Optimizing the Data-Process Relationship for Fast Mining of Frequent Itemsets in MapReduce

Saber Salah, Reza Akbarinia, Florent Masseglia

Optimizing the Data-Process Relationship For Fast Mining of Frequent Itemsets in MapReduce

Saber Salah*, Reza Akbarinia, and Florent Masseglia

Inria & LIRMM
Zenith team - Univ. Montpellier
France
first.last@inria.fr

Abstract. Despite crucial recent advances, the problem of frequent itemset mining is still facing major challenges. This is particularly the case when: i) the mining process must be massively distributed and; ii) the minimum support ($MinSup$) is very low. In this paper, we study the effectiveness and leverage of specific data placement strategies for improving parallel frequent itemset mining (PFIM) performance in MapReduce, a highly distributed computation framework. By offering a clever data placement and an optimal organization of the extraction algorithms, we show that the itemset discovery effectiveness does not only depend on the deployed algorithms. We propose ODPR (Optimal Data-Process Relationship), a solution for fast mining of frequent itemsets in MapReduce. Our method allows discovering itemsets from massive datasets, where standard solutions from the literature do not scale. Indeed, in a massively distributed environment, the arrangement of both the data and the different processes can make the global job either completely inoperative or very effective. Our proposal has been evaluated using real-world data sets and the results illustrate a significant scale-up obtained with very low $MinSup$, which confirms the effectiveness of our approach.

1 Introduction

With the availability of inexpensive storage and the progress that has been made in data capture technology, several organizations have set up very large databases, known as Big Data. This includes different data types, such as business or scientific data [1], and the trend in data proliferation is expected to grow, in particular with the progress in networking technology. The manipulation and processing of these massive data have opened up new challenges in data mining [2]. In particular, frequent itemset mining (FIM) algorithms have shown several flaws and deficiencies when processing large amounts of data. The problem of mining huge amounts of data is mainly related to the memory restrictions as well as the principles and logic behind FIM algorithms themselves [3].

In order to overcome the above issues and restrictions in mining large databases, several efficient solutions have been proposed. The most significant solution required to rebuild and design FIM algorithms in a parallel manner relying on a specific programming model such as MapReduce [4]. MapReduce is one of the most popular solutions

* This work has been partially supported by the Inria Project Lab Hemera.

for big data processing [5], in particular due to its automatic management of parallel execution in clusters of commodity machines. Initially proposed in [6], it has gained increasing popularity, as shown by the tremendous success of Hadoop [7], an open-source implementation.

The idea behind MapReduce is simple and elegant. Given an input file, and two map and reduce functions, each MapReduce job is executed in two main phases. In the first phase, called map, the input data is divided into a set of splits, and each split is processed by a map task in a given worker node. These tasks apply the map function on every key-value pair of their split and generate a set of intermediate pairs. In the second phase, called reduce, all the values of each intermediate key are grouped and assigned to a reduce task. Reduce tasks are also assigned to worker machines and apply the reduce function on the created groups to produce the final results.

Although MapReduce refers as an efficient setting for FIM implementations, most of parallel frequent itemset mining (PFIM) algorithms have brought same regular issues and challenges of their sequential implementations. For instance, invoking such best PFIM algorithm with very low minimum support ($MinSup$) could exceed available memory. Unfortunately, dealing with massive datasets (up to terabytes of data) implies working with very low supports since data variety lowers item frequencies. Furthermore, if we consider a FIM algorithm which relies on a candidate generation principle, its parallel version would remain carrying the same issues as in its sequential one. Therefore, covering the problem of FIM algorithms does not only involve the distribution of computations over data, but also should take into account other factors.

Interestingly and to the best of our knowledge, there has been no focus on studying data placement strategies for improving PFIM algorithms in MapReduce. However, as we highlight in this work, the data placement strategies have significant impacts on PFIM performance. In this work, we identify, investigate and elucidate the fundamental role of using such efficient strategies for improving PFIM in MapReduce. In particular, we take advantage of two data placement strategies: Random Transaction Data Placement (RTDP) and Similar Transaction Data Placement (STDP). In the context of RTDP, we use a random placement of data on a distributed computational environment without any data constraints, to be consumed by a particular PFIM algorithm. However, in STDP, we use a similarity-based placement for distributing the data around the nodes in the distributed environment. By leveraging the data placement strategies, we propose ODPR (Optimal Data-Process Relationship), a new solution for optimizing the global extraction process. Our solution takes advantage of the best combination of data placement techniques and the extraction algorithm.

We have evaluated the performance of our solution through experiments over ClueWeb and Wikipedia datasets (the whole set of Wikipedia articles in English). Our results show that a careful management of the parallel processes along with adequate data placement, can dramatically improve the performance and make a big difference between an inoperative and a successful extraction.

The rest of this paper is organized as follows. Section 2 gives an overview of FIM problem and Section 3 gives the necessary background on MapReduce and some basic FIM algorithms. In Section 4, we propose our techniques of data placement for an efficient execution of PFIM algorithms. Section 5 reports on our experimental validation

over synthetic and real-world data sets. Section 6 discusses related work, and Section 7 concludes.

2 Problem Definition

The problem of frequent itemset mining has been initially proposed in [8], and then numerous algorithms have been proposed to solve it. Here we adopt the notations used in [8].

Itemset: Let $I = \{i_1, i_2, ..., i_n\}$ be a set of literals called $items$. An $Itemset\ X$ is a set of items from I, i.e. $X \subseteq I$. The $size$ of the itemset X is the number of items in it.
Transaction: A transaction T is a set of elements such that $T \subseteq I$ and $T \neq \emptyset$. A transaction T supports the item $x \in I$ if $x \in T$. A transaction T supports the $itemset\ X \subseteq I$ if it supports any item $x \in X$, i.e. $X \subseteq T$.
Database: A database D is a set of transactions.
Support: The $support$ of the $itemset\ X$ in the database D is the number of transactions $T \in D$ that contain X.
Frequent Itemset: An $itemset\ X \subseteq I$ is $frequent$ in D if its $support$ is equal or higher than a ($MinSup$) threshold.

The goal of FIM is as follows: given a database D and a user defined minimum support $MinSup$, return all frequent itemsets in D.

Example 1. Let us consider database D with 4 transactions as shown in Table 2. With a minimum support of 3, there will be no frequent items (and no frequent itemsets). With a minimum support of 2, there will be 6 frequents itemsets:$\{(a), (b), (e), (f), (ab), (ef)\}$.

TID	Transaction
T_1	a, b, c
T_2	a, b, d
T_3	e, f, g
T_4	d, e, f

Table 1. Database D

In this paper, we consider the specific problem of PFIM, where the data set is distributed over a set of computation nodes. We consider MapReduce as a programming framework to illustrate our approach, but we believe that our proposal would allow to obtain good performance results in other parallel frameworks too.

3 Requirements

In this section, we first describe briefly MapReduce and its working principles. Then, we introduce some basic FIM algorithmic principles which we use in our PFIM algorithms.

3.1 MapReduce and job execution

Each MapReduce job includes two functions: map and reduce. For executing the job, we need a master node for coordinating the job execution, and some worker nodes for executing the map and reduce tasks. When a MapReduce job is submitted by a user to the cluster, after checking the input parameters, e.g., input and output directories, the input *splits* (blocks) are computed. The number of input splits can be personalized, but typically there is one split for each 64MB of data. The location of these splits and some information about the job are submitted to the master. The master creates a job object with all the necessary information, including the map and reduce tasks to be executed. One map task is created per input split.

When a worker node, say w, becomes idle, the master tries to assign a task to it. The map tasks are scheduled using a locality-aware strategy. Thus, if there is a map task whose input data is kept on w, then the scheduler assigns that task to w. If there is no such task, the scheduler tries to assign a task whose data is in the same rack as w (if any). Otherwise, it chooses any task.

Each map task reads its corresponding input split, applies the map function on each input pair and generates *intermediate key-value* pairs, which are firstly maintained in a buffer in main memory. When the content of the buffer reaches a threshold (by default 80% of its size), the buffered data is stored on the disk in a file called spill. Once the map task is completed, the master is notified about the location of the generated intermediate key-values.

In the reduce phase, each intermediate key is assigned to one of the reduce workers. Each reduce worker retrieves the values corresponding to its assigned keys from all the map workers, and merges them using an external merge-sort. Then, it groups pairs with the same key and calls the reduce function on the corresponding values. This function will generate the final output results. When, all tasks of a job are completed successfully, the client is notified by the master.

During the execution of a job, there may be idle nodes, particularly in the reduce phase. In Hadoop, these nodes may be used for *speculative* task execution, which consists in replicating the execution of incomplete slow tasks in those nodes. When one of the replicated tasks gets complete, its results are kept and the rest of copies are stopped and their results discarded.

3.2 PFIM

One of the primordial FIM algorithms is Apriori [8]. This algorithm starts mining the database D by figuring out frequent items of size one, say L_1. Then, builds the potential frequent itemsets of size two C_2 by joining items in L_1. The algorithm tests the $support$ of each C_2 element in D, and returns a list of frequent itemsets L_2. The mining process is carried out until there is no more frequent itemset in D. The main drawback of Apriori is the size of intermediate itemsets that need to be generated. Actually, with itemsets having a maximum length of n, Apriori needs to compute n generation of candidates, each being supersets of the previous frequent itemsets. Usually, the number of intermediate itemsets grows follows a normal distribution according to the generation number. In other words, the number of candidates reaches its higher number in the middle of

the process. A straightforward implementation of this algorithm in MapReduce is very easy since each database scan is replaced by a MapReduce job for candidate support counting. However, the performances are very bad mainly because intermediate data have to be communicated to each mapper.

In the context of investigating PFIM in MapReduce and the effect of data placement strategies, we need to briefly describe the SON [9] algorithm that simplifies the mining process by dividing the FIM problem into two steps, and this makes it very suitable for being used in MapReduce. The steps of SON are as following:

Step 1: It divides the input database D into $|P| = n$ chunks where $P = \{p_1, p_2, ..., p_n\}$. Then, it mines each data chunk in the memory, based on a $localMinSup$ and given FIM algorithm. Thus, the first step of SON algorithm is devoted to determine a list of local frequent itemsets (LFI).
Step 2: Based on the first step, the algorithm filters the list of LFI by comparing them against the entire database D using a $globalMinSup$. Then, it returns a list of global frequent itemsets (GFI) which is a subset of LFI.

As stated in the first step of SON, a specific FIM algorithm can be applied to mine each data chunk. In this work, we have implemented and tested different algorithms for this step. The first one is Apriori as described above. The second one is CDAR [10], which relies on the following mining principle:

Step 1: The algorithm divides the database D into $|P| = n$ data partitions, $P = \{p_1, p_2, ..., p_i, ...p_n\}$. Each partition p_i in P only holds transactions whose length is i, where the length of a transaction is the number of items in it.
Step 2: Then, CDAR starts mining the data partitions according to transaction lengths in decreasing order. A transaction in each partition accounts for an itemset. If a transaction T is frequent in partition p_{i+1} then, it will be stored in a list of frequent itemsets L, otherwise, CDAR stores T in a temporary data structure $Temp$. Then, after checking the frequency of all T in p_{i+1}, CDAR generates i subsets of all T in $Temp$ and adds them to partition p_i. The same mining process is carried out until visiting all partitions $p_i \subset D$. Before, counting the $support$ of a transaction T, CDAR checks its inclusion in L, and if it is included, then CDAR does not consider T, as it is already in L which is considered as frequent.

4 Optimal Data-Process Relationship

Let us now introduce our PFIM architecture, called Parallel Two Steps (P2S), which is designed for data mining in MapReduce. From the mining point of view, P2S is inspired from SON algorithm [9]. The main reason behind opting SON as a reference to P2S is that a parallel version of the former algorithm does not require costly overhead between mappers and reducers. However, as illustrated by our experiments in Section 5, a straightforward implementation of SON in MapReduce would not be the best solution for our research problem. Therefore, with P2S, we propose new solutions for PFIM mining, within the "two steps" architecture.

The principle of P2S is drawn from the following observation. Dividing a database D into n partitions $p_1, p_2, ..., p_n$, where $\cup p_i = D$, $i = 1...n$

$$GFI \subseteq \cup LFI \tag{1}$$

where GFI denotes global frequent itemsets and LFI refers to local frequent itemsets. This particular design allows it to be easily parallelized in two teps as follow:

Job 1: Each mapper takes a data split, and performs particular FIM algorithm. Then, it emits a list of local frequent itemsets to the reducer
Job 2: Takes an entire database D as input, and filters the global frequent itemsets from the list of local frequent itemsets. Then, it writes the final results to the reducer.

P2S thus divides the mining process into two steps and uses the dividing principle mentioned above. As one may observe from its pseudo-code, given by Algorithm 1, P2S is very well suited for MapReduce.

Algorithm 1: P2S

Input: Database D and $MinSup\ \delta$
Output: Frequent Itemsets

```
1  //Map Task 1
2  map( key:Null : K1, value = Whole Data Split: V1 )
3      - Determine a local MinSup ls from V1 based on δ
4      - Perform a complete FIM algorithm on V1 using ls
5      emit (key: local frequent itemset, value: Null)
6  //Reduce Task 1
7  reduce( key:local frequent itemset, list(values) )
8      emit (key,Null)
9  //Map Task 2
10 Read the list of local frequent itemsets from Haddop Distributed Cache LFI once
11 map( key:line offset : K1, value = Database Line: V1 )
12     if an itemseti ∈ LFI and i ⊆ V1 then
13         key ← i
14     emit (key:i, value: 1)
15 //Reduce Task 2
16 reduce( key:i, list(values) )
17     sum ← 0 while values.hasNext() do
18         sum+ = values.next().get()
19     if sum >= δ then
20         emit (key:i, value: Null)
```

The first MapReduce job of P2S consists of applying specific FIM algorithm at each mapper based on a local minimum support ($localMinSup$), where the latter is computed at each mapper based on $MinSup\ \delta$ percentage and the number of transactions

of the split being processed. At this stage of P2S, the job execution performance mainly depends on a particular data placement strategy (i.e. RDTP or STDP). This step is done only once and the resulting placement remains the same whatever the new parameters given to the mining process (*e.g.* $MinSup$ δ, local FIM algorithm, etc.). Then P2S determines a list of local frequent itemsets LFI. This list includes the local results of all data splits found by all mappers. The second step of P2S aims to deduce a global frequent itemset GFI. This step is carried out relying on a second MapReduce job. In order to deduce a GFI list, P2S filters the LFI list by performing a global test of each local frequent itemset. At this step, each mapper reads once the list of local frequent itemset stored in Hadoop Distributed Cache. Then, each mapper takes a transaction at a time and checks the inclusion of its itemsets in the list of the local frequent itemset. Thus, at this map phase of P2S algorithm, each mapper emits all local frequent itemsets with their complete occurrences in the whole database (i.e. key: itemset, value: 1). The reducer of the second P2S step, simply computes the sum of the count values of each key (i.e. local frequent itemset) by iterating over the value list of each key. Then, the reducer compares the number of occurrences of each local frequent itemset to $MinSup$ δ, if it is greater or equal to δ, then, the local frequent itemset is considered as a global frequent itemset and it will be written to the Hadoop distributed file system. Otherwise, the reducer discards the key (i.e. local frequent itemset).

Theoretically, based on the inner design principles of P2S algorithm, different data placements would have significant impacts on its performance behavior. In particular, the performance of P2S algorithm at its first MapReduce job, and specifically at the mapper phase, strongly depends on RDTP or STDP used techniques. That is due to the sensitivity of the FIM algorithm being used at the mappers towards its input data.

The goal of this paper is to provide the best combination of both data placement and local algorithm choice in the proposed architecture. In Section 4.1, we develop two data placement strategies and explain more their role in the overall performances.

4.1 Data Placement Strategies

The performance of PFIM algorithms in MapReduce may strongly depend on the distribution of the data among the workers. In order to illustrate this issue, consider an example of a PFIM algorithm which is based on a candidate generation approach. Suppose that most of the workload including candidate generation is being done on the mappers. In this case, the data split or partition that holds most lengthy frequent itemsets would take more execution time. In the worst case, the job given to that specific mapper would not complete, making the global extraction process impossible. Thus, despite the fairly automatic data distribution by Hadoop, the computation would depend on the design logic of PFIM algorithm in MapReduce.

Actually, in general, FIM algorithms are highly susceptible to the data sets nature. Consider, for instance, the Apriori algorithm. If the itemsets to be extracted are very long, it will be difficult for this algorithm to perform the extraction. And in case of very long itemsets, it is even impossible. This is due to the fact that Apriori has to enumerate each subset of each itemset. The longer the final itemset, the larger the number of subsets (actually, the number of subsets grows exponentially). Now let us consider Job 1, mentionned above. If a mapper happens to contain a subset of D that will lead to

lengthy local frequent itemsets, then it will be the bottleneck of the whole process and might even not be able to complete. Such a case would compromise the global process.

On the other hand, let us consider the same mapper, containing itemsets with the same size, and apply the CDAR algorithm to it. Then CDAR would rapidly converge since it is best suited for long itemsets. Actually, the working principle of CDAR is to first extract the longest patterns and try to find frequent subsets that have not been discovered yet. Intuitively, grouping similar transactions on mappers, and applying methods that perform best for long itemsets seems to be the best choice. This is why a placement strategy, along with the most appropriate algorithm, should dramatically improve the performances of the whole process.

From the observations above, we claim that optimal performances depend on a particular care of massive distribution requirements and characteristics, calling for particular data placement strategies. Therefore, in order to boost up the efficiency of some data sensitive PFIM algorithms, P2S uses different data placement strategies such as *Similar Transaction Data Placement (STDP)* and *Random Transaction Data Placement (RTDP)*, as presented in the rest of this section.

RTDP Strategy RTDP technique merely refers to a random process for choosing bunch of transactions from a database D. Thus, using RTDP strategy, the database is divided into n data partitions $p_1, p_2, ..., p_n$ where $\cup p_i = D$, $i = 1...n$. This data placement strategy does not rely on any constraint for placing such bunch of transactions in same partition p.

STDP Strategy Unlike RTDP data placement strategy, STDP relies on the principle of similarity between chosen transactions. Each bucket of similar transactions is mapped to the same partition p. Therefore, the database D is split into n partitions and $\cup p_i = D$, $i = 1...n$.

In STDP, each data split would be more homogeneous, unlike the case of using RDTP. More precisely, by creating partitions that contain similar transactions, we increase the chance that each partition will contain frequent local itemset of high length.

4.2 Data Partitioning

In STDP, data partitioning using similarities is a complex problem. A clustering algorithm may seem appropriate for this task. However, we propose a graph data partitioning mechanism that will allow a fast execution of this step, thanks to existing efficient algorithms for graphs partitioning such as Min-Cut [11]. In the following, we describe how transaction data can be transformed into graph data for doing such partitioning.

- First, for each unique $item$ in D, we determine the list of transactions L that contain it. Let D' be the set of all transaction lists L.
- Second, we present D' as a graph $G = (V, E)$, where V denotes a set of vertices and E is a set of edges. Each transaction $T \in D$ refers to a vertex $v_i \in G$ where $i = 1...n$. The weight w of an edge that connects a pair of vertices $p = (v_i, v_j)$ in G equals to the number of common items between the transactions representing v_i and v_j.

– Then, after building the graph G, a Min-Cut algorithm is applied in order to partition D'.

In the above approach, the similarity of two transactions is considered as the number of their common items, i.e. the size of their intersection. In order to illustrate our graph partitioning technique, let us consider a simple example as follows.

Example 2. Let us consider D, the database from Table 2. We start by mapping each item in D to its transactions holder. As illustrated in the table of figure 4.2, T_1 and T_2 have 2 common itims, likewise, T_3 and T_4 have 2 common items, while the intersection of T_2 and T_3 is one. The intersection of transactions in D' refers to the weight of their edges. In order to partition D', we first build a graph G from D' as shown in Figure 4.2. Then, the algorithm Min-Cut finds a minimum cut in G (red line in Figure 5), which refers to the minimum capacity in G. In our example, we created two partitions: $Partition_1 =< T_1, T_2 >$ and $Partition_2 =< T_3, T_4 >$.

TID	Transaction
T_1	a, b, c
T_2	a, b, e
T_3	e, f, g
T_4	d, e, f

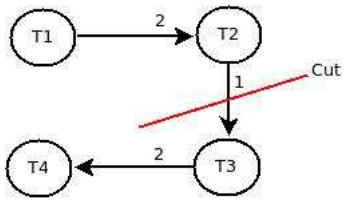

Fig. 1. Transactions of a database (left) & Graph representation of the database (right)

We have used a particular graph partitioning tool namely PaToH [12] in order to generate data partitions. The reason behind opting for Patoh lies in its set of configurable properties, e.g. the number of partitions and the partition load balance factor.

Based on the architecture of P2S and the data placement strategies we have developped and efficiently designed two FIM mining algorithms. Namely Parallel Two Steps CDAR (P2SC) and Parallel Two Steps Apriori (P2SA) depending on the itemset mining algorithm implemented for itemset mining on the mapper, in the first step of P2S. These two algorithms are highly data-sensitive PFIM algorithms.

For instance, if we consider P2SC as a P2S algorithm with STDP strategy, its performance would not be the same as we feed it with RDTP. Because relying on STDP, each split of data fed to such a mapper holds similar transactions, thus, there is less generation of transaction subsets. These expectations correspond to the intuition given in subsection 4.1. The impact of different data placement strategies will be better observed and illustrated through out experimental results as shown in section 5.

As shown by our experimental results in Section 5, P2S has given the best performance when instanciated with CDAR along with STDP strategy.

5 Experiments

To assess the performance of our proposed mining approach and investigate the impact of different data placement strategies, we have done an extensive experimental evaluation. In Section 5.1, we depict our experimental setup, and in Section 5.2 we investigate and discuss the results of our experiments.

5.1 Experimental Setup

We implemented our P2S principle and data placement strategies on top of Hadoop-MapReduce, using Java programming language. As mining algorithms on the mappers, we implemented Apriori as well as CDAR. For comparison with PFP-Growth [13], we adopted the default implementation provided in the Mahout [14] machine learning library (Version 0.7). We denote by P2Sx-R and P2Sx-S the use of our P2S principle with STPD (P2Sx-S) or RTPD (P2Sx-R) strategy for data placement, where local frequent itemsets are extracted by means of the 'x' algorithm. For instance, P2SA-S means that P2S is executed on data arranged according to STPD strategy, with Apriori executed on the mappers for extracting local frequent itemsets. MR-Apriori is the straightforward implementation of Apriori in MapReduce (one job for each length of candidates, and database scans for support counting are replaced by MapReduce jobs). PApriori does not use any particular data placement strategy. To this end, we just opted to test the algorithm with a RTDP data placement strategy for a comparison sake. Eventually, the instance of P2S architecture with Apriori exploited for local frequent itemset mining on the mappers and data arranged according to the RTPD strategy has to be considered as a straightforward implementation of SON. Therefore, we consider this version of P2S being the original version of SON in our experiments.

We carry out all our experiments based on the Grid5000 [15] platform, which is a platform for large scale data processing. We have used a cluster of 16 and 48 machines respectively for Wikipedia and ClueWeb data set experiments. Each machine is equipped with Linux operating system, 64 Gigabytes of main memory, Intel Xeon $X3440$ 4 core CPUs, and 320 Gigabytes SATA II hard disk.

To better evaluate the performance of ODPR and the impact of data placement strategies, we used two real-world datasets. The first one is the 2014 English wikipedia articles [16] having a total size of 49 Gigabytes, and composed of 5 millions articles. The second one is a sample of ClueWeb English dataset [17] with size of 240 Gigabytes and having 228 millions articles. For each dataset we performed a data cleaning task, by removing all English stop words from all articles and obtained a dataset where each article accounts for a transaction (where items are the corresponding words in the article) to each invoked PFIM algorithm in our experiments.

We performed our experiments by varying the $MinSup$ parameter value for each algorithm along with particular data placement strategy. We evaluate each algorithm based on its response time, in particular, when $MinSup$ is very low.

5.2 Performance Results

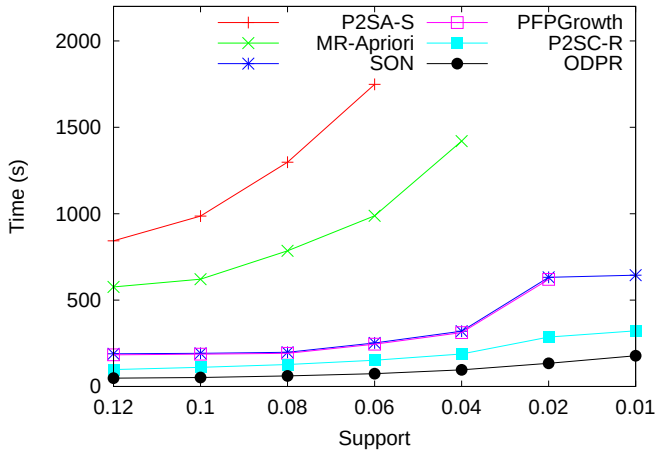

Fig. 2. All algorithms executed on the whole set of Wikipedia articles in English

Figures 2 and 3 report our results on the whole set of Wikipedia articles in English. Figures 2 gives a complete view on algorithms performances for a support varying from 0.12% to 0.01%. We see that MR-Apriori runtime grows exponentially, and gets quickly very high compared to other presented PFIM algorithms. In particular, this exponential runtime growth reaches its highest value with 0.04% threshold. Below this threshold, MR-Apriori needs more resources (e.g. memory) than what exists in our tested machines, so it is impossible to extract frequent patterns with this algorithm. Another interesting observation is that P2SA-S, i.e. the two step algorithm that use Apriori as a local mining solution, is worse that MR-Apriori. This is an important result, since it confirms that a bad choice of data-process relationship compromises a complete analytics process and makes it inoperative. Let us now consider the set of four algorithms that scale. The less effective are PFPGrowth and P2SA-R. It is interesting to see that two very different algorithmic schemes (PFPGrowth is based on the pattern tree principle and P2SA-R is a two steps principle with Apriori as a local mining solution with no specific care to data placement) have similar performances. The main difference being that PFPGrowth exceeds the available memory below 0.02%. Eventually, P2SC-R and ODPR give the best performances, with an advantage for ODPR.

Figure 3 focuses on the differences between the three algorithms that scale in Figure 2. The first observation is that P2SA-R is not able to provide results below 0.008%. Regarding the algorithms based on the principle of P2S, we can observe a very good performance for ODPR thanks to its optimization between data and process relationship. These results illustrate the advantage of using a two steps principle where an adequate data placement favors similarity between transactions, and the local mining algorithm does better on long frequent itemsets.

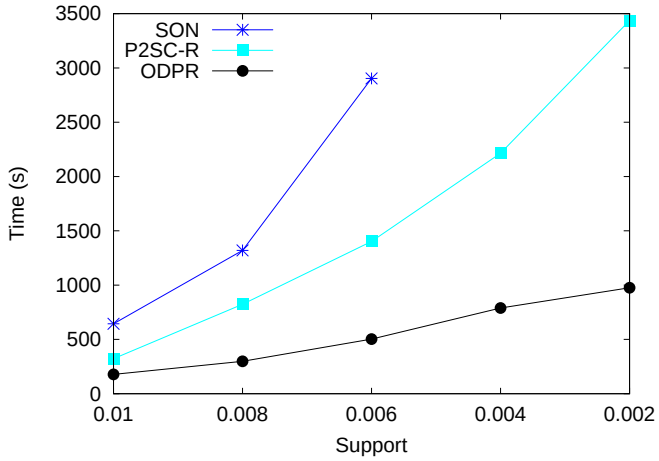

Fig. 3. A focus on algorithms that scale on Wikipedia articles in English

In Figure 4, similar experiments have been conducted on the ClueWeb dataset. We observe that the same order between all algorithms is kept, compared to Figures 2 and 3. There are two bunches of algorithms. One, made of P2SA-S and MR-Apriori which cannot reasonably applied to this dataset, whatever the minimum support. In the other bunch, we see that PFPGrowth suffers from the same limitations as could be observed on the Wikipedia dataset in Figure 2, and it follows a behavior that is very similar to that of P2SA-R, until it becomes impossible to execute.

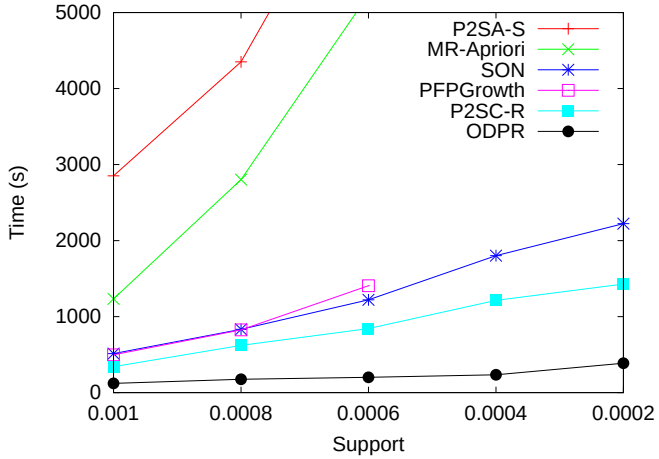

Fig. 4. Experiments on ClueWeb data

On the other hand, P2SC-R and ODPR are the two best solutions, while ODPR is the optimal combination of data placement and algorithm choice for local extraction, provinding the best relationship between data and process.

6 Related Work

In data mining literature, several efforts have been made to improve the performance of FIM algorithms [18], [19], [20]. However, due to the trend of the explosive data growth, an efficient parallel design of FIM algorithms has been highly required. There have been many solutions proposed to design most of FIM algorithms in a parallel manner [9], [13].

FP-Growth algorithm [20] has shown an efficient scale-up compared to other FIM algorithms, it has been worth to come up with a parallel version of FP-Growth [13] (i.e. PFP-Growth). Even though, PFP-Growth is distinguishable with its fast mining process, it has several drawbacks. In particular, with very low $MinSup$, PFP-Growth may run out of memory as illustrated by our experiments in Section 5. Parma algorithm [21], uses an approximation in order to determine the list of frequent itemsets. It has shown better running time and scale-up than PFP-Growth. However, it does not determine an exhaustive list of frequent itemsets, instead, it only approximates them.

A parallel version of Apriori algorithm proposed in [2] requires n MapReduce jobs, in order to determine frequent itemsets of size n. However, the algorithm is not efficient because it requires multiple database scans.

In order to overcome conventional FIM issues and limits, a novel technique, namely CDAR has been proposed in [10]. This algorithm uses a top down approach in order to determine the list of frequent itemsets. It avoids the generation of candidates and renders the mining process more simple by dividing the database into groups of transactions. Although, CDAR algorithm [10] has shown an efficient performance behavior, yet, there has been no proposed parallel version of it.

Another FIM technique, called SON, has been proposed in [9], which consists of dividing the database into n partitions. The mining process starts by searching the local frequent itemsets in each partition independently. Then, the algorithm compares the whole list of local frequent itemsets against the entire database to figure out a final list of global frequent itemsets. In this work, we inspired by SON, and proposed an efficient MapReduce PFIM technique that leverages data placement strategies for optimizing the mining process. Indeed, in order to come up with efficient solutions, we have focused on the data placement as fundamental and essential mining factor in MapReduce.

In [22], the authors proposed an algorithm for partitioning data stream databases in which the data can be appended continuously. In the case of very dynamic databases, instead of PatoH tool which we used in this paper for graph partitioning, we can use the approach proposed in [22] to perform the STDP partitioning efficiently and quickly after arrival of each new data to the database.

7 Conclusion

We have identified the impact of the relationship between data placement and process organization in a massively distributed environment such as MapReduce for frequent itemset mining. This relationship has not been investigated before this work, despite crucial consequences on the extraction time responses allowing the discovery to be done with very low minimum support. Such ability to use very low threshold is mandatory when dealing with Big Data and particularly hundreds of Gigabytes like we have done in our experiments. Our results show that a careful management of processes, along with adequate data placement, may dramatically improve performances and make the difference between an inoperative and a successful extraction.

This work opens interesting research avenues for PFIM in massively distributed environments. In general, we would like to deeply investigate a larger number of algorithms and the impact of data placement on them. More specifically, there are two main factors we want to study. Firstly, we need to better identify what algorithms can be implemented in MapReduce while avoiding to execute a large number of jobs (because the larger the number of jobs, the worse the response time). Secondly, we want to explore data placement alternatives to the ones proposed in this paper.

References

1. Hsinchun Chen, Roger H. L. Chiang, and Veda C. Storey. Business intelligence and analytics: From big data to big impact. *MIS Q.*, 36(4):1165–1188, December 2012.
2. Rajaraman Anand. *Mining of massive datasets*. Cambridge University Press, New York, N.Y. Cambridge, 2012.
3. Bart Goethals. Memory issues in frequent itemset mining. In Hisham Haddad, Andrea Omicini, Roger L. Wainwright, and Lorie M. Liebrock, editors, *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, March 14-17, 2004*, pages 530–534. ACM, 2004.
4. Tom White. *Hadoop : the definitive guide*. O'Reilly, Beijing, 2012.
5. Christian Bizer, Peter A. Boncz, Michael L. Brodie, and Orri Erling. The meaningful use of big data: four perspectives - four challenges. *SIGMOD Record*, 40(4):56–60, 2011.
6. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
7. Hadoop. http://hadoop.apache.org, 2014.
8. Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *Proceedings of International Conference on Very Large Data Bases (VLDB), Santiago de Chile, Chile*, pages 487–499, 1994.
9. Ashok Savasere, Edward Omiecinski, and Shamkant B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, pages 432–444, 1995.
10. Yuh-Jiuan Tsay and Ya-Wen Chang-Chien. An efficient cluster and decomposition algorithm for mining association rules. *Inf. Sci*, 160(1-4):161–171, 2004.
11. Shimon Even. *Graph algorithms*. Computer Science Press, Potomac, Md, 1979.
12. Patoh. http://bmi.osu.edu/ umit/PaToH/manual.pdf, 2011.

13. Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. Pfp: parallel fp-growth for query recommendation. In Pearl Pu, Derek G. Bridge, Bamshad Mobasher, and Francesco Ricci, editors, *Proceedings of the ACM Conference on Recommender Systems (RecSys) Lausanne, Switzerland*, pages 107–114. ACM, 2008.
14. Sean Owen. *Mahout in action*. Manning Publications Co, Shelter Island, N.Y, 2012.
15. Grid5000. https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home.
16. English wikipedia articles. http://dumps.wikimedia.org/enwiki/latest.
17. The clueweb09 dataset. `http://www.lemurproject.org/clueweb09.php/`, 2009.
18. Wei Song, Bingru Yang, and Zhangyan Xu. Index-bittablefi: An improved algorithm for mining frequent itemsets. *Knowl.-Based Syst.*, 21(6):507–513, 2008.
19. N. Jayalakshmi, V. Vidhya, M. Krishnamurthy, and A. Kannan. Frequent itemset generation using double hashing technique. *Procedia Engineering*, 38(0):1467 – 1478, 2012.
20. Han, Pei, and Yin. Mining frequent patterns without candidate generation. *SIGMODREC: ACM SIGMOD Record*, 29, 2000.
21. Matteo Riondato, Justin A. DeBrabant, Rodrigo Fonseca, and Eli Upfal. Parma: a parallel randomized algorithm for approximate association rules mining in mapreduce. In *21st ACM International Conference on Information and Knowledge Management (CIKM), Maui, HI, USA*, pages 85–94. ACM, 2012.
22. Miguel Liroz-Gistau, Reza Akbarinia, Esther Pacitti, Fabio Porto, and Patrick Valduriez. Dynamic Workload-Based Partitioning Algorithms for Continuously Growing Databases. *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, page 105, 2014.