Data Partitioning for Fast Mining of Frequent Itemsets in Massively Distributed Environments

Saber Salah*, Reza Akbarinia, and Florent Masseglia

Zenith team
INRIA & LIRMM - University of Montpellier, France
first.last@inria.fr

Abstract. Frequent itemset mining (FIM) is one of the fundamental cornerstones in data mining. While, the problem of FIM has been thoroughly studied, few of both standard and improved solutions scale. This is mainly the case when i) the amount of data tends to be very large and/or ii) the minimum support ($MinSup$) threshold is very low. In this paper, we propose a highly scalable, parallel frequent itemset mining (PFIM) algorithm, namely Parallel Absolute Top Down (PATD). PATD algorithm renders the mining process of very large databases (up to Terabytes of data) 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. Based on a clever and efficient data partitioning strategy, namely Item Based Data Partitioning (IBDP), PATD algorithm mines each data partition independently, relying on an absolute minimum support ($AMinSup$) instead of a relative one. PATD has been extensively evaluated using real-world data sets. Our experimental results suggest that PATD algorithm is significantly more efficient and scalable than alternative approaches.

Keywords: Machine Learning, Data Mining, Frequent Itemset, Big Data, MapReduce

1 Introduction

Since a few decades, the amount of data in the world and our lives seems ever-increasing. Nowadays, we are completely overwhelmed with data, it comes from different sources, such as social networks, sensors, etc. 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 [1]. The processing of this massive amount of data, helps leveraging and uncovering hidden relationships, and brings up new, and useful information. Itemsets are one of these tackled levers and consist in frequent correlations of features. Their discovery is known as Frequent itemset mining (FIM for short), and 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 as books and different other products. In science and engineering, FIM

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

can be used to analyze such different scientific parameters (e.g, based on their regularities). Finally, FIM methods can help to perform other data mining tasks such as text mining [2], for instance, and, as it will be better illustrated by our experiments in Section 4, FIM can be used to figure out frequent co-occurrences of words in a very large-scale text database. However, the manipulation and processing of large-scale databases have opened up new challenges in data mining [3]. First, the data is no longer located in one computer, instead, it is distributed over several machines. Thus, a parallel and efficient design of FIM algorithms must be taken into account. Second, parallel frequent itemset mining (PFIM for short) algorithms should scale with very large data and therefore very low $MinSup$ threshold. 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. They have gained increasing popularity, as shown by the tremendous success of Hadoop [6], an open-source implementation. Despite the robust parallelism setting that these solutions 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 standard PFIM algorithms do not scale. Hence, 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 best 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 and, 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. We investigate an efficient combination between a mining process (i.e, a PFIM algorithm) and an efficient placement of data, and study its impact on the global mining process.

We have designed and developed a powerful data partitioning technique, namely Item Based Data Partitioning (IBDP for short). One of the drawbacks of existing PFIM algorithms is to settle for a disjoint placement. IBDP allows, for a given item i to be placed in more than one mapper if necessary. Taking the advantages from this clever data partitioning strategy, we have designed and developed a MapReduce based PFIM algorithm, namely Parallel Absolute Top Down Algorithm (PATD for short), which is capable to mine a very large-scale database in just one simple and fast MapReduce job. We have evaluated the performance of PATD through extensive experiments over two massive datasets (up to one Terabyte and half a billion Web pages). Our results show that PATD scales very well on large databases with very low minimum support, compared to other PFIM alternative algorithms.

The rest of the paper is organized as follows. Section 2 gives an overview of FIM problem, basic used notations, and the necessary background. In Section 3, we propose our PATD algorithm and we depict its whole core working process. Section 4 reports on our experimental validation over real-world data sets. Section 5 discusses related work, and Section 6 concludes.

2 Definitions and Background

In this section, we set up the basic notations and terminology, that we are going to adopt in the rest of the paper.

The problem of FIM was first introduced in [7], and then manifold algorithms have been proposed to solve it. In definition 1, we adopt the notations used in [7].

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 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 the $database$ $\mathcal{D}$ 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. A $maximal frequent itemset$ is a frequent itemset that has no frequent superset.*

The FIM problem consists of extracting all frequent itemset from a database $\mathcal{D}$ with a minimum support $MinSup$ specified as a parameter.

Example 1. Let 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 no frequent items (and no frequent itemsets). 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 paper, we focus on 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 approach, however, we strongly believe that our proposal would have good performance results in other parallel frameworks too (*e.g.* Spark).

3 Parallel Absolute Top Down Algorithm

As we briefly mentioned in Section 1, using an efficient data placement technique, could significantly 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 consider the case when most of the workload of a PFIM algorithm is being performed on the mappers. In this case, the way the data is exposed to the mappers, could contribute to the efficiency and the performance of the whole mining process (i.e, invoked PFIM algorithm).

In this context, 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. We illustrate the impact of data placement techniques on the performance of PFIM algorithms, by considering particular PFIM algorithms which are based on two MapReduce jobs schema (2-Jobs schema for short).

In this section, first, we investigate the impact of partitioning data (i.e, impact of data placement) on 2-Jobs schema. Second, we introduce our IBDP method for data partitioning, and then we detail its working logic and principle. Finally, we introduce PATD algorithm and elucidate its design and core mining process in MapReduce.

3.1 Impact of Partitioning Data on 2-Jobs Schema

Performing a mining process in two steps was first proposed in [8] and it was designated for centralized environments. SON algorithm [8] divides a mining process as follows:

- **Step 1:** Divide the input database $\mathcal{D}$ into n data chunks (i.e, data splits), where $\mathcal{D} = \{P_1, P_2, \ldots, P_n\}$. Then, mine each data chunk (P_i) in the memory, based on a local minimum support ($LMinSup$), and a specific FIM algorithm. Thus, the first step of SON algorithm is to determine a list of local frequent itemsets (LFI).

- **Step 2:** From previous step result, proceed by filtering the local frequent itemsets in LFI list, based on a global minimum support $GMinSup$. This may be done with a scan on $\mathcal{D}$ and checking the frequency of each itemset is LFI. The main idea is that any frequent itemset on $\mathcal{D}$ will be frequent on at least one chunk P_i and will be found in LFI. Then, return a list of global frequent itemsets (GFI) which is a subset of LFI ($GFI \subseteq LFI$).

In a massively distributed environment, the main bottleneck of such 2-Jobs schema PFIM algorithm is its first execution phase, where a FIM algorithm has to be executed on the chunks. The choice of this algorithm is therefore crucial. Relying on SON mining principle, we have implemented a parallel version of CDAR [9] and Apriori [7] algorithms on MapReduce, namely, Parallel Two Round CDAR (P2RC) and Parallel Two Round Apriori (P2RA) respectively. Each version makes use of CDAR or Apriori on the chunks in the first phase. P2RC divides the mining process into two MapReduce jobs as follows:

- **Job 1:** In the first phase, the principle of CDAR (see [9] for more details) is adapted to a distributed environment. A global minimum support $GMinSup$ Δ is passed to each mapper. The latter deduces a local minimum support $LMinSup$ δ from Δ and its input data split (i.e, number of transaction in the input split). Then, each mapper divides its input data split ($\mathcal{S}$) into n data partitions, $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$. Each partition S_i in $\mathcal{S}$ holds only transactions that have length i, where the length of a

transaction is the number of items in it. Then, the mapper starts mining the data partitions $S_i ... S_n$ according to transaction lengths in decreasing order. A transaction in each partition accounts for an itemset. If a transaction T is frequent ($Support(T) \geq \delta$) in partition S_{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 frequency of all transactions T in S_{i+1}, the process continues by generating i subsets of all T in $Temp$ and adds the i generated subsets to partition S_i. The same mining process is carried out until visiting all partitions S_i in $\mathcal{S}$. Before counting the $Support$ of a transaction T, an inclusion test of T in L is performed. If the test returns true, the computation of the $Support$ of T will not be considered as T is already in L which means frequent. Each mapper emits all its local frequent itemsets to the reducer. The reducer writes all local frequent itemsets to the distributed file system.

- **Job 2:** Each mapper takes a data split $\mathcal{S}$ and a list of local frequent itemsets LFI. Each mapper determines the inclusion of LFI elements in each transaction of $\mathcal{S}$. If there is an inclusion, then the mapper emits the itemset as a key and one as value (key: itemset, value: 1). A global minimum support $GMinSup\ \Delta$ is passed to the reducer. The reducer simply iterates over the values of each received key, and sums them up in variable sum. If ($sum \geq \Delta$), then the itemset under consideration is globally frequent.

As illustrated above, the main workload of P2RC algorithm is done on the mappers independently. Intuitively, the mapper that holds more homogeneous data (i.e, homogeneous transactions) will be faster. Actually, by referring to the mining principle of CDAR, a mapper that holds homogeneous transactions (i.e, similar transactions) allows for more itemset inclusions which in turn results in less subsets generation. Thus, placing each bucket of similar transactions (non-overlapping data partitions) on the mappers would improve the performance of P2RC algorithm. This data placement technique can be achieved by means of different data partitioning methods.

In contrast, the partitioning of data based on transaction similarities (STDP for short: Similar Transaction Data Partitioning), logically would not improve the performance of Parallel Two Round Apriori (P2RA), instead it should lower it. In this case, each mapper would hold a partition of data (i.e, data split) of similar transactions which allows for a high number of frequent itemsets in each mapper. This results in a higher number of itemset candidates generation. Interestingly, using a simple Random Transaction Data Partitioning (RTDP for short) to randomly place data on the mappers, should give the best performance of P2RA. Our experiments given in Section 4 clearly illustrate this intuition.

P2RC performs two MapReduce jobs to determine all frequent itemsets. Thus, PFIM algorithms that depend on SON process design duplicate the mining results. Also, at their first mining step (i.e, first MapReduce job), 2-Jobs schema PFIM algorithms output itemsets that are locally frequent, and there is no guarantee to be globally frequent. Hence, these algorithms amplify the number of transferred data (i.e, itemsets) between mappers and reducers.

To cover the above-mentioned issues, our major challenge is to limit the mining process to one simple job. This would guarantee low data communications, less energy

power consumption, and a fast mining process. In a distributed computational environment, we take the full advantage of the available massive storage space, CPU(s) etc.

3.2 IBDP: An Overlapping Data Partitioning Strategy

Our claim is that duplicating the data on the mappers allows for a better accuracy in the first job and therefore leads to less infrequent itemsets (meaning less communications and fast processing). Consider a data placement with a high overlap (i.e, placement of data partitions that share several transactions), with for instance 10 overlapping data partitions, each holding 50% of the database. Obviously, there will be less globally infrequent itemsets in the first job (i.e, if an itemset is frequent on a mapper, then it is highly likely to be frequent on the whole database.). Unfortunately, this approach has some drawbacks, 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. However, 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 threshold Δ and X, a frequent itemset according to Δ 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 thresholds. The second expectation is that X can be extracted from S_X with Δ as a minimum threshold. 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 of non-overlapping data partitions $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$. Each one of these non-overlapping data partitions (i.e, $\bigcap_{i=1}^{n} S_i = \emptyset$)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 its "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 intercepts 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 and its parallel design on MapReduce, are given in Algorithm 1, while its principle is illustrated by Example

2.

Algorithm 1: IBDP

```
1  //Job1
   Input: Non-overlapping data partitions S = {S_1, S_2, ..., S_n} of a database D
   Output: Centroids
2  //Map Task 1
3  map( key: Split Name: K_1, value = Transaction (Text Line): V_1 )
4      - Tokenize V_1, to separate all items
5      emit (key: Item, value: Split Name)
6  //Reduce Task 1
7  reduce( key: Item, list(values) )
8      while values.hasNext() do
9          emit (key:(Split Name) values.next (Item))
10 //Job2
   Input: Database D
   Output: Overlapping Data Partitions
11 //Map Task 2
12 - Read previous job1 result once in a (key, values) data structure (DS), where key:
   SplitName and values: Items
13 map( key: Null: K_1, value = Transaction (Text Line): V_1 )
14     for SplitName in DS do if Items.Item ∩ V_1 ≠ ∅ then
15         emit (key: SplitName, value: V_1)
16 //Reduce Task 2
17 reduce( key: SplitName, list(values) )
18     while values.hasNext() do
19         emit (key: (SplitName), values.next: (Transaction))
```

- **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 a key. Then, it tokenizes each transaction (value) to determine different items and emits each item as a key coupled with its split name as value. The reducer aggregates over the keys (items) and emits each key (item) coupled with its different value (split name) in the list of values (split names).

- **Job 2** *Overlapping Partitions*: The format of the MapReduce output is set to "MultiFileOutput" in the driver class. In this case, the keys will denote the name of each overlapping data partition output (we override the "generateFileNameForKeyValue" function in MapReduce to return a string as 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. Then, 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 (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.

Example 2. Figure 1 shows a database $\mathcal{D}$ with 5 transactions. In this example, we have two non-overlapping data partitions 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 $\mathcal{D}$. This results in two overlapping data partitions in (4) where the intersections only are kept in (5). Finally, the maximal frequent itemsets are extracted in (6). Redundancy is used for the counting process of different itemsets. For instance, transaction efg is duplicated in both partitions in (5) where the upper version participates to the frequency counting of a and the lower version participates to the frequency counting of fg.

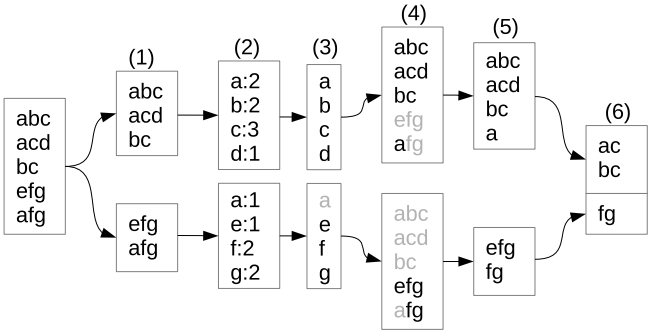


Fig. 1: Data partitioning process: (1) partitions of similar transactions are built; (2) centroids are extracted; (3) and filtered; (4) transaction are placed and filtered ; (5) to keep only the intersection of original transactions and centroids; (6) local frequent itemsets are also globally frequent.

3.3 1-Job Schema: Complete Approach

We take the full advantage from IBDP data partitioning strategy and propose a powerful and robust 1-Job Schema PFIM algorithm namely PATD. PATD algorithm limits the mining process of very large database to one simple MapReduce job and exploits the 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 $AMinSup$ Δ, the 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 $AMinSup$ Δ for all mappers, i.e, each overlapping data partition S_i is mined based on Δ. 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.

The main activities of mappers and reducers in PATD algorithm are as follows:

- **Mapper:** Each mapper is given a S_i, $i = 1...m$ overlapping data partition, and a global 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.

As illustrated in the mappers and reducers logic, PATD performs the mining process in one simple and efficient MapReduce job. These properties of PATD are drawn from the use of the robust and efficient data partitioning strategy IBDP.

Example 3. Lets take the example of Figure 1. Given an absolute minimum support $\Delta = 2$ (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 value. In our example, we have two overlapping data partitions (5). We consider two mappers m_1 and m_2, each one performs a complete CDAR with $\Delta = 2$. In Figure 1 (5) from bottom-up : mapper m_1 mines first overlapping data partition and returns {fg} as a frequent itemset. Alike, 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.

3.4 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 partitions produced by IBDP. Since, each 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 Δ, $\forall$ itemset x in $\mathcal{D}$ we have: $Support_{\mathcal{D}}(x) \geq \Delta \Leftrightarrow \exists\, \mathcal{P} \setminus Support_{\mathcal{P}}(x) \geq \Delta$ 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 $\exists\, \mathcal{P} \setminus Support_{\mathcal{P}}(x) \geq \Delta$.
Let denote by $\mathcal{X}$ the set of all unique items of $\mathcal{D}$. The intersection of all transactions $\{T_1, T_2, \ldots, T_n\}$ with $\mathcal{X}$ is $\mathcal{D}$. Thus, in this particular case, $Support_{\mathcal{D}}(x) \geq \Delta \Rightarrow \exists\, \mathcal{D} \setminus Support_{\mathcal{D}}(x) \geq \Delta$. If the set of unique items $\mathcal{X}$ is partitioned

into k partitions, then the intersection of each one of these k partitions with all $\{T_1, T_2, \ldots, T_n\}$ in $\mathcal{D}$, would result in a new data partition $\mathcal{P}$. Let denote by $\Pi = \{P_1, P_2, \ldots, P_k\}$, the set of all these new data partitions. For any given itemset x in $\mathcal{D}$, its total occurrence will be in one partition of Π, because, all items in $\mathcal{X}$ are shared among these partitions in Π. Therefore, $Support_{\mathcal{D}}(x) \geq \Delta \Rightarrow \exists\, I_P \setminus Support_{I_P}(x) \geq \Delta$

Next, we prove the inverse, i.e. if $\exists\, \mathcal{P} \setminus Support_{\mathcal{P}}(x) \geq \Delta$ then $Support_{\mathcal{D}}(x) \geq \Delta$.
This is done simply by using the fact that each 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 $\exists\, \mathcal{P} \setminus Support_{\mathcal{P}}(x) \geq \Delta \Rightarrow Support_{\mathcal{D}}(x) \geq \Delta$.
Therefore, we conclude that: $Support_{\mathcal{D}}(x) \geq \Delta \Leftrightarrow \exists\, \mathcal{P} \setminus Support_{\mathcal{P}}(x) \geq \Delta$.

4 Experiments

To assess the performance of PATD algorithm, we have carried out extensive experimental evaluations. In Section 4.1, we depict our experimental setup, and in Section 4.3 we investigate and discuss the results of our different experiments.

4.1 Experimental Setup

We implemented PATD, and all other presented algorithms on top of Hadoop-MapReduce, using Java programming language version 1.7 and Hadoop version 1.0.3. For comparing PATD performance with other PFIM alternatives, we implemented two bunches of algorithms. First, we followed SON algorithm design and implemented Parallel Two Round Apriori (P2RA) and Parallel Two Round CDAR (P2RC). These two PFIM algorithms are based on random transaction data partitioning (RTDP) and similar transaction data partitioning (STDP), respectively. Second, we designed and implemented a parallel version of standard Apriori algorithm [7], namely Parallel Apriori (PA). For comparison with PFP-Growth [10], we adopted the default implementation provided in the Mahout [11] machine learning library (Version 0.7).

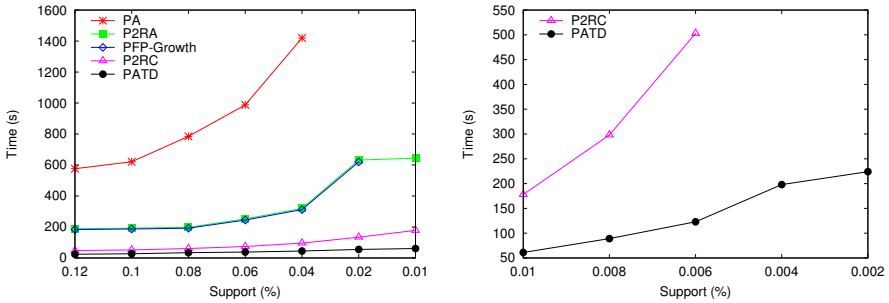
We carried out all our experiments on the Grid5000 [12] platform, which is a platform for large-scale data processing. We have used a cluster of 16 and 48 machines respectively for Wikipedia and ClueWeb data set experiments. Each machine is equipped with Linux operating system, 64 Gigabytes of main memory, Intel Xeon $X3440$ 4 core CPUs, and 320 Gigabytes SATA hard disk.

4.2 Real World Datasets

To better evaluate the performance of PATD algorithm, we used two real-world data sets. The first one is the 2014 English Wikipedia articles [13] having a total size of 49 Gigabytes, and composed of 5 million articles. The second one is a sample of ClueWeb English data set [14] with size of one Terabyte and having 632 million articles. For each data set, we performed a data cleaning task. We removed all English stop words from all

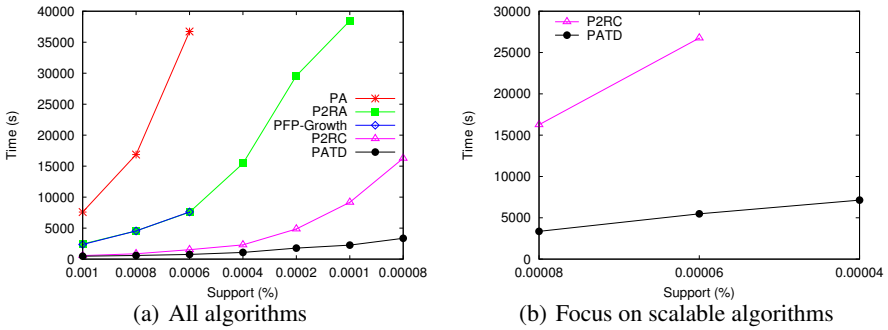
articles, we obtained data sets where each article represents a transaction (items are the corresponding words in the article) to each invoked PFIM algorithm in our experiments.

We vary the $MinSup$ parameter value for each PFIM algorithm. We evaluate each algorithm based on its response time, its total amount of transferred data, and its energy power consumption. In particular, we consider these three different measurements, when $MinSup$ is very low.



(a) All algorithms (b) Focus on scalable algorithms

Fig. 2: Runtime and scalability on English Wikipedia data set



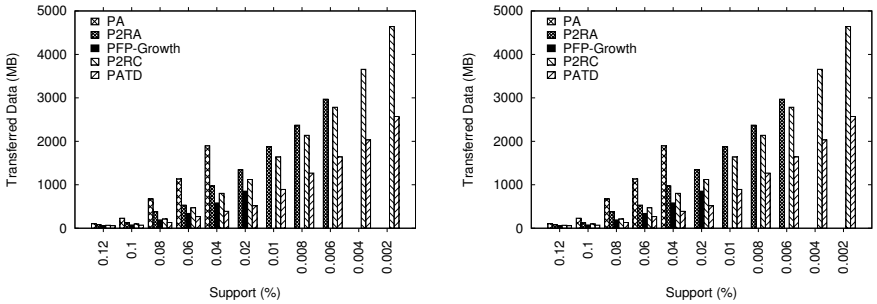
(a) All algorithms (b) Focus on scalable algorithms

Fig. 3: Runtime and scalability on ClueWeb data set

4.3 Runtime and Scalability

Figures 2 and 3 give a complete view of our experiments on both English Wikipedia and ClueWeb data sets. Figures 2(a) and 2(b) report our experimental results on the whole English Wikipedia data set. Figure 2(a) gives an entire view on algorithms performances for a minimum support varying from 0.12% to 0.01%. We see that PA algorithm runtime grows exponentially, and gets quickly very high compared to other presented PFIM algorithms. This exponential run-time reaches its highest value with 0.04% threshold. Below this threshold, PA needs more resources (e.g. memory) than what exists in our

tested machines, thus, it is impossible to extract frequent itemsets with this algorithm. Another interesting observation is that P2RA performance tends to be close to PFP-Growth until a minimum support of 0.02%. P2RA algorithm continues scaling with 0.01% while PFP-Growth does not. Although, P2RC scales with low minimum support values, PATD outperforms this algorithm in terms of running time. In particular, with a minimum support of 0.01% PATD algorithm outperforms all other presented PFIM algorithms. This difference in the performance is better illustrated in Figure 2(b).



(a) English Wikipedia data set (b) ClueWeb data set.

Fig. 4: Data communication

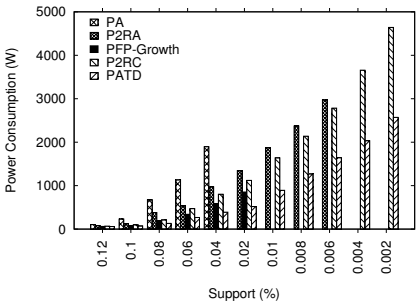


Fig. 5: Energy consumption

Figure 2(b) focuses on the differences between the four algorithms that scale in Figure 2(a). Although P2RC continues to scale with 0.002%, it is outperformed by PATD in terms of running time. With 0.002% threshold, we observe a big difference in the response time between PATD and P2RC. This very good performance of PATD is due to its clever and simple mining principle, and its simple MapReduce job property that allows a low mining time.

In Figures 3(a) and 3(b), similar experiments have been conducted on the ClueWeb data set. We observe that the same order between all algorithms is kept compared to Figures 2(a) and 2(b). There are three bunches of algorithms. One made of PA which cannot reasonably applied to this data set, whatever the minimum support. In the second

bunch, we see that PFP-Growth suffers from the same limitations as could be observed on the Wikipedia data set in Figure 2(a), and it follows a behavior that is very similar to that of P2RA, until it becomes impossible to execute. P2RA continues scaling until stops executing with a minimum support of 0.0001%. In the third bunch of algorithms, we see P2RC and PATD scale until 0.00008%. We decreased the minimum support parameter, and we zoom on these two algorithms. As shown in Figure 3(b), we observe a very good performance of PATD compared to P2RC. The P2RC 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 capability of PATD in terms of scaling and response time. With both, Gigabytes and Terabytes of data, PATD gives a very good and significant performance. Whatever, the data size, the number of transactions, and the minimum support, PATD scales and achieves very good results.

4.4 Data Communication and Energy Consumption

Let's now study the amount of transferred data for each PFIM algorithm. Figure 4(a) shows the transferred data (in mega bytes) of each presented algorithm on English Wikipedia data set. We observe that PA has the highest peak, this is simply due to its several round of MapReduce executions. In other hand, we see that P2RA, P2RC and PFP-Growth represent smaller peaks. Among all the presented algorithms in Figure 4(a), we clearly distinguish PATD algorithm. We can see that whatever the used $MinSup$, PATD does not allow much data transfer compared to other algorithms. This is because PATD does not rely on chains of jobs like other presented alternatives. In addition, contrary to other PFIM algorithms, PATD limits the mappers from emitting non frequent itemsets. Therefore, PATD algorithm does not allow the transmission of useless data (itemsets).

In Figure 4(b), we report the results of the same experiment on ClueWeb data set. We observe that PATD algorithm always has the lowest peak in terms of transferred data comparing to other algorithms.

We also measured the energy consumption of the compared algorithms during their execution. We used the Grid5000 tools that measure the power consumption of the nodes during a job execution. Figure 5 shows the total amount of the power consumption of each presented PFIM algorithm. We observe in Figure 5, that the consumption increases when decreasing the minimum support for each algorithm. We see that PATD still gives a lower consumption comparing to other algorithms. Taking the advantage from its parallel design, PATD allows a high parallel computational execution. This impacts the mining runtime to be fast and allows for a fast convergence of the algorithm and thus, a less consumption of the energy. PATD also transfers less data over the network, and this is another reason for its lower energy consumption.

5 Related Work

In the literature, several endeavors have been made to improve the performance of FIM algorithms [15] [16].

Recently, and due to the explosive data growth, an efficient parallel design of FIM algorithms has been highly required. FP-Growth algorithm [16] 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 [10] (i.e. PFP-Growth). Although, PFP-Growth is distinguishable with its fast mining process, PFP-Growth has accounted for different flaws. In particular, with very low minimum support, PFP-Growth may run out of memory as illustrated by our experiments in Section 4. PARMA algorithm [17] 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, PARMA algorithm does not return an exhaustive list of frequent itemsets, it only approximates them.

A parallel version of Apriori algorithm [18] 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 [9]. This algorithm uses a transactions grouping based approach to determine the list of frequent itemsets. CDAR avoids the generation of candidates and renders the mining process more simple, by dividing the database into groups of transactions. Although, CDAR [9] 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 [8], 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 have focused on the data placement as a fundamental and essential mining factor in MapReduce. We proposed the PATD algorithm that not only reduces the total response time of FIM process, but also its communication cost and energy consumption.

6 Conclusion

We proposed a reliable and efficient MapReduce based parallel frequent itemset algorithm, namely PATD, that has shown significantly efficient in terms of runtime and scalability, data communication as well as energy consumption. PATD takes the advantage of the efficient data partitioning technique IBDP. IBDP allows for an optimized data placement on MapReduce. 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 outperforms other existing PFIM alternatives, and makes the difference between an inoperative and a successful extraction.

References

1. Alexandros Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *Proc. VLDB Endow.*, 5(12):2032–2033, August 2012.

2. Michael Berry. *Survey of Text Mining Clustering, Classification, and Retrieval*. Springer New York, New York, NY, 2004.
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.
6. Hadoop.
7. 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.
8. 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.
9. 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.
10. 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.
11. Sean Owen. *Mahout in action*. Manning Publications Co, Shelter Island, N.Y, 2012.
12. Grid5000. `https://www.grid5000.fr/mediawiki/index.php/Grid5000:Home`.
13. English wikipedia articles. `http://dumps.wikimedia.org/enwiki/latest`, 2014.
14. The clueweb09 dataset. `http://www.lemurproject.org/clueweb09.php/`, 2009.
15. Wei Song, Bingru Yang, and Zhangyan Xu. Index-bittablefi: An improved algorithm for mining frequent itemsets. *Knowl.-Based Syst.*, 21(6):507–513, 2008.
16. Han, Pei, and Yin. Mining frequent patterns without candidate generation. *SIGMODREC: ACM SIGMOD Record*, 29, 2000.
17. 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.
18. Rajaraman Anand. *Mining of massive datasets*. Cambridge University Press, New York, N.Y. Cambridge, 2012.