Data placement in massively distributed environments for fast parallel mining of frequent itemsets

Saber Salah, Reza Akbarinia, Florent Masseglia

Data Placement in Massively Distributed Environments for Fast Parallel Mining of Frequent Itemsets

Saber Salah[⋆], Reza Akbarinia, and Florent Masseglia

INRIA & LIRMM
France
FirstName.LastName@inria.fr

Abstract. Frequent itemset mining presents one of the fundamental building blocks in data mining. However, despite the crucial recent advances that have been made in data mining literature, few of both standard and improved solutions scale. This is particularly the case when i) the quantity of data tends to be very large and/or ii) the minimum support is very low. In this paper, we address the problem of parallel frequent itemset mining (PFIM) in very large databases, and study the impact and effectiveness of using specific data placement strategies in a massively distributed environment. By offering a clever data placement and an optimal organization of the extraction algorithms, we show that the arrangement of both the data and the different processes can make the global job either completely inoperative or very effective. In this setting, we propose two different highly scalable, PFIM algorithms, namely P2S (Parallel-2-Steps) and PATD (Parallel Absolute Top Down). P2S algorithm allows discovering itemsets from large databases in two simple, yet efficient parallel jobs, while PATD renders the mining process of very large databases more simple and compact. Its mining process is made up of only one parallel job, which dramatically reduces the mining runtime, the communication cost and the energy power consumption overhead in a distributed computational platform. Our different proposed approaches have been extensively evaluated on massive real-world data sets. The experimental results confirm the effectiveness and scalability of our proposals by the important scale-up obtained with very low minimum supports compared to other alternatives.

1 Introduction

Frequent itemset mining (FIM for short) is one of the fundamental building bricks in data mining. Itemsets might be, for instance, the words occurring in a document, items bought by a customer in a supermarket etc. FIM presents an essential and fundamental role in many domains. In business and e-commerce, for instance, FIM techniques can be applied to recommend new items, such

[⋆] This work has been partially supported by the Inria Project Lab Hemera.

as books and different other products. In science and engineering, FIM can be used to analyze different scientific parameters (e.g., based on their regularities). Finally, FIM methods can aid to perform other data mining tasks such as text mining [1], for instance, and as it will be better illustrated by our experiments in Section 4, FIM techniques can be used to figure out frequent co-occurrences of words in a very large-scale text database.

In the literature, there are several FIM algorithms for mining databases. However, the recent exponential growth of data prevents most of the standard and improved FIM algorithm to scale and achieve reasonable performance when mining Big Data [2]. The manipulation and processing of large-scale databases have opened up new challenges in data mining [3]. The data is no longer located in one computer, instead, it is distributed over several machines. Thus, an efficient parallel design of FIM algorithms must be taken into account. Whatever the size of the database and the minimum support ($MinSup$) threshold, parallel frequent itemset mining (PFIM for short) algorithms should scale and achieve reasonable processing time.

Fortunately, with the availability of powerful programming models, such as MapReduce [4] or Spark [5], the parallelism of most FIM algorithms can be elegantly achieved. MapReduce has gained increasing popularity as shown by the tremendous success of Hadoop [6], an open-source implementation. It is one of the most popular solutions for big data processing [7], in particular due to its automatic management of parallel execution in clusters of machines. Initially proposed in [4], MapReduce divides the computation in two phases, namely map and reduce, which in turn are carried out by several tasks that process the data in parallel. 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 key-value 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.

Despite the robust parallelism setting that the parallel frameworks such as MapReduce and Spark offer, PFIM algorithms remain holding major crucial challenges. With very low $MinSup$, and very large data, as will be illustrated by our experiments, most of the standard and improved PFIM algorithms do not scale. Therefore, the problem of mining large-scale databases does not only depend on the parallelism design of FIM algorithms. In fact, PFIM algorithms have brought the same regular issues and challenges of their sequential implementations. For instance, given a FIM algorithm X and its parallel version X', consider a very low $MinSup$ δ and a database $\mathcal{D}$, if X runs out of memory in a local mode, then, with a large database $\mathcal{D}'$, X' might also exceed available memory in a distributed mode. Thus, the parallelism, all alone, does not guarantee a successful and exhaustive mining of large-scale databases. Therefore, to improve

PFIM algorithms in MapReduce, other issues should be taken into account. Our claim is that the data placement is one of these issues [8].

By thoroughly studying the deep relationships between the distribution principles of massively parallel environments and data mining algorithms characteristics, we investigate an efficient combination between a mining process (i.e., a PFIM algorithm) and an efficient data placement. We show the impact of such placement on the global mining process in these environments. Interestingly and to the best of our knowledge, there has been no focus on studying data placement strategies for improving PFIM algorithms in massively distributed environments. However, as we highlight in this work, the data placement strategies have significant impacts on PFIM performance. We have evaluated the performance of our different proposed solutions through extensive experiments over ClueWeb (up to one Terabyte of size and half a billion of Web pages) and Wikipedia data sets (the whole set of Wikipedia articles in English). Our results show that a careful management of the parallel processes along with an adequate data placement can dramatically improve the performance and make a big difference between an inoperative and a successful extraction. In particular, our results stress on the significant scale-up obtained with very low minimum support for our algorithms compared to other alternatives.

The rest of the paper is structured as follows. Section 2 presents a formal definition of the FIM problem, basic notations, and the necessary background. Section 3 elucidates our data placement techniques and details the working process of our parallel FIM solutions. Section 4 reports on our experiments over real-world data sets. Section 5 discusses related work, and Section 6 concludes.

2 Definitions and Background

In this section, first, we describe the basic notations and terminology that we are going to adopt in the rest of the paper, and give a formal definition of FIM problem. Then, we give the necessary background on MapReduce, and introduce some specific PFIM algorithms that will serve us in the rest of the paper.

2.1 Problem Statement

The problem of frequent itemset mining was first introduced in [9] and then various algorithms have been proposed to solve it. In definition 1, we adopt the notations used in [9].

Definition 1. *Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a set of literals called items. An itemset X is a set of items from $\mathcal{I}$ i.e., $X \subseteq \mathcal{I}$. The size of the itemset X is the number of items in it. A transaction T is a set of elements (i.e., items) such that $T \subseteq \mathcal{I}$ and $T \neq \emptyset$. A transaction T supports the item $x \in \mathcal{I}$ if $x \in T$. A transaction T supports the itemset $X \subseteq \mathcal{I}$ if it supports any item $x \in X$ i.e., $X \subseteq T$. A database $\mathcal{D}$ is a set of transactions. The support of the itemset X in $\mathcal{D}$ is the number of transactions $T \in \mathcal{D}$ that contain X. An itemset $X \subseteq \mathcal{I}$ is $frequent$ in $\mathcal{D}$ if its support is equal or higher than a $MinSup$ threshold.*

The FIM problem consists of extracting all $frequent\ itemset$ from a *database* $\mathcal{D}$ with a minimum support $MinSup$ given by the user.

Example 1. Let us consider a database $\mathcal{D}$ with 5 transactions as shown in Table 1. The items in each presented transaction are delimited by commas. With a minimum support of 3, there will be two frequent itemsets of size one: $\{\{a\}, \{c\}\}$. With a minimum support of 2, there will be 8 frequent itemsets:$\{\{a\}, \{b\}, \{c\}$, $\{f\}, \{g\}$, $\{a, c\}, \{b, c\}, \{f, g\}\}$.

TID	Transaction
T_1	a, b, c
T_2	a, c, d
T_3	b, c
T_4	e, f, g
T_5	a, f, g

Table 1: Database $\mathcal{D}$

In this work, we focus on the parallel frequent itemset mining problem, where the data is distributed over several computational machines. We have adopted MapReduce as a programming model to illustrate our mining approaches.

2.2 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 64 MB 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.

2.3 Parallel Frequent Itemset Mining

One of the primordial FIM algorithms is Apriori [9]. Apriori starts mining the *database* $\mathcal{D}$ by determining a list of $frequent\ items$ of *size* one, say L_1. Then, builds a list of potential $frequent\ itemsets$ of *size* two, say C_2 by joining the *items* in L_1. The algorithm tests the *support* of each C_2 element in $\mathcal{D}$ and returns a list of $frequent\ itemsets$, say L_2. The mining process is carried out until there is no more $frequent\ itemset$ in $\mathcal{D}$. The main drawback of Apriori is the *size* of intermediate *itemsets* that need to be generated. With *itemsets* having a maximum length of n, Apriori needs to generate n times the candidates (i.e., *itemsets*), each being a superset of the previous $frequent\ itemsets$. Usually, the number of intermediate *itemsets* 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 and studying the effect of data placement strategies in MapReduce, we need to briefly describe the SON [10] algorithm. SON simplifies the mining process by dividing the FIM problem into two steps. This makes it very suitable for being used in MapReduce. The steps of this algorithm are as follow.

> **Step 1:** Divide the input *database* $\mathcal{D}$ into $|\mathcal{P}| = n$ chunks (i.e., data partitions), $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$. Mine each chunk in the memory based on a local minimum support $localMinSup$ and a given FIM algorithm. Thus, the first step is devoted to determine a list of local $frequent\ itemsets$ (LFI).
>
> **Step 2:** Filter the LFI list by comparing its elements (i.e., local frequent itemsets) against the entire *database* $\mathcal{D}$ using a global minimum support $globalMinSup$. Return 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 chunk locally. For the needs of this work, we have implemented and tested two different algorithms for this step. The first one is Apriori as described above and the second one is CDAR [11]. The steps of CDAR are given as follow.

Step 1: Divide the *database* $\mathcal{D}$ into $|\mathcal{P}| = n$ chunks, $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, such that each data partition p_i in $\mathcal{P}$ holds only the *transactions* (i.e., *itemsets*) that have length of i.

Step 2: Start mining the data partitions according to *transaction* lengths in descending order. A *transaction* in each data partition represents an *itemset*. If a *transaction* T is $frequent$ in a data partition p_{i+1}, then it will be stored in a list of $frequent$ $itemsets$ L, otherwise, T will be stored in a temporary data structure $Temp$. After checking the occurrence frequency of all T in p_{i+1}, generate i subsets of all T in $Temp$ and adds them to the data partition p_i. The same mining process is carried out until visiting all data partitions $p_i \subset \mathcal{D}$. Before counting the *support* of a *transaction* T, its inclusion in L is checked. If it is included, T is ignored, as it is already in L which is considered as $frequent$.

3 Data Placement

As we briefly mentioned in Section 1, using an efficient data placement technique could improve the performance of PFIM algorithms in MapReduce. This is particularly the case, when the logic and the principle of a parallel mining process is highly sensitive to its data. For instance, let us consider the case when most of the workload of a PFIM algorithm is being done on the mappers. In this case, the way the data is exposed to the mappers could highly contribute to the efficiency and the performance of the whole mining process (i.e., PFIM algorithm). In this setting, we point out to the data placement as a custom placement of *database transactions* in MapReduce. To this end, we use different *data partitioning* methods. In this section, we introduce two PFIM architectures designed for massively distributed environments that exploit our new data placement principles. The first one, called Parallel Two Steps (P2S), performs the extraction in two jobs, with a careful data partitioning, where no overlapping is allowed from one partition to the other. For the second one, called Parallel Absolute Top-Down (PATD), we study even further the data placement techniques and show how part of the computation may be replaced by data duplication. The latter allows overlapping between partitions, hence the organization of this section in two parts, for the presentation of our strategies: non-overlapping and overlapping.

3.1 Non-Overlapping Strategy

The principle of P2S is as follows. Divide the input *database* $\mathcal{D}$ into $|P| = n$ data partitions $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, where $\cup p_i = \mathcal{D}$. Then, perform the mining of frequent itemsets using two jobs:

Job 1: Each mapper takes a data split (partition) and performs a particular FIM algorithm. Then, the mapper emits a list of local frequent itemsets to the reducer.

Job 2: Each mapper takes a *database* $\mathcal{D}$ as input, and filters the global frequent itemsets from the list of local frequent itemsets. Then, writes the final results to the reducer.

Thus, P2S divides the mining process into two steps. As one may observe from its pseudo-code, given by Algorithm 1, P2S is very well suited for MapReduce. From the mining point of view, P2S is inspired from the SON algorithm [10]. The main reason behind opting SON as a reference for 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 4, a straightforward implementation of SON in MapReduce would not be the best solution for the FIM problem. Therefore, with P2S we propose new solutions for PFIM problem within the "two steps" architecture.

Algorithm 1: P2S

```
Input: Database $\mathcal{D}$ and $MinSup\ \delta$
Output: Frequent Itemsets
//Map Task 1
map( key:Null : $\mathcal{K}_1$, value = Whole Data Split: $\mathcal{V}_1$ )
    - Determine a local $MinSup\ ls$ from $\mathcal{V}_1$ based on $\delta$
    - Perform a complete FIM algorithm on $\mathcal{V}_1$ using $ls$
    emit (key: local frequent itemset, value: Null)
//Reduce Task 1
reduce( key:local frequent itemset, list(values) )
    emit (key,Null)
//Map Task 2
Read the list of local frequent itemsets from Hadoop Distributed Cache $LFI$ once
map( key:line offset : $\mathcal{K}_1$, value = Database Line: $\mathcal{V}_1$ )
    if an itemset $i \in LFI$ and $i \subseteq \mathcal{V}_1$ then
        key $\leftarrow$ i
    emit (key:i, value: 1)
//Reduce Task 2
reduce( key:i, list(values) )
    $sum \leftarrow 0$
    while $values.hasNext()$ do
        $sum += values.next().get()$
    if $sum >= \delta$ then
        emit (key:i, value: Null)
```

Actually, the first MapReduce job of P2S consists of applying specific FIM algorithm at each mapper based on a local minimum support (*localMinSup*). A local minimum support is computed at each mapper based on a global $MinSup$ δ percentage and the number of transactions of the data split being processed. For instance, suppose that we are given a database $\mathcal{D}$ with 10 transactions. Let us consider a global minimum support of 4 (i.e., an itemset is frequent in $\mathcal{D}$ if it appears 4 times or more). Suppose we divide $\mathcal{D}$ into two data partitions P_1 and P_2 where each one holds 5 transactions from $\mathcal{D}$. Then, the local minimum support in P_1 and P_2 would be equal to 2 (40% of 5). 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 list of 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 m reads once the list of local frequent itemset stored in Hadoop Distributed Cache. Then, m takes one *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, each mapper emits all local frequent itemsets with their complete occurrences in the whole *database* (i.e., key: *itemset*, value: 1). The reducer simply computes the sum of the count values of each key (i.e., local frequent itemset) by iterating over the list of values (i.e., ones) of each key. Finally, the reducer compares the number of occurrences of each local frequent itemset to $MinSup$ δ (i.e., global minimum support), if it is greater or equal to δ then the local frequent itemset is considered as a global frequent itemset, otherwise, the reducer discards the key (i.e., local frequent itemset).

The two steps approach (i.e., P2S) would not miss any itemset as false negative case i.e., at the first MapReduce job since each mapper holds a specific number k of the database transactions. k is involved when computing the local minimum support based on the global one (e.g., the local minimum support is proportional to k). Thus, regardless the skewness of the data being distributed among the mappers, we always guarantee the local frequency of the itemsets. Hence, a local frequent itemset cannot be missed at the first job of P2S. Then, its global frequency is determined at the second job of P2S.

The performance of PFIM algorithms in MapReduce may strongly depend on the distribution of the data among the workers. To illustrate this, let us consider an example of a PFIM algorithm that 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 data partition that holds long $frequent\ itemsets$ would take more execution time. In the worst case, the job given to that specific mapper would not complete and making the global extraction process impossible. Thus, despite the fairly automatic data distribution by Hadoop, the computation would highly depend on the design logic of PFIM algorithms in MapReduce. 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. 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 P2S first Job. If the split of a mapper contains a subset of $\mathcal{D}$ that would lead to lengthy local frequent itemsets, then, it would be the bottleneck of the whole mining process and might even not be able to complete. 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 well suited for long *itemsets*. Actually, the working principle of CDAR is to first extract the longest patterns (i.e., *itemsets*) 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 mining 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 improve the efficiency of some data sensitive PFIM algorithms, P2S uses different data placement strategies such as *Random Transaction Data Placement* and *Similar Transaction Data Placement.*

3.1.1 Random Transaction Data Placement (RTDP): this technique merely refers to a random process for choosing bunch of *transactions* from a *database* $\mathcal{D}$. Thus, using RTDP strategy, the *database* is divided into $|P| = n$ data partitions $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, where $\cup p_i = \mathcal{D}$. RTDP does not rely on any constraint for placing such bunch of *transactions* in same data partition.

3.1.2 Similar Transaction Data Placement (STDP): 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 data partition p. Therefore, the *database* $\mathcal{D}$ is split into n disjoint data partitions. In STDP, each data split would be more homogeneous, unlike the case of using RTDP. More precisely, by creating data partitions that contain similar transactions, we increase the chance that each data split will contain local frequent itemsets of high length.

Partitioning the data according to similarities is a complex problem. A clustering algorithm may seem appropriate for this task, but it might be time consuming. We propose a graph data partitioning mechanism that will allow for a fast execution of this step, thanks to existing efficient algorithms for graph partitioning such as Min-Cut [12]. In the following, we describe how transaction data can be transformed into graph data for doing such partitioning.

Step 1: For each unique *item* in $\mathcal{D}$, we determine the list of *transactions* L that contain it. Let $\mathcal{D}'$ be the set of all *transaction* lists L.

Step 2: We present $\mathcal{D}'$ as a graph $G = (V, E)$, where V denotes a set of vertices and E is a set of edges. Each *transaction* $T \in \mathcal{D}$ refers to a vertex

$v_i \in G$ where $i = 1, \ldots, n$. 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.

Step 3: After building the graph G, a Min-Cut algorithm is applied in order to partition $\mathcal{D}'$.

In the above approach, the similarity of two *transactions* is evaluated by 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 $\mathcal{D}$, the *database* from Table 1. We start by mapping each *item* in $\mathcal{D}$ to its *transactions* holder. As illustrated in the table of figure 3.1.2, T_1 and T_2 have 2 common *items*. 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 $\mathcal{D}'$ refers to the weight of their edges. In order to partition $\mathcal{D}'$, we first build a graph G from $\mathcal{D}'$ as shown in Figure 3.1.2. Then, the algorithm Min-Cut finds a minimum cut in G (red line in Figure 4) 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

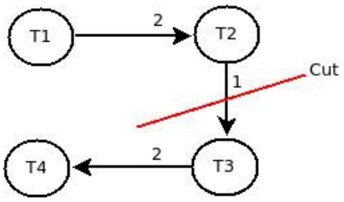


Fig. 1: Transactions of a database (left) & Graph representation of the database (right)

We have used a particular graph partitioning tool namely PaToH [13] to generate data partitions. The reason behind opting Patoh lies in its set of configurable properties e.g., the number of data partitions and the partition load balance factor.

Based on the architecture of P2S and the data placement strategies we have developed and efficiently designed two PFIM mining algorithms. Parallel Two Steps CDAR (P2SC) and Parallel Two Steps Apriori (P2SA). P2SC and P2SA respectively represents instances of P2S algorithm with CDAR [11] and Apriori (i.e., implemented at P2S first job).

In a massively distributed environment, the main bottleneck of P2S algorithm would be its first execution phase (i.e., mapper execution of P2S first job to determine local frequent itemsets). Intuitively, in the case of P2SC, the mapper that holds more homogeneous data (i.e., similar *transactions*) would be faster. By referring to CDAR [11] mining principle, conceivably, a mapper that holds homogeneous *transactions* allows for more *itemset* inclusions, which results in less subset generation. Thus, placing each bucket of similar *transactions* on the mappers would improve the performance of P2SC algorithm. This data placement technique (i.e., STDP) can be achieved by means of different data partitioning methods. In this setting, and because it is designed for an optimal relationship between data placement and the adequate FIM algorithm, we denote by Opt-P2S the algorithm P2SC that is based on STDP data placement.

In contrast, logically, STDP would not improve the performance of P2SA, instead it should lower it. Intuitively, each mapper would hold a data partition (i.e., data split) of similar *transactions* that allows for a high number of $frequent$ *itemsets*. This results in a huge number of *itemset* candidate generations. Using RTDP to randomly place *transactions* on the mappers, should give the best performance of P2SA. Our experiments given in Section 4 clearly illustrate this intuition.

PFIM algorithms that depend on P2S mining design perform two MapReduce jobs to determine an exhaustive list of all frequent itemsets. Unfortunately, this may drop down the whole mining process by duplicating the mining results. P2S based algorithms may output *itemsets* that are locally frequent with no guarantee about their global frequency. Hence, in fact they amplify the number of transferred data (i.e., *itemsets*) between mappers and reducers.

To cover the different above-mentioned issues, our major challenge is to limit the mining process to one simple MapReduce job. This should guarantee a very fast *itemset* mining process, low data communications and less energy power consumption in a distributed computing environment. To this end, we take the full advantage of the available massive data storage space, CPU(s) etc. We efficiently designed and developed our Parallel Absolute Top Down (PATD) algorithm to tackle all above mentioned issues.

3.2 Overlapping: a Novel Strategy for Parallel FIM

As illustrated above, paying particular care to data placement is crucial for efficient PFIM algorithms in massively distributed environments. Building on this first result, we explore this idea further and propose novel solutions for transferring one part of the computation from the FIM algorithm itself, to the data placement. This is feasible due to two factors. First, the disk space available in these environments is not a limit, and data might be duplicated without causing any trouble. Second, the frequency of an itemset X is the same on two different partitions P_1 and P_2, if and only if it is guaranteed that each transaction that contains X occurs in both P_1 and P_2.

In the following, we introduce our Item Based Data Partitioning Strategy (IBDP for short). We depict and illustrate its process through an illustrative example. Then, we introduce our PATD algorithm and we detail its core mining process and principle for a fast and successful frequent *itemset* extraction from very large *databases*. Finally, we validate our proposed PATD algorithm and provide a proof of correctness.

The main idea is to prepare the data with a partitioning scheme that will allow better performances in the itemset extraction. The partitioning scheme is denoted IBDP in the following. IBDP takes 2 jobs to complete, as explained in Section 3.2.1. The itemset mining algorithm applied on the partitions of IBDP is denoted PATD. PATD takes one additional job, after IBDP. In total, the complete pattern mining process of IBDP+PATD is thus done in 3 jobs. However, as long as the mininum support given to PATD is at least equal to the minimum support given to IBDP, PATD may be applied many times without calling IBDP. Our goal is to offer a data mining algorithm that corresponds well to real cases of data analytics. Actually, it is not unusual to apply a pattern mining algorithm with a given minimum support and then apply it again with another support (for instance a higher support if the current mining process is too slow, or it gives too much patterns, or a lower support if no pattern was extracted with the current support, etc.). Pattern mining generally calls for empiric parameter settings, and the consequence is to apply the algorithm with different values of minimum support until the results are satisfying from the end-user's point of view. In order to provide a suitable workflow, we consider that IBDP should be applied with a very low support only once. Then, each call to PATD will need only one job.

3.2.1 Itemset Based Data Placement. Our claim is that duplicating the data on the mappers allows for a better accuracy in the first P2S job and therefore leads to less infrequent *itemsets* (meaning, less communications and fast processing). Consider a data placement with a high overlap, for instance, with 10 data partitions, each holding 50% of the *database*. Obviously, there will be less globally infrequent *itemsets* in the first job (in other words, if an *itemset* is frequent on a mapper, then it is highly likely to be frequent on the whole *database*). Unfortunately, such an approach is not realistic. First, we still need a second job to filter the local frequent itemsets and check their global frequency. Furthermore, such a thoughtless placement is absolutely not plausible, given the massive data sets we are dealing with.

Thus, we take advantage of this duplication opportunity and propose IBDP, an efficient strategy for partitioning the data over all mappers, with an optimal amount of duplicated data, allowing for an exhaustive mining in just one MapReduce job. The goal of IBDP is to replace part of the mining process by a clever placement strategy and optimal data duplication. The main idea of IBDP is to consider the different groups of frequent *itemsets* that are usually extracted. Let us consider a minimum support $\varDelta$ and X a frequent *itemset* according to $\varDelta$ on $\mathcal{D}$. Let S_X be the subset of $\mathcal{D}$ restricted to the *transactions* supporting X. The first expectation is to have $|S_X| \ll |\mathcal{D}|$ since we are working with very low minimum support thresholds. The second expectation is that X can be extracted from S_X with $\varDelta$ as a minimum support. The goal of IBDP is a follows: for each frequent itemset X, build S_X the subset from which the extraction of X can be done in one job. Fortunately, itemsets usually share a lot of items between each other. For instance, with Wikipedia articles, there will be a group of itemsets related to the *Olympic games*, another group of itemsets related to *Algorithms*, etc. IBDP exploits these affinities between itemsets. It divides the search space by building subsets of $\mathcal{D}$ that correspond to these groups of itemsets, optimizing the size of duplicated data.

More precisely, given a database of transactions $\mathcal{D}$, and its representation in the form of a set $\mathcal{S}$ of n non-overlapping data partitions $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$. Each one of these non-overlapping data partitions (i.e, $S_i \cap S_j = \emptyset$ for $\forall i, j \in \{1, \ldots, n\} \wedge i \neq j$), holds a set of similar transactions (the union of all elements in $\mathcal{S}$ is $\mathcal{D}$, $\bigcup_{i=1}^{n} S_i = \mathcal{D}$). For each non-overlapping data partition S_i in $\mathcal{S}$, we extract a "centroid". The centroid of S_i contains the different items, and their number of occurrences, in S_i. Only the items having a maximum number of occurrences over the whole set of partitions, are kept for each centroid. Once the centroids are built, IBDP simply intersects each centroid of S_i with each transaction in $\mathcal{D}$. If a transaction in $\mathcal{D}$ shares an item with a centroid of S_i, then the intersection of this transaction and the centroid will be placed in an overlapping data partition called S_i'. If we have n non-overlapping data partitions (i.e, n centroids), IBDP generates n overlapping data partitions and distributes them on the mappers. The core working process of IBDP data partitioning in MapReduce is given in Algorithm 2.

Job 1: *Centroids*: each mapper takes a *transaction* (line of text) from non-overlapping data partitions as a value, $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, and the name of the split being processed as key. Then, it tokenizes each *transaction* (i.e., value) to determine different *items*, and emits each *item* as a key coupled with its split name as a value. After mappers termination, the reducer aggregates over the keys (i.e., *items*) and emits each key coupled with its different value (i.e., split name) in the list of values (i.e., split names).

Job 2: *Overlapping Data Partitions*: the format of the MapReduce output is set to "MultiFileOutput" in the driver class. In this case, each key will denote the name of each overlapping data partition (override the "generateFileNameForKeyValue" function in MapReduce to return a string as a key). In the map function, first, we store once the previous MapReduce job centroids in a key-value data structure (e.g., MultiHashMap etc.). The key in the used data structure is the split name and the value is a list of *items*. Each mapper takes a *transaction* (line of text) from the *database* $\mathcal{D}$, and for each key in the used data structure, if there is an intersection between the values (i.e., list of *items*) and the *transaction* being processed, then the mapper emits the key as the split name (in the used data structure) and value as the *transaction* of $\mathcal{D}$. The reducer simply aggregates over the keys (split names) and writes each *transaction* of $\mathcal{D}$ to an overlapping data partition file.

3.2.2 Parallel Absolute Top Down: Complete Approach. We take the full advantage from IBDP data partitioning strategy and propose a powerful and robust PFIM algorithm namely PATD. PATD algorithm limits the mining process of very large *databases* to one simple MapReduce job and exploits the

Algorithm 2: IBDP

```
//Job1
Input: Non-overlapping data partitions $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ of a database $\mathcal{D}$
Output: Centroids
//Map Task 1
map( key: Split Name: $\mathcal{K}_1$, value = Transaction (Text Line): $\mathcal{V}_1$ )
    - Tokenize $\mathcal{V}_1$, to separate all items
    emit (key: Item, value: Split Name)

//Reduce Task 1
reduce( key: Item, list(values) )
    while values.hasNext() do
        emit (key:(Split Name) values.next (Item))

//Job2
Input: Database $\mathcal{D}$
Output: Overlapping Data Partitions
//Map Task 2
- Read previous job1 result once in a (key, values) data structure (DS), where
key: SplitName and values: Items
map( key: Null: $\mathcal{K}_1$, value = Transaction (Text Line): $\mathcal{V}_1$ )
    for SplitName in DS do  if Items.Item $\cap \mathcal{V}_1 \neq \emptyset$ then
        emit (key: SplitName, value: $\mathcal{V}_1$)


//Reduce Task 2
reduce( key: SplitName, list(values) )
    while values.hasNext() do
        emit (key: (SplitName), values.next: (Transaction))
```

natural design of MapReduce framework. Given a set of overlapping data partitions ($\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$) of a *database* $\mathcal{D}$ and an absolute minimum support *AMinSup* $\varDelta$. PATD algorithm mines each overlapping data partition S_i independently. At each mapper m_i, $i = 1, ..., n$, PATD performs CDAR algorithm on S_i. The mining process is based on the same *AMinSup* $\varDelta$ for all mappers i.e., each overlapping data partition S_i is mined based on $\varDelta$. The mining process is carried out in parallel on all mappers. The mining result (i.e., $frequent$ $itemsets$) of each mapper m_i is sent to the reducer. The latter receives each $frequent$ $itemsets$ as its key and null as its value. The reducer aggregates over the keys ($frequent$ $itemsets$) and writes the final result to a distributed file system (i.e., HDFS).

The main activities of the mappers and reducers in PATD algorithm are summarized as follow.

> **Mapper:** Each mapper is given a S_i, $i = 1...m$ overlapping data partition, and a global absolute minimum support (i.e., *AMinSup*). The latter, performs CDAR algorithm on S_i. Then, it emits each $frequent$ $itemset$ as a key and null for its value, to the reducer.
>
> **Reducer:** The reducer simply aggregates over the keys ($frequent$ $itemsets$ received from all mappers) and writes the final result to a distributed file system.

Example 3. Let us take the example of Figure 2. Given an absolute minimum support $\varDelta = 2$ (i.e., an *itemset* is considered frequent, if it appears at least in *two* transactions in $\mathcal{D}$). Following PATD mining principle, each mapper is given an overlapping data partition S_i as a value. In our example, we have two overlapping data partitions. We consider *two* mappers m_1 and m_2, each one performs a complete CDAR with a minimum support $\varDelta = 2$. In Figure 2 from bottom-up : mapper m_1 mines first overlapping data partition and returns {fg} as a $frequent$ $itemset$. Likewise, mapper m_2 mines second overlapping data partition and returns {{ac}, {bc}}. All the results are sent to the reducer, the reducer aggregates over the keys ($frequent$ $itemsets$) and outputs the final result to a distributed file system.

3.2.3 Proof of Correctness.

To prove the correctness of PATD algorithm, it is sufficient to prove that if an *itemset* x is $frequent$, then it is $frequent$ in at least one of the data partitions produced by IBDP. Since each data partition is locally mined by one mapper, then x will be found as $frequent$ by one of the mappers. Thus, the correctness proof is done by the following lemma.

Lemma 1. *Given a database $\mathcal{D} = \{T_1, T_2, \ldots, T_n\}$, and an absolute minimum support $\varDelta$, then $\forall$ itemset $x \subseteq \mathcal{D}$ we have:*

$Support_{\mathcal{D}}(x) \geq \varDelta \Leftrightarrow \exists\, \mathcal{P} \mid Support_{\mathcal{P}}(x) \geq \varDelta$, where $\mathcal{P}$ denotes one of the data partitions obtained by performing IBDP on $\mathcal{D}$.

Proof.

We first prove that if $Support_{\mathcal{D}}(x) \geq \Delta$ then $\exists\, \mathcal{P} \mid Support_{\mathcal{P}}(x) \geq \Delta$.

$\forall\, T_i \in \{T_1, T_2, \ldots, T_n\}$, the intersection of T_i with $\mathcal{I}$ (the set of unique *items* of $\mathcal{D}$) is T_i and the union of these all intersections is $\mathcal{D}$. Thus, in this particular case, $Support_{\mathcal{D}}(x) \geq \Delta \Rightarrow \exists\, \mathcal{D} \mid Support_{\mathcal{D}}(x) \geq \Delta$. If the set of unique *items* $\mathcal{I}$ is partitioned into k items partitions, then the intersection of each one of these k items partitions with all transactions $\{T_1, T_2, \ldots, T_n\}$ in $\mathcal{D}$, would result in a new data partition $\mathcal{P}$. Let us denote by $\Pi = \{P_1, P_2, \ldots, P_k\}$, the set of all these new data partitions.

Suppose a given *itemset* x in $\mathcal{D}$. Since $x \subseteq \mathcal{I}$ (i.e., x is included in the set of items), then there is at least one items partition, say X_j, such that the intersection of x and X_j is not empty. Let i be an item included in the intersection of x and Xj, i.e., $i \in x \cap X_j$. Let P_j be the partition associated to X_j, then P_j contains all transactions containing i. Thus, P_j contains all transactions containing x. Therefore, if x is globally frequent, then it is also frequent in P_j, i.e. $Support_{\mathcal{D}}(x) \geq \Delta \Rightarrow \exists\, I_P \mid Support_{I_P}(x) \geq \Delta$

Next, we prove the inverse i.e., if $\exists\, \mathcal{P} \mid Support_{\mathcal{P}}(x) \geq \Delta$ then $Support_{\mathcal{D}}(x) \geq \Delta$.
This is done simply by using the fact that each data partition $\mathcal{P}$ is a subset of $\mathcal{D}$. Hence, if the support of x in $\mathcal{P}$ is higher than Δ, then this will be the case in $\mathcal{D}$. Thus, we have: if $\exists\, \mathcal{P} \mid Support_{\mathcal{P}}(x) \geq \Delta \Rightarrow Support_{\mathcal{D}}(x) \geq \Delta$.

Therefore, we conclude that: $Support_{\mathcal{D}}(x) \geq \Delta \Leftrightarrow \exists\, \mathcal{P} \mid Support_{\mathcal{P}}(x) \geq \Delta$.

An illustration of the complete process (IBDP+PATD) is given in example 4.

Example 4. Figure 2 illustrates the principle of PATD on the *transaction database* $\mathcal{D}$ of Table 1. In this example, we have two non-overlapping data partitions, built according to transaction similarities, at step (1) and thus two centroids at step (2). The centroids are filtered in order to keep only the *items* having the maximum number of occurrences (3). IBDP intercepts each one of these two centroids with all *transactions* in $\mathcal{D}$. A transaction T is copied to the partition of a centroid c if $T \cap c \neq \emptyset$. Then, on each partition, the locally infrequent items are removed (4). This results in two overlapping data partitions in (5). Finally, the $frequent\ itemsets$ are extracted (6) locally (note that only the maximal frequent itemset are shown in figure 2). Redundancy is used for the counting process of different *itemsets*. For instance, transaction afg is duplicated in both data partitions in (4) where the upper version participates to the frequency counting of a and the lower version participates to the frequency counting of fg.

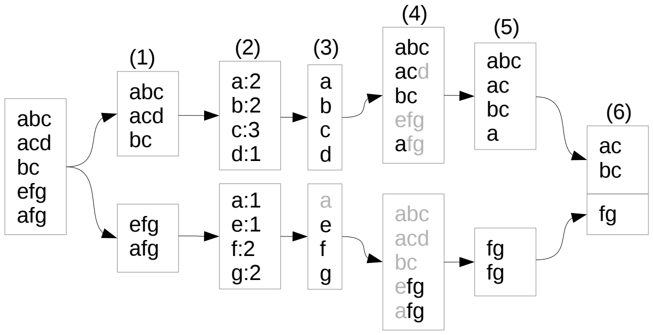


Fig. 2: Data Partitioning Process: (1) partitions of similar transactions are built; (2) centroids are extracted; (3) and filtered; (4) transaction are placed and filtered ; (5) overlapping partitions are ready for pattern extraction; (6) local frequent itemsets are also globally frequent.

4 Experiments

To assess the performance of our proposals, we have carried out extensive experimental evaluations. In Section 4.1, we depict our experimental setup. In Section 4.2, we describe the data sets used for our various experiments. In Section 4.3 and Section 4.4, we discuss the results of our experiments.

4.1 Experimental Setup

We implemented all different presented PFIM algorithms and data placement strategies on top of Hadoop MapReduce using the Java programming language version 1.7 and Hadoop version 1.0.3. For comparison with PFPGrowth [14], we adopted the default implementation provided in the Mahout [15] machine learning library (Version 0.7). Since our algorithm and the algorithms from the literature used for comparisons do not filter on the top-k results, we have set the parameter k (top-k frequent itemsets to be extracted) of PFPGrowth algorithm to 10^6. This allows us to obtain the whole set of frequent itemsets with PFPGrowth, and not only the top-k ones.

We have carried out two bunch of experiments. In the first set of experiments, described by section 4.3, we evaluate the performances of P2S, compared to existing work in the literature. In the second bunch of experiments, presented in section 4.4, we stress our experimental evaluations on our PATD algorithm.

We carried out all our experiments in Grid5000 [16], which is a platform for large-scale data processing. We have used a cluster of 16 machines for both the English Wikipedia [17] and the Amazon Reviews [18] data sets, while we have used 48 machines for the ClueWeb [19] data set. All of these data sets are described in Section 4.2.

When assessing the performance of PATD algorithm using the English Wikipedia data set, we have varied the number of machines from 16 up to 48 to get an overview of the impact of using more nodes on the speedup performance of the different presented PFIM algorithms.

In our different experiments, we have used clusters of homogeneous machines (i.e., having similar properties), where each machine is equipped with a Linux operating system, 64 Gigabytes of main memory, Intel Xeon $X3440$ 4 core CPUs, and 320 Gigabytes SATA II hard disk. For 2-Jobs Schema, we performed our experiments by varying the minimum support $MinSup$ parameter value for each algorithm along with a particular data placement strategy (STDP or RTDP).

To evaluate the performance of our different proposals and compare them to existing solutions in the literature, we have taken into account different measures. We consider the execution time (i.e., elapsed mining time: from the beginning of the mining process until its end) to evaluate the performance of both PATD and P2S algorithms. Furthermore, we consider the total amount of transferred data and the energy power consumption in the Grid5000 [16] platform when assessing the performance of PATD algorithm. i.e., the total amount of transferred data represents the sum of the quantity of data being transmitted between the mappers and the reducers, while the energy power consumption accounts for the quantity of power consumed by the machines when executing a particular PFIM algorithm.

When assessing both the impact of data placement and the PATD algorithm, we have used 500 mappers for all the data sets and number of reducers equals to the number of used machines (i.e., 16 and 48 reducers). This choice of the number of the reducer is taken based on several attempts to get an optimal configuration.

4.2 Data Sets

We have performed our experiments on three real-world data sets. To evaluate the performance of PATD algorithm, we used as a first data set which is the latest dump of the 2014 English Wikipedia articles [17] having a total size of 49 Gigabytes, and composed of 5 millions articles. The second data set is the 2013 Amazon Reviews [18] data set having a size of 34 Gigabytes, and composed of 34 millions reviews. The third data set is a sample of the ClueWeb English data set [19] with size of one Terabyte and having 632 million articles. Likewise, to assess the performance of Opt-P2S and the impact of different used data placement strategies (i.e., STDP and RTDP), we used the same English Wikipedia, Amazon Reviews data sets, and a sample of the ClueWeb English data set [19] with size of 240 Gigabytes and having 228 millions articles.

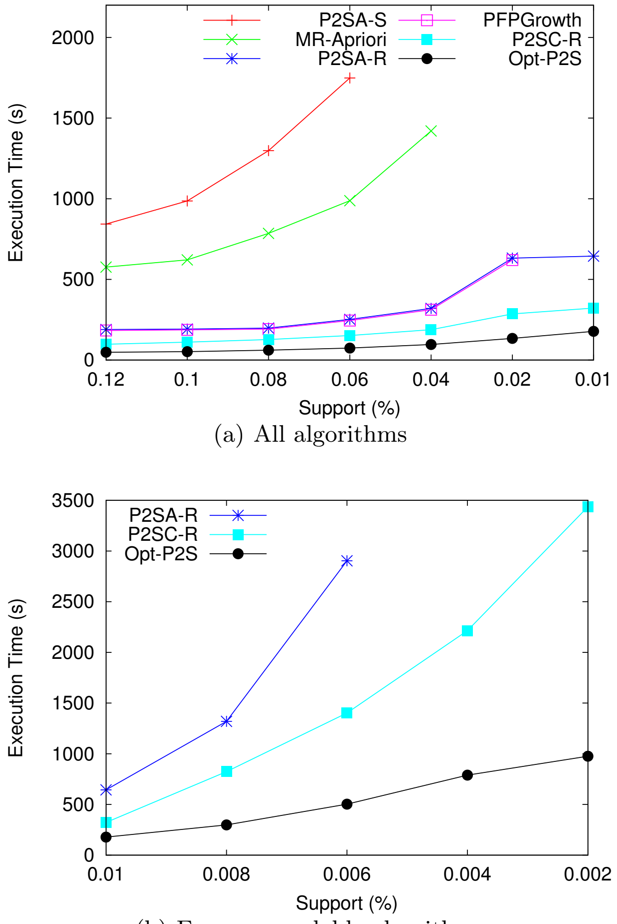
For each data set we performed a data cleaning task by removing all English stop words from all articles and obtained a data set where each article represents a transaction (the items are the corresponding words in the article) to each invoked PFIM algorithm in our experiments.

4.3 2-Jobs Schema

Here, we evaluate the impact of RTDP and STDP data placements on the performance of PFIM algorithms that is based only on P2S design schema. Consider a FIM algorithm called 'x'. We denote by P2Sx-R and P2Sx-S the use of our P2S principle with STDP (P2Sx-S) or RTDP (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 STDP 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). In our experiments there has been no difference between using STDP and RTDP on MR-Apriori performance. Thus, we decided to restrict our comparison to only MR-Apriori with RTDP data placement strategy. We denote by P2SC-R for Parallel Two Steps CDAR with RTDP data placement. Finally, we denote by Opt-P2S the optimized combination between P2S architecture and a FIM algorithm (i.e., P2SC algorithm with STDP data placement strategy).

Figures 3(a) and 3(b) report our results on the whole set of Wikipedia articles in English. Figures 3(a) 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 test machines, so it is impossible to extract frequent patterns with this algorithm. Another interesting observation is that P2SA-S i.e., the two steps algorithm that use Apriori as a local mining solution, shows lower performance than MR-Apriori. This is an important result, since it confirms that a bad choice of data-process relationship compromises a complete analytic process and makes it inoperative. Let us now consider the set of the 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 is that PFPGrowth exceeds the available memory below 0.02%. Eventually, P2SC-R and Opt-P2S give the best performances, with an advantage for Opt-P2S. In this experiment on the English Wikipedia data set, the time for creating the STDP data partitions (500 data partitions) using PaToH is 133 seconds. Since it is done only once and all the runs of Opt-P2S use the same data placement, this time is mentionned here in the text but is not considered in Figures 3(a) and 3(b)

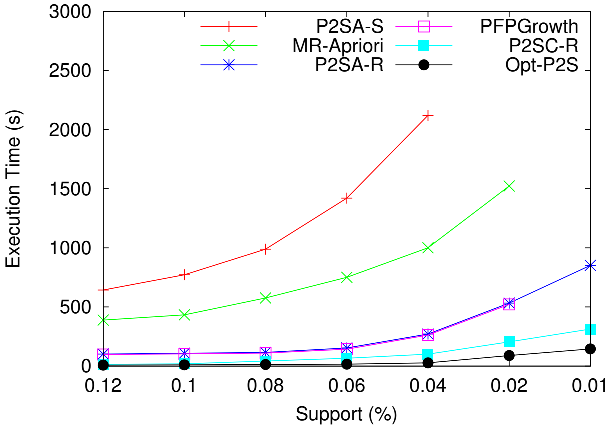
Figure 3(b) focuses on the differences between the three algorithms that scale in Figure 3(a). The first observation is that P2SA-R is not able to provide results below 0.006%. Regarding the algorithms based on the principle of P2S, we can observe a very good performance for Opt-P2S thanks to its optimization between data and process relationship. These results illustrate the advantage of



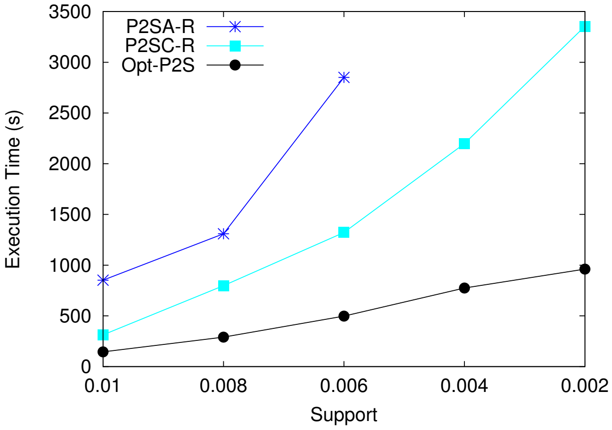
(a) All algorithms

(b) Focus on scalable algorithms

Fig. 3: Runtime and scalability on English Wikipedia data set



(a) All algorithms



(b) Focus on scalable algorithms

Fig. 4: Runtime and scalability on Amazon Reviews data set

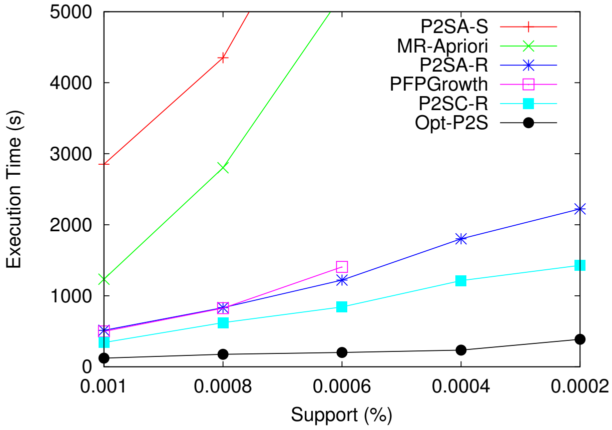


Fig. 5: Runtime on ClueWeb data set

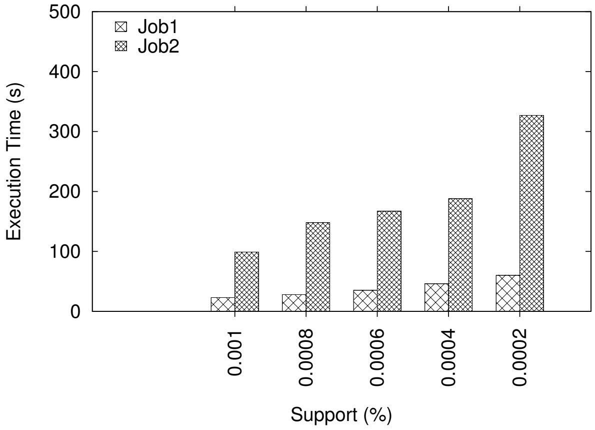


Fig. 6: Opt-P2S: Job1 Vs Job2

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*.

Figures 4(a) and 4(b) illustrate our results on the Amazon Reviews data set. Figures 4(a) shows a global view on algorithms performances for a support varying from 0.12% to 0.01%. As on the English Wikipedia data set, we observe that the performance of MR-Apriori is poor compared to other presented PFIM alternatives. Particularly, below a minimum support of 0.02%, MR-Apriori is not able to extract the frequent itemsets. On the other hand, we see that P2SA-S (the two steps algorithm that uses Apriori as a local mining solution) gives very poor performance which is lower than MR-Apriori performance. In Particular, with a minimum support that is less than 0.04%, P2SA-S becomes unable to extract the frequent itemsets. Again, as on the English Wikipedia data set, this performance result of P2SA-S algorithm obviously confirms the prominent role of using a particular data placement strategy along with a specific FIM algorithm. Similarly, as in Figure 3(a), we see in Figure 4(a) that PFPGrowth and P2SA-R algorithms give better performance compared to MR-Apriori and P2SA-S algorithms. Finally, we record better performances of P2SC-R and Opt-P2S compared to other presented PFIM alternatives. Particularly, our solution Opt-P2S shows very good performance compared to P2SC-R and all other presented PFIM algorithms.

Figure 4(b) highlights the differences between the algorithms that scale in Figure 4(a). We see that P2SA-R algorithm cannot continue scaling below a minimum support of 0.006%. On the other side, we observe that P2SC-R continues scaling. However, with a minimum support of 0.002% P2SC-R is outperformed by Opt-P2S algorithm. This very good performance of Opt-P2S algorithm confirms the fundamental role of using an adequate data placement strategy along with a specific FIM algorithm to successfully optimize a mining process and renders it highly operative.

In this experiment on the Amazon Reviews data set, the overall runtime for building the STDP data partitions (500 data partitions) using PaToH is 110 seconds.

In Figure 5, similar experiments have been conducted on the ClueWeb data set. We observe that the same order between all algorithms is kept, compared to Figures 3(a) and 4(a). There are two bunches of algorithms. One, made of P2SA-S and MR-Apriori which cannot reasonably applied to this data set, whatever the minimum support. In the other bunch, we see that PFPGrowth suffers from the same limitations as could be observed on the Wikipedia data set in Figure 3(a) and on the Amazon Reviews data set in Figure 4(a). PFPGrowth follows a behavior that is very similar to that of P2SA-R, until it becomes impossible to execute.

On the other hand, P2SC-R and Opt-P2S are the two best solutions, while Opt-P2S is the optimal combination of data placement and algorithm choice for local extraction, providing the best relationship between data and process. In this particular experiment with this very large data set size, we recorded 663 seconds to build the STDP data partitions using PaToH.

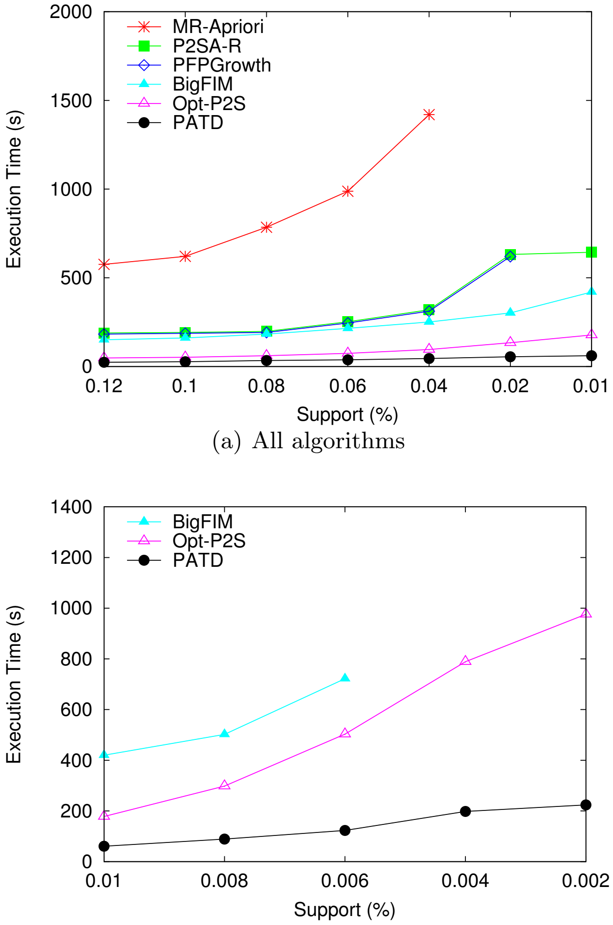
Figure 6 gives an entire overview on the difference between the execution times of the first and the second job of Opt-P2S algorithm on the ClueWeb data set. We observe that by varying the minimum support from 0.001% to 0.0002%, the performance of the first MapReduce job is always faster than the second job. This difference in the runtime performance is more significant when the minimum support threshold (e.g., 0.0002% in Figure 6). This behavior of Opt-P2S first job confirms our intuition on the optimization of the mining process by combining the data with a specific mining process. The bottleneck of the two steps mining approach is due to its second step. This result confirms our analysis on the limitations of the two steps approach that are solved in our 1-Job schema approach described in the following.

4.4 1-Job Schema

In this section, first we focus on the runtime and scalability of PATD algorithm compared to other proposed PFIM algorithms in the literature. Second, we discuss the impact of increasing the number of machines on the speedup performance of each PFIM algorithm. Eventually, we discuss the amount of transferred data and energy power consumption of different compared algorithms. In particular, we consider all different measurements when the minimum support is very low. Beside the PFIM algorithms that we have used to evaluate the impact of using data placement strategies along with a specific FIM algorithm, we have compared the performance of our PATD solution to BigFIM [20] algorithm.

4.4.1 Runtime and Scalability. Figures 7, 8 and 9 show a complete view of our experiments on English Wikipedia, ClueWeb and Amazon Reviews data sets. Figures 7(a) and 7(b) illustrate our experimental results on the whole English Wikipedia data set. Figure 7(a) gives an entire view on algorithms performances for a minimum support ranging from 0.12% to 0.01%. We see that MR-Apriori algorithm presents a very weak performance. With a minimum support below 0.04%, MR-Apriori algorithm becomes unable to scale. In the other side, we see that P2SA-R performance tends to be very closed to PFPGrowth until a minimum support of 0.02%. P2SA-R algorithm continues scaling with 0.01% while PFPGrowth does not. Although P2SA-R algorithm is outperformed by BigFIM, its scalability performance behavior explains the fundamental role of using such adequate data placement strategy (in this case RTDP). This observation, is better illustrated by the performance given by Opt-P2S algorithm. Opt-P2S scales with low minimum support values and outperforms the other algorithms. This stresses the high capability of data placements techniques (i.e., RTDP or STDP) to deviate the performance of the whole mining process. Although, these techniques (i.e., data placements) have resulted in high performance improvements of P2S based algorithms, they are outperformed by PATD algorithm. We see that with a minimum support threshold of 0.01%, PATD achieves a very good and significant performance compared to other presented algorithm. This difference in the performance is better illustrated in Figure 7(b). In fact, The high performance behavior of PATD algorithm in both response time and scalability, illustrates the high impact of a careful and clever placement of the data on a parallel distributed environment (thanks to the efficient and clever IBDP data partitioning technique). In this experiment, the time for generating the overlapping data partitions using IBDP (which is done only once) is fast, compared to the mining time. Actually, it takes 289 seconds to generate 500 overlapping data partitions.

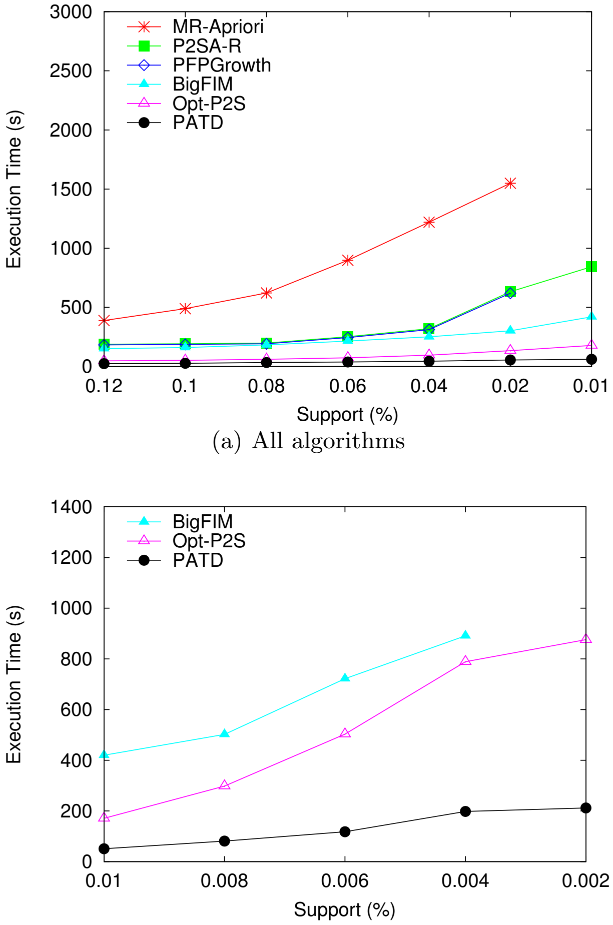
Figure 7(b) illustrates the differences between the three algorithms that scale in Figure 7(a). We see that BigFIM algorithm does not scale below a minimum



(a) All algorithms

(b) Focus on scalable algorithms

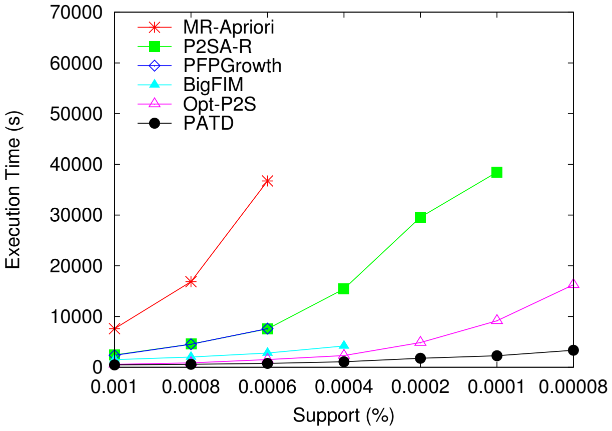
Fig. 7: Runtime and scalability on English Wikipedia data set



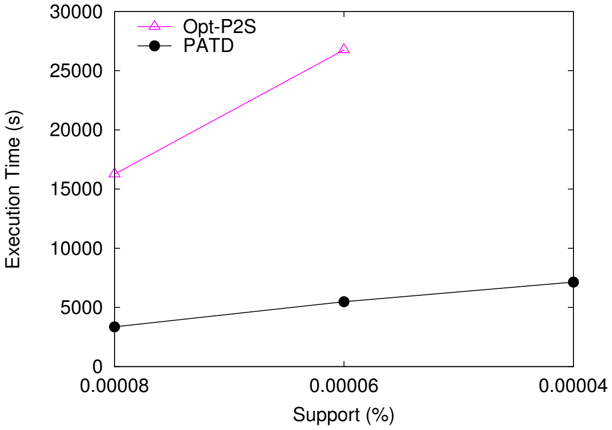
(a) All algorithms

(b) Focus on scalable algorithms

Fig. 8: Runtime and scalability on Amazon Reviews data set



(a) All algorithms



(b) Focus on scalable algorithms

Fig. 9: Runtime and scalability on ClueWeb data set

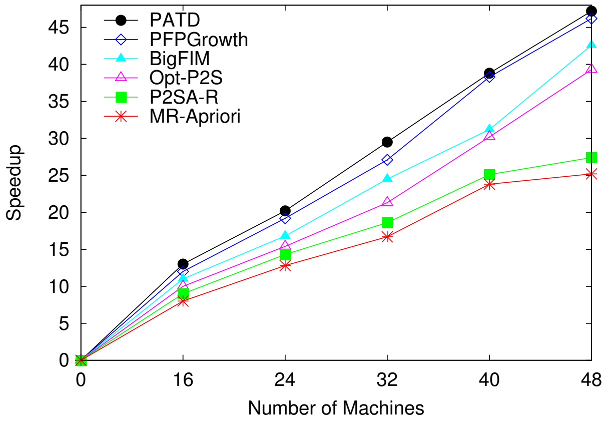


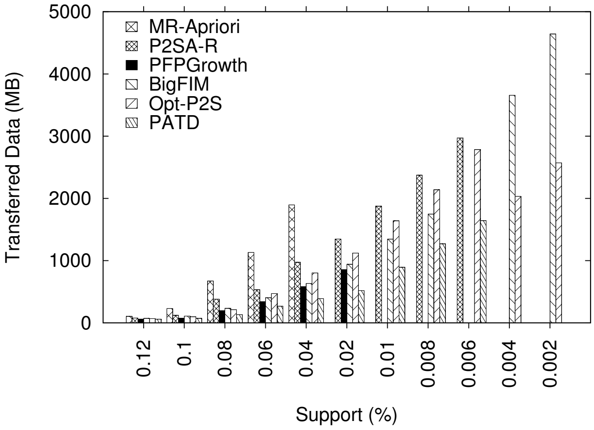
Fig. 10: Speedup on the English Wikipedia data set

support threshold of 0.006%. Looking at other presented algorithms, we see that Opt-P2S continues to scale with 0.002% while it is outperformed by PATD in terms of the response time. With a minimum support of 0.002%, we observe a big difference in the execution time between PATD and Opt-P2S. This very good performance of PATD is due to its clever and simple mining principle, and its simple one MapReduce job property that allows for a very fast mining of frequent itemsets.

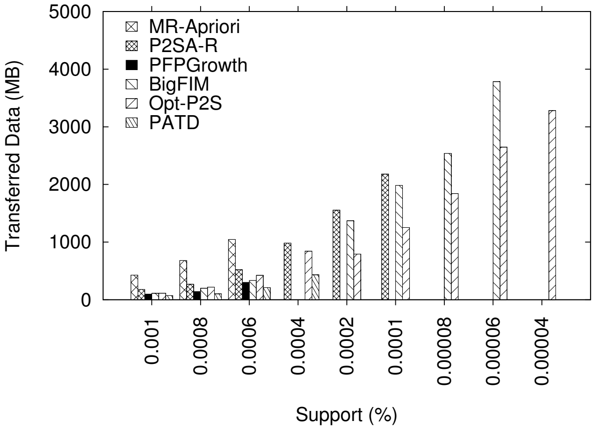
Figures 8(a) and 8(b) report our experimental results on the Amazon Reviews data set. Figure 8(a) gives a complete view on algorithms performances for a minimum support ranging from 0.12% to 0.01%. By looking at the performance of the MR-Apriori algorithm, we see that below a minimum support of 0.02%, MR-Apriori does not scale. In the other side, we see that P2SA-R and BigFIM continue scaling, while PFPGrowth does not. Although the scalability of these two PFIM algorithms, they are outperformed by Opt-P2S which is in turn outperformed by PATD algorithm.

In this experiment using the Amazon Reviews data set, the running time of IBDP to builds 500 overlapping data partitions is 243 seconds. This step is done once for all and we do not consider this running time when reporting the performance of PATD in Figures 8(a) and 8(b).

Figure 8(b) shows the differences between the algorithms that scale in Figure 8(a). We see that BigFIM algorithm does not scale below a minimum support of 0.004%. Looking at other presented algorithms, we see that Opt-P2S continues to scale with 0.002% while it is outperformed by PATD. With a minimum support of 0.002%, we observe a big difference in the execution time between PATD and Opt-P2S. This very good performance of PATD is due to its clever and simple mining principle, and its simple one MapReduce job property that allows for a very fast mining of frequent itemsets.



(a) English Wikipedia data set



(b) ClueWeb data set

Fig. 11: Data communication

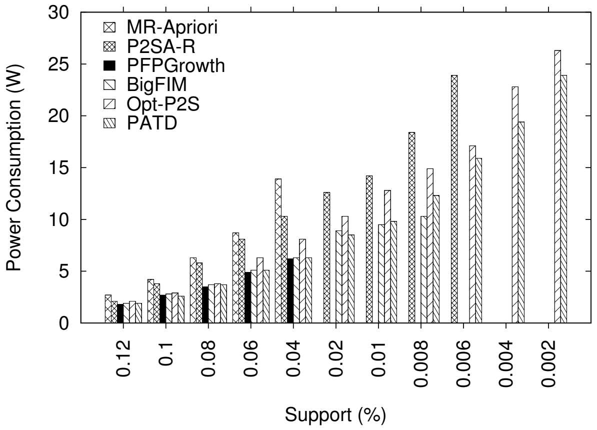


Fig. 12: Energy consumption

In Figures 9(a) and 9(b), same experiments have been carried out on the ClueWeb data set. We see that the same order between all algorithms is kept as in Figures 7(a), 7(b), 8(a) and 8(b). We distinguish three sets of algorithms. The first one consists of MR-Apriori which is unable to scale and impossible to be applied with this large data set. In the second set, we see that PFPGrowth presents the same limitations as on the Wikipedia data set in Figure 7(a). We can see that PFPGrowth performance is very close to P2SA-R. By decreasing further the minimum support threshold PFPGrowth becomes unable to scale, while, P2SA-R continues scaling until it stops executing with a minimum support of 0.0001%. In the third set of algorithms, we see Opt-P2S and PATD scale until 0.00008%. By further decreasing the minimum support threshold as shown in Figure 9(b), we observe a very good performance of PATD compared to Opt-P2S. The Opt-P2S algorithm becomes inoperative with a minimum support below 0.00006%, while PATD continues scaling very well. This big difference in the performance behavior between PATD and all other presented algorithms shows the high capacity of PATD in terms of scaling and response time. Particularly, in this experiment with this very large data set, it is worth to mention that the time for generating the overlapping data partitions using IBDP takes 1123 seconds to build 500 overlapping data partitions on this data set of one terabytes.

With both, Gigabytes and Terabytes of data, PATD gives a very good and significant performance. Whatever the data set size of our experiments, the number of transactions, and the minimum support, PATD scales and achieves very good results.

4.4.2 Speedup. Let us investigate the impact of varying the number of nodes on the speedup of the PATD algorithm and the other presented alternatives. In our experiments, we compute the speedup s by considering the centralized and the parallel execution of each algorithm i.e., the speedup s is $s = \frac{T_1}{T_m}$, where

T_1 represents the execution time of the algorithm on a single machine, and T_m accounts for the execution time of the same algorithm on m machines.

Figure 10 reports our experimental result on the English Wikipedia data set. By fixing the minimum support to 0.04% (i.e., to consider all the algorithms) and varying the number of used machines from 16 up to 48, we see that the speedup of MR-Apriori is not highly impacted by increasing the number of nodes. Although the availability of more resources (e.g., memory), MR-Apriori (straightforward implementation of Apriori) needs to scan the database multiple times in a serial manner. Looking at the mining principle of MR-Apriori, the increase in the number of the machines impacts the data communication overhead which degrades the speedup performance (e.g., as shown in Figure 10, by using 40 to 48, there is no much gain in the speedup).

On the other side, we see that P2SA-R has a better speedup compared to MR-Apriori. This is due to its 2-jobs compact architecture to extract the frequent itemsets. However, each data partition (i.e., each mapper at the first round) of P2SA-R needs to be mined using Apriori algorithm i.e., each data partition is scanned multiple times (cannot be parallelized), and this impacts the overall speedup of the algorithm. By looking at the Opt-P2S algorithm, we see that its speedup is better compared to both MR-Apriori and P2SA-R. This is because, the CDAR algorithm being applied at each data partition (with STDP) at the first round of Opt-P2S does not scan the data partition in a sequential manner as P2SA-R and MR-Apriori do. In addition, Opt-P2S benefits from the parallelism (i.e., resources) particularly when generating the itemset subsets. However, the main problem that impacts its speedup performance is the data communication. Meanwhile, we see that the BigFIM algorithm gives better speedup than previously mentioned algorithm, however it is outperformed by PFPGrowth. Interestingly, we see that PATD algorithm gives very good speedup compared to PFPGrowth. This is due to the fact that PATD does not allow much data to be communicated that impacts the overall speedup performance. By increasing the number of the machines, PATD highly benefits from the available resources.

4.4.3 Data Communication and Energy Consumption. Let us now study the amount of transferred data over the network when executing different PFIM algorithms. Figure 11(a) shows the transferred data (in mega bytes) of each presented algorithm on the Wikipedia data set. We observe in this figure that MR-Apriori has the highest peak. This is simply due to its multiple round of MapReduce executions. In other hand, we see that P2SA-R, BigFIM, Opt-P2S and PFPGrowth represent smaller peaks. Among all the performances of the presented algorithms in Figure 11(a), we clearly distinguish the performance of PATD algorithm. We can see that whatever the used *MinSup*, PATD does not allow much data transfer compared to other presented PFIM alternatives. This is because PATD does not rely on chains of jobs (i.e., just one simple job). In addition, unlike other presented PFIM algorithms, PATD limits the mappers from emitting non *frequent itemsets* (i.e., it does not duplicate the data). Therefore, PATD algorithm does not allow the transmission of useless data (*itemsets*).

In Figure 11(b), we report the results of the same experiment on the ClueWeb data set. We observe that PATD algorithm always has the lowest peak in terms of transferred data compared to other algorithms.

We also measured the energy power consumption of the compared algorithms during their execution. For measuring the power consumption, we used the Grid5000 [16] tools that allow us to measure the power consumption of the nodes during a job execution. Figure 12 shows the total amount of the power consumption of each algorithm. We observe that the energy power consumption increases when decreasing the minimum support threshold for each algorithm. We see that PATD still gives a lower consumption compared to other presented alternatives. Taking the advantage from its parallel design, PATD allows a high parallel computational execution. This, impacts the mining runtime to be fast, which is in turn, allows for a fast convergence of the algorithm and thus, lower energy consumption. PATD also transfers less data over the network, and this is another reason for its lower energy consumption.

5 Related Work

In data mining literature, several endeavors have been made to improve the performance of frequent itemset mining (FIM) algorithms [21], [22], [23]. Due to the explosive growth of data, an efficient parallel design of FIM algorithms has been highly required to handle large volumes of data.

Candidate Distribution algorithm [24] accounts for a parallel frequent itemset mining process that is based on data partitioning approach. The algorithm partitions the itemset candidates among the different processors, then redistributes the database transactions among the processor in such a way that each data partition can be mined independently from other partitions. However, The algorithm uses Apriori algorithm to mine each data partition, this results in multiple scans of each data partition which degrades the overall performance. In other hand, our proposed PATD algorithm is based on CDAR [11] algorithm to mine each data partition independently. Each one of these data partitions contains similar transactions, and this strategy highly increases the efficiency and scalability of the mining process.

Intelligent Data Distribution (IDD for short) algorithm [25] is an improvement of the Data Distribution (DD for short) algorithm [24]. Likewise, DD algorithm was proposed to solve the limitations (i.e., memory issues with high number of items) of the Count Distribution (CD for short) algorithm [24]. To this end, the DD algorithm partitions the the set of candidate itemsets among the processors in a round robin fashion. DD exploits better the available memory than the CD algorithm, however the DD algorithm has accounted for several flaws, particularly the cost of the data communication between the processors (i.e., sending the locally stored portions of the database transactions) which highly degrades the overall mining performance. IDD algorithm solves the problems presented in the DD algorithm by sending the locally stored portions of the database to other processors by using the ring-based all-to-all broadcast [26].

FPGrowth algorithm [23] has shown an efficient scale-up compared to other FIM algorithms , it has been worth coming up with a parallel version of FP-Growth [14] (i.e., PFPGrowth). The partitioning scheme of PFPGrowth follows that of FPGrowth, in the sense that FPGrowth divides the search space into "families of itemsets". The principle of FPGrowth is to explore the search space according to the F-List (the list of frequent items, sorted by decreasing order). For each frequent item in the reverse order of frequency, FPGrowth constructs its conditional pattern-base, and then its conditional FP-tree. The process is repeated on each newly created conditional FP-tree. The partitioning principle of PFPGrowth follows the construction of conditional FP-Trees. For each item, a partition that corresponds to the conditional tree is created and then mined for frequent itemsets. The principle of PATD is different since we build independent local databases that i) will locally ensure the extraction of a subset of the global frequent itemsets and ii) guarantee that the union of local frequent itemsets is exactly the set of global frequent itemsets. Although, PFPGrowth is distinguishable with its fast mining process, it has several flaws. In particular, with very low minimum support $MinSup$, PFPGrowth may run out of memory as illustrated by our experiments in Section 4.

BigFIM [20] algorithm has been proposed to cover the limitations of PF-PGrowth. This algorithm has shown better performance in terms of scalability compared to PFPGrowth. It represents a hybrid method between Apriori [9] and Eclat algorithm [27]. BigFIM first uses Apriori to extract the frequent itemsets of length k and later on switches to Eclat when the projected databases fit into the memory.

PARMA algorithm [28] uses an approximation in order to determine the list of $frequent\ itemsets$. It has shown better running time and scale-up than PFPGrowth. However, PARMA algorithm does not return an exhaustive list of $frequent\ itemsets$, it only approximates them.

A parallel version of Apriori algorithm [21] 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. To overcome conventional FIM issues and limits, a novel FIM technique, namely CDAR has been proposed in [11]. This algorithm uses a top down approach in order to determine the list of $frequent\ itemsets$. CDAR algorithm avoids the generation of candidates and renders the mining process more simple, by dividing the *database* into groups of transactions of equal *sizes*. Although, CDAR algorithm [11] has shown significant performance improvement in mining FIM, yet there has been no proposed parallel version of it.

Another FIM technique, called SON, has been proposed in [10], consists of dividing the *database* into n data partitions. Its mining process starts by searching the local frequent itemsets in each data partition independently. Then, it 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 address the problem of parallel frequent itemset mining (PFIM) in Big Data [2]. In particular, we take our inspiration from SON [10] algorithm. We propose two different parallel MapReduce based solutions that allow a fast and efficient parallel mining of frequent itemsets in very large databases.

To this end, we call for different data placement strategies in a massively distributed environment as efficient solutions for optimizing the mining process of PFIM. In this setting, we relate our work on data placement to data partitioning as a key idea that allows a customized data placement in massively distributed environments. We have used the PatoH [13] tool for graph partitioning, however, other various techniques and approaches can be used. In [29], 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 [29] to perform the STDP (refer to Section 3.1.2 for more details) partitioning efficiently and quickly after arrival of each new data to the *database*.

6 Conclusion

In this paper, we identified and studied 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. We proposed a reliable and efficient MapReduce based parallel frequent itemset algorithm, namely PATD, that has shown significant efficiency in terms of; i) runtime and scalability; i) low data communication; and low energy consumption. PATD algorithm takes the advantage of an efficient data partitioning technique namely IBDP. IBDP data partitioning strategy allows for an optimized data placement on MapReduce. This placement technique has not been investigated before this work. It allows PATD algorithm to exhaustively and quickly mine very large databases. Such ability to use very low minimum supports is mandatory when dealing with Big Data and particularly hundreds of Gigabytes like what we have done in our experiments. Our results show that PATD algorithm dramatically outperforms other existing PFIM alternatives, and makes the difference between an inoperative and a successful extraction.

7 Appendix

7.1 Two Rounds Closed Itemset Mining

Closed itemsets present a compact representation of the whole set of frequent itemsets. Very efficient solutions have proposed for their extraction, with orders of magnitude in performance improvements. Therefore, one would be tempted to apply frequent closed itemset mining algorithms as a local solution for pattern extraction in the first step of a 2 job-schema mining algorithm such as P2S or PATD. However, frequent closed itemset have very constraining characteristics

TID	A	B	C	D	E	F
1	X	X	X	X	X	
2	X		X	X	X	X
3	X	X	X		X	
4	X		X		X	X
5			X	X		X
6	X		X	X	X	X
7	X		X	X	X	
8			X	X	X	X
9	X	X	X	X		
10	X	X	X		X	
11		X	X	X		
12		X	X	X		

Table 2: Database $\mathcal{D}$

that may prevent from using them is a 2-job approach. Let us consider the illustration given by Example 5 to show that a global frequent closed itemset might never be a local frequent closed itemset (in no partition).

Example 5. Table 2 presents a database $\mathcal{D}$ with 12 transactions. Suppose we divide $\mathcal{D}$ into four data partitions $P_1 = \{T_1, T_2, T_3\}$, $P_2 = \{T_4, T_5, T_6\}$, $P_1 = \{T_1, T_2, T_3\}$ and $P_2 = \{T_4, T_5, T_6\}$, where $\mathcal{D} = P_1 \cup P_2 \cup P_3 \cup P_4$.

The set of global frequent closed itemsets on $\mathcal{D}$, along with their support is: $\{(C : 12)(A, C : 8)(C, D : 8)(C, E : 8)\}$. In this set, the only itemset that is also a local frequent closed itemset is (C, D), because it is supported by 3 transactions in P_3 and it has no superset with the same support. All the remaining global frequent closed itemsets are either unfrequent or not closed in the partitions. Let us consider, for instance, the itemset (C). In partition P_1 it has the same support as (A, C, E). In partition P_2 it has the same support as (C, F). In partition P_3 it has the same support as (C, D). In partition P_4 it has the same support as (B, C).

This simple counter example shows that, even though closed frequent itemsets are an appealing research track for distributed environments, their usage call for particular care. It would be interesting to investigate their properties in a distributed scheme like PATD, but it calls for proofs, or counter examples, of their compatibility with such a scheme.

References

1. Michael Berry. *Survey of Text Mining Clustering, Classification, and Retrieval.* Springer New York, New York, NY, 2004.
2. Alexandros Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *Proc. VLDB Endow.*, 5(12):2032–2033, August 2012.
3. Wei Fan and Albert Bifet. Mining big data: Current status, and forecast to the future. *SIGKDD Explor. Newsl.*, 14(2):1–5, April 2013.

4. Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.
5. Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.
6. Hadoop. `http://hadoop.apache.org`, 2014.
7. 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.
8. Saber Salah, Reza Akbarinia, and Florent Masseglia. Data partitioning for fast mining of frequent itemsets in massively distributed environments. In *26th International Conference on Database and Expert Systems Applications (DEXA)*, 2015.
9. 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.
10. 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.
11. Yuh-Jiuan Tsay and Ya-Wen Chang-Chien. An efficient cluster and decomposition algorithm for mining association rules. *Inf. Sci. Inf. Comput. Sci.*, 160(1-4):161–171, March 2004.
12. Shimon Even. *Graph algorithms.* Computer Science Press, Potomac, Md, 1979.
13. Patoh. http://bmi.osu.edu/ umit/PaToH/manual.pdf, 2011.
14. 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 2008 ACM Conference on Recommender Systems, RecSys 2008, Lausanne, Switzerland, October 23-25, 2008*, pages 107–114. ACM, 2008.
15. Sean Owen. *Mahout in action.* Manning Publications Co, Shelter Island, N.Y, 2012.
16. Grid5000. `https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`.
17. English wikipedia articles. `http://dumps.wikimedia.org/enwiki/latest`, 2014.
18. Amazon. `http://snap.stanford.edu/data/web-Amazon-links.html`.
19. The clueweb09 dataset. `http://www.lemurproject.org/clueweb09.php/`, 2009.
20. S. Moens, E. Aksehirli, and B. Goethals. Frequent itemset mining for big data. In *Big Data, 2013 IEEE International Conference on*, pages 111–118, Oct 2013.
21. Rajaraman Anand. *Mining of massive datasets.* Cambridge University Press, New York, N.Y. Cambridge, 2012.
22. Wei Song, Bingru Yang, and Zhangyan Xu. Index-bittablefi: An improved algorithm for mining frequent itemsets. *Knowl.-Based Syst.*, 21(6):507–513, 2008.
23. Han, Pei, and Yin. Mining frequent patterns without candidate generation. *SIGMODREC: ACM SIGMOD Record*, 29, 2000.
24. Rakesh Agrawal and John C. Shafer. Parallel mining of association rules. *IEEE Trans. on Knowl. and Data Eng.*, 8(6):962–969, December 1996.
25. Mohammed J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, 7(4):14–25, October 1999.
26. Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms.* Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

27. Mohammed J. Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, and Wei Li. Parallel algorithms for discovery of association rules. *Data Min. Knowl. Discov.*, 1(4):343–373, December 1997.
28. 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.
29. 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.