It was implemented as a multi-threaded program where the threads were co-scheduled to maximize cache reuse.

[11] stated some parallelization techniques based on shared memory SMP architectures for data mining algorithms. Using these techniques, they parallelized the FP growth algorithm and scaled it to process datasets of few hundred megabytes (MBs). Similarly, [12] used two techniques: (i) a cache-conscious FP-array, which is a data reorganization of the FP tree and, and (ii) lock-free FP-tree construction, for parallelizing the FP growth algorithm on multi-core processors. These techniques aim to solve problems like poor data locality and insufficient parallelism of data mining algorithms on multi-core processors. [13] implemented a multithreaded solution on the multi-core CPU and GPU. Both implementations were based on FP-arrays [12].

[14] implemented a parallel FP-growth algorithm on a distributed memory multiprocessor system using message passing. It is based on a master-slave approach. In this solution, each processor is assigned a partition of the database and performs the FP-growth algorithm in parallel on the local partition. The processors initially compute local frequency lists, from which the global frequency list is created and distributed to all processors. Using the global frequency list, each processor builds local FP trees. [15] implemented an architecture-aware solution, Distributed FP growth (DFP), for parallelizing the FP growth algorithm on a message-passing cluster.

3.2. Works that focused on tasks related to maintaining distributed systems

PFP [1] is a distributed and parallel implementation of the FP growth algorithm on Hadoop's MapReduce framework. The grouping strategy used in the PFP [1] algorithm is: compute- the size of each group (n) by dividing the total number of items in the frequency list by the number of groups. Starting from the most frequent item, n consecutive items are grouped into one group. For each group, the group dependent transactional database is generated. Each group is assigned to a different reducer. The grouping strategy does not take load balancing into consideration. Hence, the processing needed for different groups can be varied and the load on the nodes performing FP growth on the groups can be different. The overall running time of the PFP algorithm can be extended by the time taken by the reducer with the largest load. In the PFP [1] algorithm, the group dependent transactional databases generated for each reducer key cannot be split. Hence the PFP [1] is a non-distributive MapReduce program and cannot use tools like Perfect Balance for load balancing. The program implementation must take the responsibility of load balancing.

A load balancing strategy was introduced for PFP called Balanced PFP (BPFP) [6]. This algorithm performs load balancing based on a load factor which was measured in terms of the log of the location of the item in the reverse sorted frequency list. The load factor was based on estimations for measuring the load of an item in the FP growth phase. The FP tree construction phase, which also contributes to the load of an item, was not part of the estimations.

Improved Parallel FP growth (IPFP) algorithm [16] integrates a small file processing strategy into the PFP algorithm. IPFP works efficiently for datasets that have a very large number of small files, where each file size can be as small as 64 KB. Such datasets are not suitable for Hadoop (both with respect to HDFS and MapReduce), as they tend to have high memory cost, high I/O overhead, and low computing performance. IPFP tries to address these problems by merging the small files as a preparatory step before applying the PFP algorithm. The IPFP algorithm does not perform any load balancing.

PARMA [17] is a combination of parallelization and randomization. It mines, in parallel, a set of small random samples and then filters and aggregates the collections of frequent itemsets or association rules obtained from each sample.

[20] created a closed FIM algorithm basing the grouping method on a greedy strategy, such that all groups receive an equivalent number of transactions.

FiDoop-DP [21] balances the load on the reducer nodes by using a voronoi based partitioning technique. The partitioning technique almost creates clusters of transactions and mines the clusters in parallel in different reducers. FiDoop-DP not only aims to balance the load of the reducers, but also avoids duplicate transactions in the intermediate data. But FiDoop-DP needs a pre-processing step. Their experimental results do not account for this preprocessing step and hence the overhead of the pre-processing step is not known.

[22] introduced two algorithms that used the MapReduce framework to deal with two aspects of the challenges of Frequent Itemset Mining of Big Data. They introduced Dist-Eclat to optimize for speed and BigFIM optimized for a truly Big Data implementation using a hybrid algorithm.

[23] proposed MapFIM+, a two-phase approach to frequent itemset mining in very large datasets benefiting both from a MapReduce-based distributed Apriori method and local in-memory Frequent Itemset Mining methods. In this approach, first, MapReduce was used to generate frequent itemsets, then an optimized local in-memory mining process was used to generate all remaining frequent itemsets from each prefix-projected database on individual processing nodes.

Our heuristic approach to balance groups, which seems to be natural and simple for a parallel implementation of the FP growth algorithm, especially for large sparse datasets, has not been attempted or tested by any of the earlier works. In our work, every group has an equal share of more frequent items and less frequent items, and the algorithm of the mappers and reducers is unaffected by the balancing strategy.

And although there are some commercially available tools like Oracle Big Data Appliance's Perfect Balance [5] that perform load balancing in the reducer phase of the MapReduce programs, these tools are not applicable to all MapReduce programs. For example, Perfect Balance can only be used with distributive MapReduce programs, that is, programs where the group of records associated with a reduce key can be split. Hence, for algorithms that do not meet this criterion, program implementation must take the responsibility of load balancing.

4. Background for current work

This section describes the two Parallel FP-Growth Algorithms, Mahout's PFP [7] and Balanced PFP [6], that our HBPFP will be compared with.

4.1. Mahout PFP

Mahout [7] is an open-source scalable machine learning library which contains an implementation of the PFP algorithm [1] on the Hadoop platform, which we will refer to as MPFP. The MPFP algorithm uses two MapReduce jobs: the first MapReduce job generates the reverse sorted frequency list of frequent items and the second job generates group dependent transactional databases and executes the serial FP growth algorithm on all the group dependent transactional databases in parallel. Before running the second MapReduce job, the MPFP algorithm groups the items of the frequency list into multiple groups. The number of groups to be created is a user given constant.

The grouping strategy used in MPFP computes the size of each group (n) by dividing the total number of items in the frequency list by the number of groups. Starting from the most frequent item, n consecutive items are placed in one group. Following is an example of how the MPFP grouping algorithm works based on the transactions shown in Table 1. Let the minimum support be 3 and the number of groups be 3.

As the first step of FP-growth algorithm, the reverse sorted frequency list is computed as shown in Table 2. The number of items in the frequency list is 6. The number of items per group is calculated as number of items in the frequency list divided by the number of groups. For this example it is 6/3, i.e. 2 items per group.

In the MPFP growth algorithm, the first 2 items will be placed in group1, the next 2 items will be placed in group2 and the last 2 items will be placed in group3, as shown in Table 3. The length of the prefix path is a measure of the size of the conditional pattern base of an item. If the length of the prefix path is small, the conditional pattern base will also be small and hence the time taken to mine the small conditional pattern base will be less.

Table 1
Example transaction dataset.

Transaction-ID	Transaction
1	102, 97, 99, 100, 103, 105, 109, 112
2	97, 98, 99, 102, 108, 109, 111
3	98, 102, 104, 106, 111
4	98, 99, 107, 115, 112
5	97, 102, 99, 101, 108, 112, 109, 110

Table 2
Frequency list.

Item-id	Frequency
102	4
99	4
109	3
112	3
97	3
98	3

Table 3
Grouping by MPFP.

Group	Members
1	102, 99
2	109, 112
3	97, 98

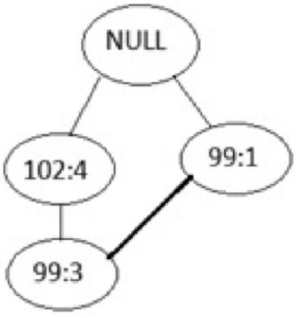

Fig. 1. MPFP: Group1 FP-Tree.

Fig. 1 shows the FP-Tree for group1, as constructed by the MPFP growth algorithm. The depth of the FP-Tree is 2. The length of item 102's prefix path is 0. Hence there is no frequent pattern generation for item 102. Item 99 has two paths in the tree, one with prefix path length of 1 and the other with prefix path length of 0. Hence, only one frequent pattern is generated for item 99.

Fig. 2 shows the FP-Tree for group2, as constructed by the MPFP growth algorithm. The depth of the FP-Tree is 4. The length of the prefix path for item 109 is 2. Item 112 has two paths in the tree, one with a prefix path length of 3 and the other with a prefix path length of 1. Hence, the size of the conditional pattern bases for items 109 and 112 is greater than the size of the conditional pattern base for group1's items 102 and 99. Hence, group2 will take more time than group1 to generate the frequent items.

Fig. 3 shows the FP-Tree for group3, as constructed by the MPFP growth algorithm. The depth of the FP-Tree is 5. Item 97 has two paths in the tree, one with prefix path length of 4 and the other with prefix path length of 3. Item 98 has three paths in the tree with prefix path lengths of 4, 2 and 1 respectively. Hence, the conditional pattern bases for items 97 and 98 are larger in size than the conditional pattern bases of items in group1 and group2. Therefore, group3 will take the longest time to generate the frequent patterns.

As stated by Ref. [6], the local FP growth algorithm running in the reducer phase of the second MapReduce job consumes 50% of the total

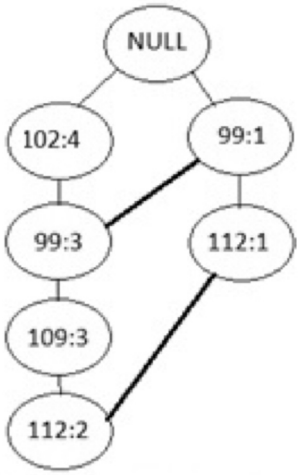

Fig. 2. MPFP: Group2 FP-Tree.

Fig. 3. MPFP: Group3 FP-Tree.

mining time. For the group with the less frequent items, the local FP growth algorithm will spend more time on tree construction and also more time in the growth phase. The time spent by the groups are not equivalent and hence they are not balanced. Though the local FP growth algorithm takes less time for some groups, the overall running time of the MPFP algorithm is extended by the local FP growth algorithm that takes the longest time to execute. This difference in running time is *due* to the load not being balanced among the reducers. Hence the local FP growth influences the overall performance of the MPFP algorithm. Any performance savings at this stage can result in significant performance gains. The time taken by the local FP growth in the reduce phase can be influenced by the grouping strategy.

4.2. Balanced PFP

Balanced PFP (BPFP) [6] attempts to balance the load among the reducers in the second MapReduce job of the PFP algorithm by using a load factor that measures the log of the location of the item in the reverse sorted frequency list. BPFP uses a priority queue for grouping items. This priority queue consumes more time for grouping the items.

5. Heuristic based load balancing

HBPFP aims to balance the load on all reducers performing the local FP growth algorithm based on a heuristic that accounts for all work done for every item. When the reducers' load is balanced, it will complete the local FP growth algorithm in almost an equivalent amount of time, therefore reducing the overall run time of the HBPFP algorithm.

Items with high frequencies occur closer to the root of an FP tree. Hence, with respect to the tree construction phase, more time is spent on less frequent items and less time is spent on more frequent items. For the items that occur towards the root of the FP-tree, the prefix paths are shorter and their conditional pattern bases are smaller. Sparse datasets will typically have more less frequent items, so grouping them with the more frequent items makes this algorithm especially suitable for sparse datasets.

Our heuristic approach to balance the groups, is to combine the less frequent items with the more frequent items. The balanced groups are formed from the reverse sorted frequency list as follows:

1. The top 'n' most frequent items are each placed in 'n' groups.
2. For the remaining items in the frequency list, the items are placed in the groups in a round robin fashion, starting from the least frequent item, that is, from the end of the frequency list.

Hence, less frequent items and more frequent items appear in the

Table 4
Grouping by heuristic balancing strategy.

Group	Members
1	102, 98
2	99, 97
3	109, 112

same group. Every group should have an almost equal share of most frequent items and least frequent items. We will demonstrate our load balancing strategy using the transactions presented in Table 1 and the frequency list presented in Table 2.

With the heuristic based balancing strategy, the groups will be formed as shown in Table 4. Figs. 4–6 shows the FP-Trees for the groups formed by heuristic based grouping strategy. As can be seen, the FP-Trees for all the groups have almost the same depth.

For group1 (Fig. 4), the depth of the FP-Tree is 5. The length of item 102's prefix path is 0. Hence there is no frequent pattern generation for item 102. Item 98 has three paths in the tree, with prefix path lengths of 4, 2 and 1 respectively. Though more work has to be done for mining frequent patterns for item 98, there are no frequent patterns to be generated for item 102. Hence the longer time spent for item 98 is balanced by the shorter time spent for item 102.

For group2 (Fig. 5), the depth of the FP-Tree is 5. Item 99 has two paths in the tree, with prefix path lengths of 1 and 0. Item 97 has two paths in the tree, with prefix path lengths of 4 and 3. The conditional pattern base of item 97 is larger than that of item 99. Hence, the longer time spent for item 97 is balanced by the shorter time spent for item 99.

For group3 (Fig. 6), the depth of the FP-Tree is 4. The length of the prefix path for item 109 is 2. Item 112 has two paths in the tree, with prefix path lengths of 4 and 1. The conditional pattern base of item 112 is larger than that of item 109. Hence, the longer time spent for item 112 is balanced by the shorter time spent for item 109. Hence, all groups take almost the same amount of time to generate frequent patterns. As can be seen from this example, our heuristic based grouping strategy groups the items into balanced groups such that the time taken to generate the frequent patterns is almost equivalent for all groups. Hence the load is balanced across the groups.

6. Implementation details of the Heuristic based balanced parallel FP growth algorithm

6.1. Grouping algorithm used for the Heuristic based balanced parallel FP growth algorithm

The grouping algorithm is presented in Algorithm 1. It uses a First In First Out (FIFO) queue of groups. A hash map is maintained which maps a group id to its member items. A reverse hash map is also maintained, which maps the items to their group ids. The hash map and reverse hash map help to efficiently look up an item's group id or the member items of a group. The first while loop (steps 3 to 9) performs step 1 mentioned in Section 4. It creates *n* groups and adds the top *n* items of the frequency list to these groups. The second while loop (steps 11 to 18) performs step 2 mentioned in Section 4, that is, starting from the end of the frequency list, it adds the remaining items to the groups in FIFO order.

Since the entire fList has to be scanned for grouping, the complexity of the grouping algorithm is O(n), that is, it is proportional to fList size. The BPFP algorithm, however, uses a priority queue for grouping and also traverses the entire fList. The complexity of their grouping algorithm is O(n*log(n)). By comparison of complexities, it can be seen that HBPFP's grouping algorithm is more efficient than the grouping algorithm of BPFP. The results section clearly demonstrates that the overall running time of HBPFP is less than that of BPFP. Also, the groups produced by HBPFP are more balanced than the groups produced by BPFP. Both BPFP and HBPFP store the groups in a map and thus have a complexity of O(1) for group lookup.

In MPFP, the fList items are grouped by using hash partitioning. For every item, the group id is determined by dividing the item position in

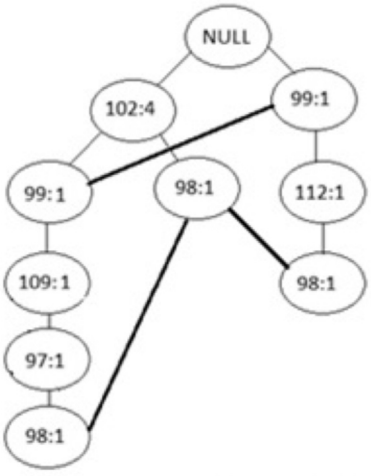

Fig. 4. HBPFP: Group1 FP-Tree.

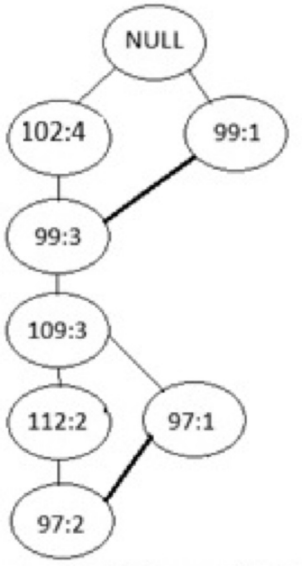

Fig. 5. HBPFP: Group2 FP-Tree.

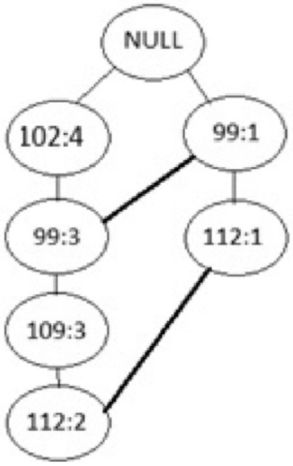

Fig. 6. HBPFP: Group3 FP-Tree.

the fList by the number of items per group. Hence, complexity of grouping in MPFP is O(1). Though MPFP does not spend much time in grouping, it creates unbalanced groups. This penalizes some of the reduce tasks by slowing down the entire MapReduce job.

Algorithm 1
Heuristic Based Grouping.

```
Input:  reverse sorted list of frequent items, fList
        number of groups to be created, numGroups
Output: hash map that maps group ids to member items, map
        reverse hash map that maps items to group ids, reverseMap

1:  i := 1;
2:  queue := createFIFOQueue(numGroups);
3:      while (i <= numGroups)
4:          begin
5:              queue.add(i,fList[i]);
6:              map.put(i,fList[i];
7:              reverseMap.put(fList[i],i);
8:              i++;
9:          end
10: i := fList.size();
11:     while(i >= numGroups)
12:         begin
13:             temp = queue.remove();
14:             queue.add(temp.groupId(), temp.items + fList[i]);
15:             map.put(temp.groupId(), temp.items + fList[i]);
16:             reverseMap.put(fList[i], temp.groupId());
17:             i--;
18:         end
```

6.2. FP-growth algorithm implementation for the Heuristic Based Balanced Parallel FP growth algorithm

The FP-Growth algorithm is implemented as a MapReduce job. It is similar to the Parallel FP Growth step of the MPFP algorithm. The fList and the map containing the groups are copied to all map tasks and reduce tasks using the distributed cache feature of the MapReduce framework. Every map and reduce task reads them from the distributed cache. The group map is used in the map task to determine the group id for a given item, and is used in the reduce task to determine the items that belong to a given group id.

The map tasks parse the transactions. They eliminate the transaction items not found in the fList and sort the remaining items in the same order as the fList. Starting from the most frequent transaction item, the map tasks determine the group id for each item and emit the group id as the key, and the transaction is emitted in the form of a transaction tree as the value. The map tasks emit a transaction only once for a group id. Hence all transactions that belong to the items of a group are sent to the same reducer. The reducer task combines all the transaction trees for a group and constructs the FP-Tree for the group items. This FP-Tree is complete with respect to the group items, hence it can only be mined for generating frequent itemsets of the group items. The FP-Tree is mined recursively and the frequent item sets for the group items are generated. The generated frequent itemsets are emitted as the value, with the group id as the key.

7. Experimental setup

HBPFP was implemented on an in-house Hadoop cluster that consists of three management nodes and six data nodes. The cluster runs Hadoop 2.6.0-CDH5.5.1 with JDK 1.7. It uses the MR2 (YARN) MapReduce framework where the six data nodes have node managers, and hence run the map and reduce tasks. The data nodes are identical in configuration, where each node has two Xeon E5-2650 processors and 128 GB RAM.

For this work, a transactional dataset based on spidered collection of web html documents from the Frequent Itemset Mining Dataset Repository [19] was used. This dataset, which can be considered a sparse dataset, was built from about 1.7 million web documents. The html tags and the most common words (stop words) were filtered out of the web documents. Each transaction represents a web document and the items in the transactions represent the set of distinct terms that appear in that document. The resulting dataset was 1.48 GB in size, containing 1,692,082 transactions with 5,267,656 distinct items. The maximal length of a transaction was 71,472.

The three algorithms, MPFP [8], BPFP [6] and our HBPFP, were executed on the webdocs dataset [18] for different minimum support values. They were compared for run-time performance, their ability to balance the load across the cluster as well as memory performance. The run-time performance is measured in terms of overall time taken to generate the frequent itemsets. The ability to balance the load is expressed by the standard deviation of the running time taken to process each group dependent transactional database. The memory performance is expressed by the data size that can be processed for a given memory space, and also the time taken for a given minimum support threshold and memory space.

8. Results and discussion

Results are presented, for the three algorithms, MPFP, BPFP, and HBPFP, in terms of run-time performance, load balancing, memory performance and scalability.

8.1. Run-time performance

The three algorithms were run for different minimum support thresholds, from 7.5% to 12%, in increments of 0.5%. The reduce memory size was set at 5,354 MB for all three algorithms. The grouping algorithm was dependent on the size of the fList, shown in Table 5, not the size of the transaction database.

From Table 5, it can be observed that, as the minimum support threshold decreases, the size of the fList increases. When the fList increases, more items are considered to be frequent. These need to be grouped and included in the transaction trees emitted as the intermediate values. Hence, a lower minimum support threshold results in an increase in the intermediate data size, which in turn increases the load on the reducers. The fList size, however, is negligible when compared to the size of the transaction database and the intermediate data size which has to be processed by the reducer phase.

Fig. 7 shows the average running times (in minutes) for HBPFP, MPFP and BPFP for the various minimum support values. The running time includes the time taken for creating the fList, grouping the items, constructing the group-specific FP-Trees and mining them. To get robust results for the average running times, each run was performed 10 times and averaged. The running time for the algorithms decrease as the minimum support increases. This is because, as the minimum support increases, the size of the frequency list, that is, the number of items for which the frequent patterns have to be mined, decreases. For all the different minimum support thresholds, HBPFP runs the fastest amongst the three PFP algorithms.

Table 5
fList Size for various Minimum Support Threasholds.

Minimum Support Threshold(%)	fList Size
7.5	396
8	364
8.5	331
9	307
9.5	283
10	262
10.5	245
11	221
11.5	204
12	189

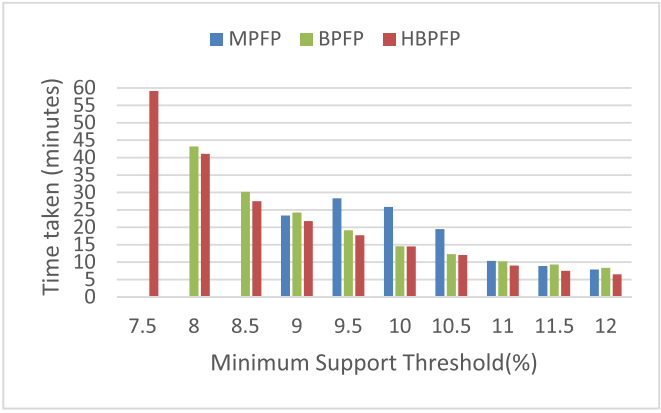

Fig. 7. Comparison of average running times of MPFP, BPFP and HBPFP algorithms.

As can be seen from Fig. 7, that we do not have data for MPFP for minimum support of 7.5%, 8% and 8.5%, and for BPFP for minimum support of 7.5%. Both MPFP and BPFP run out of memory when mining frequent itemsets with lower minimum support values because of the increased load on the reducers, which was not well-balanced. Some of the reducers needed to mine larger FP-Trees. The HBPFP algorithm runs successfully for all values of minimum support from 7.5% and above. This indicates that our HBPFP algorithm balances the load better than MPFP and BPFP. With HBPFP, none of its reducers are burdened with very large FP-Trees. The average percent improvement in running time of HBPFP over MPFP is 23.82%. On the average, BPFP runs faster than MPFP, but slower than HBPFP. HBPFP is 8.59% faster than BPFP.

8.2. Load balancing

To illustrate the load balancing performed by the three algorithms, the time taken for generating frequent itemsets, at a minimum support of 10%, for individual groups, 1 to 79 groups, was recorded, as shown in Fig. 8. The minimum support threshold was arbitrarily selected from Fig. 7, trying to strike a balance between best running time and lowest minimum support threshold.

Fig. 8 shows that MPFP is the most unbalanced, that is, the variation in running times for the different groups is high for MPFP. For BPFP, the variations in running times of different groups is less than that of MPFP. This is because BPFP does a better job of balancing the load of the groups than MPFP. When compared with the BPFP and MPFP, HBPFP has the least variation in running times for the different groups, demonstrating a more balanced load than the other two algorithms, no matter what group size is used.

Fig. 9 presents the standard deviation of the running times of all groups (averaged) for the three algorithms at different minimum support values. Again, compared to the BPFP and MPFP, for all minimum support values, HBPFP has the lowest standard deviation. Hence HBPFP distributes the load almost equivalently among all the groups, irrespective of the size of the group.

8.3. Memory performance

FP-growth is a memory intensive algorithm. A deep FP-Tree will require more memory, and mining a deep FP-Tree involves many recursive calls in the growth phase. This can generate a large number of pattern trees. The load on the reducers determines the memory requirements of the reducers. When the load on the reducers is not balanced, some reducers will have to construct and mine very large FP-Trees, while other reducers will have to only construct and mine small FP-trees. Reducers handling smaller FP-trees may complete the FP-growth mining successfully, while reducers handling larger FP-Trees may run out of memory and not be able to complete the job. A

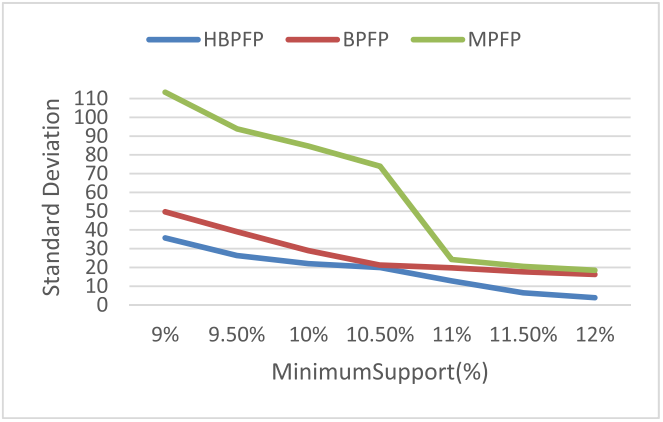

Fig. 9. Standard Deviation of Group Running Times (all groups averaged).

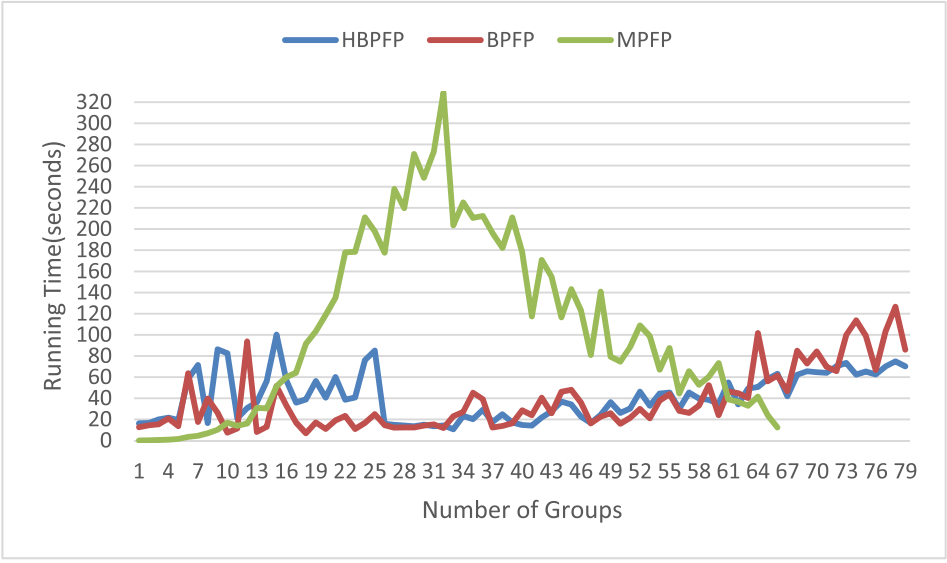

Fig. 8. Group Running Times for minimum support of 10%.

Table 6
Average Running Times for various Reduce Task Memory Size at different Minimum Support Threshholds.

Min. Support (%)	6.5	7	7.5	8	8.5	9	9.5	10	10.5	11	11.5	12
8 GB												
MPFP	–	–	–	89.85	73.35	43.53	35.54	29.87	25.40	14.50	12.70	10.70
BPFP	145.90	80.30	52.35	43.22	36.00	28.97	23.79	18.83	16.36	14.14	12.50	11.50
HBPFP	102.00	72.94	49.84	42.01	34.39	28.17	23.36	19.48	17.01	12.85	10.80	9.68
7 GB												
MPFP	–	–	–	121.12	87.75	37.84	30.97	25.79	21.91	13.26	11.67	9.80
BPFP	–	–	50.73	37.66	31.18	26.06	21.16	17.12	14.98	12.69	11.68	10.84
HBPFP	–	79.36	46.69	36.80	30.69	25.15	20.56	17.58	15.29	11.86	10.27	9.05
6 GB												
MPFP	–	–	–	–	–	38.74	31.26	26.22	22.17	13.46	11.55	10.05
BPFP	–	–	–	39.60	31.86	26.76	21.59	16.86	15.04	12.76	11.61	10.69
HBPFP	–	–	57.65	40.64	30.54	24.99	20.86	17.40	15.32	11.93	10.12	9.05
5 GB												
MPFP	–	–	–	–	–	69.13	38.61	27.46	22.73	13.15	11.74	9.98
BPFP	–	–	–	–	–	–	–	17.26	14.94	12.91	11.84	10.97
HBPFP	–	–	–	–	35.95	25.26	21.01	18.00	15.17	11.93	10.19	9.12
4 GB												
MPFP	–	–	–	–	–	–	–	–	–	10.00	8.98	7.91
BPFP	–	–	–	–	–	–	–	19.14	11.86	10.47	9.54	8.85
HBPFP	–	–	–	–	–	–	16.59	13.97	11.86	9.37	8.53	7.75
3 GB												
MPFP	–	–	–	–	–	–	–	–	–	11.34	9.04	7.52
BPFP	–	–	–	–	–	–	–	–	–	–	–	8.92
HBPFP	–	–	–	–	–	–	–	–	–	9.73	8.52	7.39

MapReduce job fails when one of its reducers fails. Thus the load on the reducers needs to be balanced.

To analyze memory performance, the three algorithms, MPFP, BPFP and HBPFP, were run with different reducer memory sizes at various minimum support thresholds. In Hadoop, the minimum memory for a reduce task is 1 GB (1024 MB), and the maximum size that can be allocated to a reduce task in our in-house Hadoop cluster is 8 GB (8192 MB). Hence we ran the three algorithms, MPFP, BPFP and HBPFP, varing the reduce task memory size from 8 GB to 1 GB in decrements of 1 GB, for different minimum support thresholds, 6.5%–12% in increments of 0.5%.

None of the algorithms ran for reduce task memory of 1 GB with our minimum support threshholds of up to 12%. Only HBPFP ran with reduce task memory of 2 GB for a minimum support threshold of 12%, but since the other two algorithms, MPFP and BPFP, did not run on a reduce task memory size of 2 GB either, we did not present in these results in Table 6. Table 6 presents the running times for the algorithms (in minutes) for the various reduce memory sizes, 8 GB–3 GB, in decrements on 1 GB, at the various minimum support threshholds.

From Table 6, we can note: (i) As the minimum support threshold is increased, the running time goes down for all algorithms; (ii) As the reduce memory size is increased, the running time goes up for all three algorithms, if the minimum support threshold is kept the same; in order to get faster running times, as the memory size is reduced, the minimum support needs to be increased too; (iii) HBPFP outperforms both BPFP and MPFP, both in terms of being able to run with lower reduce memory size and minimum support threshold. That is, of the three algorithms, HBPFP ran at the lowest minimum support threshold for each task memory size. BPFP performed the second best.

For example, for the reduce memory size of 8 GB, both BPFP and HBPFP successfully complete the jobs for the minimum support threshold of 6.5%, and HBPFP runs faster than BPFP for all minimum support thresholds except the 10% and 10.5% thresholds. MPFP cannot handle large datasets for minimum support thresholds of 6.5%, 7% and 7.5%, even at the 7 or 8 GB reduce task memory size. At the reduce task memory size of 7 GB, BPFP could not even handle large datasets of minimum support thresholds of 7%.

At the low 3 GB reduce task memory size, both HBPFP and MPFP ran successfully at a minimum support threshold of 11%. However, HBPFP ran faster than both BPFP and MPFP. BPFP could handle large datasets for minimum support thresholds of 11% and 11.5%.

So, from Table 6 it is also clear that the reduce task memory determines the data size that can be processed by the FP-growth algorithm in the reduce phase. As the reduce task memory increases, larger data sizes can be processed. These experiments show that HBPFP's load balancing technique distributes the load equivalently, such that individual groups can be processed with less memory. Hence, HBPFP not only improves the run time performance of the PFP algorithm, but also the space requirements.

8.4. Scalability

To demonstrate scalability, the three PFP based algorithms were executed on the AWS EMR cluster, after increasing the data size to 100 GB. The EC2 instance type m3.xlarge was chosen as it was best suited for general purpose applications. The Hadoop version used was the same as the Hadoop version of the home-grown cluster, i.e. Hadoop-2.6.0. A minimum support of 9% was chosen since this was the lowest minimum support that would run for all the three algorithms for this data size without throwing a heap size exception. The number of groups was kept at 79, that is, the same as the number of groups used while running the algorithms on the home-grown cluster. The number of reducers was set to be the same as the number of groups, that is, 79. 22 mappers were used.

As per the above specifications, the three algorithms, MPFP, BPFP and HBPFP, were run on various number of nodes: 6, 8, 10, 12, 14 and 16. Fig. 10 presents the running times (in minutes) for the different number of nodes in the cluster. It can be noted that HBPFP consistently had the lowest running times, and of course the running times reduced as the number of nodes increased. MPFP consistenly had the highest running times of all three algorithms.

8. Conclusions

In a distributed and parallel environment like Hadoop's MapReduce framework, balancing the load on all the cluster nodes plays an important role on the overall performance of the algorithm. Our results show that our heuristic based strategy, which has a lower order of complexity, balances the load among reducers better than the mahout implementation of PFP and BPFP's load factor based balancing strategy, especially for large sparse datasets. The total running time of our algorithm is found to be less than that of both MPFP and BPFP. In addition to scaling well with cluster size, HBPFP utilizes cluster resources like task memory more efficiently. The HBPFP algorithm processes the largest dataset size for a

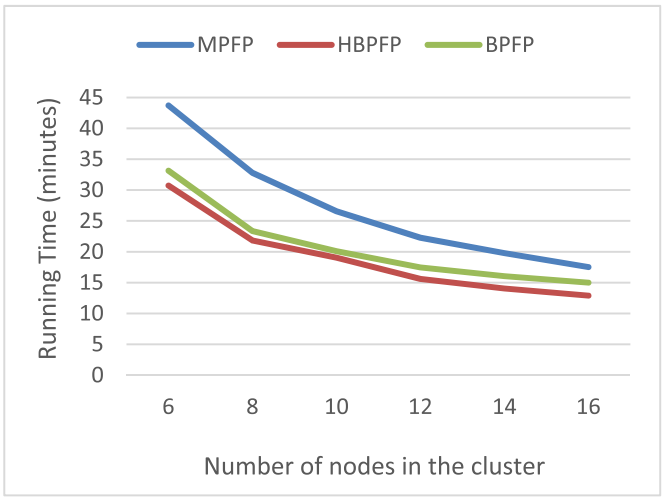

Fig. 10. Average Running Time (in minutes) Vs Number of cluster nodes.

given memory space and runs fastest for any given dataset size and memory space. The final recommendation would be to use HBPFP for sparse datasets.

Credit author statement

Dr. Sikha Bagui and Keerthi Devulapalli conceptualized the paper. Keerthi did most of the programming and both Dr. Bagui and Keerthi were involved in the writing of the paper. Dr. Coffey was involved in the editing of the paper.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work has been partially supported by the Askew Institute at The University of West Florida.

References

[1] Li H, Wang Y, Zhang D, Zhang M, Chang EY. Pfp: parallel fp-growth for query recommendation. In: Proceedings of the 2008 ACM conference on Recommender systems. ACM; 2008. p. 107–14.
[2] Agrawal R, Srikant R. "Fast algorithms for mining association rules", InProc. 20th int. conf. very large data bases. VLDB 1994 Sep 12;1215:487–99.
[3] Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. In: ACM Sigmod Record, vol. 29; 2000. p. 1–12. 2.
[4] Apache hadoop. http://hadoop.apache.org/.
[5] Perfect balance. https://docs.oracle.com/cd/E41604_01/doc.22/e41241/balance.htm#BIGUG279.
[6] Zhou L, Zhong Z, Chang J, Li J, Huang J, Feng S. "Balanced parallel fp-growth with mapreduce," in information computing and telecommunications (YC-ICT), 2010 IEEE youth conference on. IEEE; 2010. p. 243–6.
[7] Apache mahout. http://mahout.apache.org.
[8] Mahout PFP. http://grepcode.com/file/repo1.maven.org/maven2/org.apache.mahout/mahout-core/0.9/org/apache/mahout/fpm/pfpgrowth/package-info.java?av=f.
[9] Zaïane OR, El-Hajj M, Lu P. Fast parallel association rule mining without candidacy generation. In: Data mining, ICDM 2001, proceedings IEEE international conference; 2001. p. 665–8.
[10] Ghoting A, Buehrer G, Parthasarathy S, Kim D, Nguyen A, Chen YK, Dubey P. Cache-conscious frequent pattern mining on a modern processor. In: Proceedings of the 31st international conference on Very large data bases; 2005. p. 577–88.
[11] Jin R, Yang G, Agrawal G. Shared memory parallelization of data mining algorithms: techniques, programming interface, and performance. IEEE Trans Knowl Data Eng 2005;17(1):71–89.
[12] Liu L, Li E, Zhang Y, Tang Z. Optimization of frequent itemset mining on multiple-core processor. In: Proceedings of the 33rd international conference on Very large data bases; 2007. p. 1275–85.
[13] Arour K, Belkahla A. "Frequent pattern-growth algorithm on multicore CPU and GPU processors". CIT. J Comput Inf Technol 2014;22(3):159–69.
[14] Moonesinghe HDK, Chung MJ, Tan PN. Fast parallel mining of frequent itemsets. Michigan State University; 2006.
[15] Buehrer G, Parthasarathy S, Tatikonda S, Kurc T, Saltz J. Toward terabyte pattern mining: an architecture-conscious solution. In: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming; 2007. p. 2–12.
[16] Xia DA, Rong ZH, Zhou YA, Li Y, Shen Y, Zhang Z. A novel parallel algorithm for frequent itemsets mining in massive small files datasets. ICIC Express Lett B: Appl 2014;5(2):459–66.
[17] Riondato M, DeBrabant JA, Fonseca R, Upfal E. Parma: a parallel randomized algorithm for approximate association rules mining in mapreduce. In: Proceedings of the 21st ACM international conference on Information and knowledge management. ACM; 2012. p. 85–94.
[18] Vavilapalli VK, Murthy AC, Douglas C, Agarwal S, Konar M, Evans R, Graves T, Lowe J, Shah H, Seth S, Saha B. Apache hadoop yarn: yet another resource negotiator. In: Proceedings of the 4th annual symposium on cloud computing; 2013 Oct 1. p. 5.
[19] FIMI. http://fimi.cs.helsinki.fi/data.
[20] Chen GP, Yang YB, Zhang Y. "Mapreduce-based balanced mining for closed frequent Itemset" in Web Services (ICWS). In: 2012 IEEE 19th international conference on. IEEE; 2012, June. p. 652–3.
[21] Xun Y, Zhang J, Qin X, Zhao X. FiDoop-DP: data partitioning in frequent itemset mining on hadoop clusters. IEEE Trans Parallel Distr Syst 2017;28(1):101–14.
[22] Moens S, Aksehirli E, Goethals B. Frequent itemset mining for Big data. Proc Big Data 2013;111–118.
[23] Doung, K-C., Bamba, M., Giacometti, A., Li, D., Soulet, A., & Vrain, C. "MapFIM+: memory aware parallelized frequent itemset mining in very large datasets," Transactions on large-scale data- and knowledge-centered systems XXXiX. Lecture notes in computer science, vol. 11310, Springer, Berlin, Heidelberg.