MapFIM+: Memory Aware Parallelized Frequent Itemset Mining in Very Large Datasets

Khanh-Chuong Duong[1,2], Mostafa Bamha[2], Arnaud Giacometti[1], Dominique Li[1], Arnaud Soulet[1] and Christel Vrain[2]

[1] Université de Tours, LIFAT EA 6300, Blois, France
{arnaud.giacometti,dominique.li,arnaud.soulet}@univ-tours.fr

[2] Université d'Orléans, INSA Centre Val de Loire, LIFO EA 4022, France
{khanh-chuong.duong,mostafa.bamha,christel.vrain}@univ-orleans.fr

Abstract

Mining frequent itemsets in large datasets has received much attention in recent years relying on the MapReduce programming model. For instance, many efficient frequent itemset mining (a.k.a. FIM) algorithms have been parallelized to MapReduce principle such as *Parallel Apriori*, *Parallel FP-Growth* and *Dist-Eclat*. However, most approaches focus on job partitioning and/or load balancing without considering the extensibility depending on required memory assumptions. Thus, a challenge in designing parallel FIM algorithms consists therefore in finding ways to guarantee that the data structures used during the mining process always fits in the local memory of working nodes during all computation steps. In this paper, we propose MapFIM+, a two-phase approach to frequent itemset mining in very large datasets benefiting both from a MapReduce-based distributed Apriori method and local in-memory FIM mining methods. In our approach, MapReduce is first used to generate frequent itemsets until getting local memory-fitted prefix-projected databases, and an optimized local in-memory mining process is then launched to generate all remaining frequent itemsets from each prefix-projected database on individual processing nodes. Indeed, MapFIM+ improves our previous algorithm MapFIM by using an accurate evaluation of prefix-projected database sizes during the MapReduce phase, and this improvement makes MapFIM+ more efficient especially for databases leading to huge candidate sets by significantly reducing communication and disks I/O costs. Our performance evaluations show that MapFIM+ is more efficient and more extensible than existing MapReduce based frequent itemset mining approaches.

Keywords: Frequent itemset mining, MapReduce programming model, Distributed file systems, Hadoop framework.

1 Introduction

Frequent pattern mining [2] is an important field of Knowledge Discovery in Databases that aims at extracting itemsets occurring frequently inside database entries (as transactions, event sets, etc.). Usually, a minimum support threshold is fixed in this problem and frequent patterns are defined as patterns whose frequency is greater than this threshold. All the algorithms rely on an important anti-monotonicity property for pruning the search space stating that when an itemset is extended then its support, i.e., the number of transactions it covers, decreases. In other terms, given an itemset, the supports of its supersets are lower or equal to its support. For more than 20 years, a large number of algorithms have been proposed to mine frequent patterns as efficiently as possible [1]. In the Big Data era, proposing efficient algorithms that handle huge volumes of transactions still remains an important challenge due to the memory space requirements while mining all frequent patterns. To tackle this issue, recent approaches

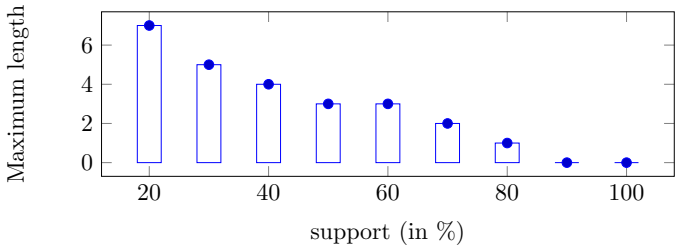

Figure 1: Maximum length of frequent itemsets in WebDocs dataset

have been made to work in distributed environments and the major idea is to distinguish two mining phases: a global phase and a local one. In such schemes, the first global mining phase often relies on MapReduce [6] to find among the frequent patterns the ones whose calculation requires a huge amount of data that does not fit in memory; then, the local mining phase mines on single nodes all the supersets of the patterns obtained at the global phase. Obviously, the idea is that such supersets can be extracted using a part of data that can fit in the memory of a single machine. Intuitively, the first phase guarantees the possibility of working on huge datasets while the second phase preserves a reasonable execution time. Unfortunately, existing approaches are all difficult to be fully extensible, *i.e.* mining becomes intractable as soon as the number of transactions is too large or the minimum frequency threshold is too low.

For the efficiency of two-phase frequent mining approaches, a major difficulty consists in determining the balance between the global and the local mining phases. Indeed, if an approach relies too heavily on local mining, it will be limited to large minimum frequency thresholds only, for which the amount of candidate patterns and/or the projected databases fit in memory. For instance, Parallel FPF algorithm [8] with distributed projected databases cannot deal with very low minimum thresholds when projected databases do not fit in local memory of a machine. Conversely, if an approach relies too much on global mining, it will be less efficient since the cost of communications is high. For instance, Parallel Apriori [9] is relatively slow for low thresholds because all patterns are extracted during the global phase. In BigFIM [15], a parameter k must be set by the users: it represents the minimum length below which the itemsets are mined globally while itemsets that are larger than k are locally mined since they cover a smaller set of transactions that is supposed to fit in memory. However, a practical problem is that such a length is difficult to determine as it depends on datasets and on available memory. To illustrate the issue raised by the setting of the threshold k, Figure 1 plots the maximum lengths of frequent itemsets with the WebDocs dataset (see Section 5 for details) when the minimum frequency threshold varies. In [15], it is suggested to use a global phase for itemsets of size $k = 3$, assuming that, for larger itemsets, the conditional databases will fit in memory. However, from Figure 1, with respect to the used dataset, it is easy to see that 3 is not a sufficiently high threshold since there is at least one itemset of size 4 that covers more than 40% of transactions. Moreover, two patterns of the same size may have very different frequencies and this point is not taken into account in BigFIM.

We introduced in [7] MapFIM (Memory aware parallelized Frequent Itemset Mining) algorithm which is, to the best of our knowledge, the first algorithm extensible with respect to the number of transactions. The advantage of this extensibility is that, it is possible to process large volumes of data (although the addition of machines does not necessarily improve run-time performance as it is the case with scalability). The key idea is to introduce a maximum frequency threshold β above which frequency counting for an itemset is distributed on several machines.

We proved that there exists at least one setting of β for which the algorithm is extensible under the conditions that the FIM algorithm used locally takes a memory space bounded with respect to the size of a projected database and that the set of items holds in memory. We showed how to determine this parameter in practice. Indeed, the higher this threshold, the faster the mining (because more patterns are mined locally). Performance evaluation showed the extensibility and the efficiency of MapFIM compared to the best state-of-the-art algorithms. Nevertheless, in MapFIM, β parameter is estimated on the average of transaction lengths. This may induce a rough estimation of β parameter in some extreme database cases, whereas β is the criterion used to switch from the global to the local mining phase, and finding an accurate evaluation of β is fundamental for efficiency.

In this paper we introduce MapFIM+, an extended version of MapFIM. MapFIM+ improves our previous approach by using an efficient MapReduce routine to generate candidate sets, which makes it more efficient than MapFIM by reducing communication and disks I/O costs; this is all the more true in the case of huge candidate sets. Moreover, MapFIM+ is based on an exact evaluation of prefix-projected database sizes which makes MapFIM+ insensitive to an estimation error of MapFIM's β parameter. However, to be able to adapt MapFIM+ to the amount of memory available on processing nodes for local mining phase, we introduce a simplified parameter called γ that is used as a prefix-projected database size threshold for local mining. This parameter is fixed only using the amount of memory available on processing nodes.

Contributions.

- We present a transaction-extensible algorithm MapFIM+ for mining frequent itemsets, i.e., it manages to mine all frequent itemsets whatever the number of transactions.
- MapFIM+ is an extension of MapFIM, that relies on a single parameter γ, depending only on the amount of memory available on processing nodes. The parameter γ allows to determine when the frequent supersets of a frequent itemset can be mined locally.
- We prove that our algorithm is correct and complete under the condition that the local mining algorithm is prefix-complete. We also prove that our algorithm is transaction-extensible.
- We propose a method for automatically calibrating γ.
- We conduct experiments, on WebDocs dataset and on artificially generated datasets, allowing to compare MapFIM+ first with MapFIM and then, with the best approaches for itemset mining using Hadoop MapReduce framework (BigFIM and PFP). We also illustrate the transaction-extensibility of MapFIM+, showing that it can deal with databases that BigFIM and PFP can not handle.

The rest of the paper is organized as follows. Section 2 formulates the problem of frequent itemset mining in an extensible way in order to deal with a huge volume of transactions. Section 3 studies existing approaches in literature in terms of extensibility. In Section 4, we present how our improved algorithm MapFIM+ works and in particular, we detail the two phases (i.e., global and local mining processes) and the switch between them. In Section 5, we empirically compare the performance of MapFIM+ to MapFIM [7] and also against the state-of-the-art methods by comparing execution times and memory consumption. Section 6 concludes this paper.

transaction	items	transaction	items
t_1	a	t_6	a, d
t_2	a, b	t_7	b, c
t_3	a, b, c	t_8	c, d
t_4	a, b, c, d	t_9	c, e
t_5	a, c	t_{10}	f

Table 1: Original dataset

2 Problem Formulation

2.1 Frequent Itemset Mining problem

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a set of n literals called *items*. An itemset (or a pattern) is a subset of $\mathcal{I}$. The language of itemsets corresponds to $2^{\mathcal{I}}$. A transactional database $\mathcal{D} = \{t_1, t_2, \ldots, t_m\}$ is a multi-set of itemsets of $2^{\mathcal{I}}$. Each itemset t_i, usually called a *transaction*, is a database entry. For instance, Table 1 gives a transactional database with 10 transactions t_i described by 6 items $\mathcal{I} = \{a, b, c, d, e, f\}$.

Pattern discovery takes advantage of interestingness measures to evaluate the relevancy of an itemset. The *frequency* of an itemset X in the transactional database $\mathcal{D}$ is the number of transactions covered by X [2]: $freq(X, \mathcal{D}) = |\{t \in \mathcal{D} : X \subseteq t\}|$ (or $freq(X)$ for sake of brevity). Then, the *support* of X is its proportion of covered transactions in $\mathcal{D}$: $supp(X, \mathcal{D}) = freq(X, \mathcal{D})/|\mathcal{D}|$. An itemset is said to be *frequent* when its support exceeds a user-specified minimum threshold α. **Given a set of items $\mathcal{I}$, a transactional database $\mathcal{D}$ and a minimum support threshold, frequent itemset mining (FIM) aims at enumerating all frequent itemsets.**

In our approach, in order to distribute the computation of frequent itemsets, we use the usual notion of *projected database*. More precisely, given an arbitrary total order over the set of all items, the projected database of an itemset is defined as follows:

Definition 1 (Projected database)**.** *Given an arbitrary total order $<_{\mathcal{I}}$ over the set $\mathcal{I}$ of all items and a database $\mathcal{D}$, let X be an itemset. The projected database of X, denoted $\mathcal{D}_X$, is defined by: $\mathcal{D}_X = \{\sigma_{>X}(Y) : Y \in \mathcal{D}, X \subset Y\}$ where $\sigma_{>X}(Y) = \{i \in Y : (\forall j \in X)(j <_{\mathcal{I}} i)\}$. Moreover, the size of $\mathcal{D}_X$, denoted $\|\mathcal{D}_X\|$, is defined by $\|\mathcal{D}_X\| = \sum_{Y \in \mathcal{D}_X} |Y|$.*

For instance, considering the database $\mathcal{D}$ presented Table 1 and the total order $a <_{\mathcal{I}} b <_{\mathcal{I}} \cdots <_{\mathcal{I}} f$, the projected database of itemset ab is $\mathcal{D}_{ab} = \{c, cd\}$ and we have $\|\mathcal{D}_{ab}\| = 1 + 2 = 3$.

2.2 MapReduce programming model

MapReduce is a simple yet powerful programming model initialized by Google [6] for implementing distributed applications without having extensive prior knowledge of issues related to data redistribution, task allocation or fault tolerance in large scale distributed systems.

The core functioning of MapReduce is based on two functions, **map** and **reduce**, that developers are supposed to provide to the framework. These two functions should have the following signatures:

Map: $(k_1, v_1) \longrightarrow list(k_2, v_2)$**,**
Reduce: $(k_2, list(v_2)) \longrightarrow list(k_3, v_3)$**.**

The **map** function has two input parameters, a key k_1 and an associated value v_1, and outputs a list of intermediate key/value pairs (k_2, v_2). This list is partitioned by the MapReduce framework depending on the values of k_2, with the constraint that all elements with the same value of k_2 belong to the same group. The **reduce** function has two parameters as inputs: an intermediate key k_2 and a list of intermediate values $list(v_2)$ associated with k_2. It applies the user defined merge logic on $list(v_2)$ and outputs a list of values $list(k_3, v_3)$.

MapReduce excels in the treatment of data parallel applications, where computation can be decomposed into many independent tasks, involving large input data. However MapReduce's performance may degrade in the case of dependent tasks or in the presence of skewed data due to the fact that, in Map phase, all the emitted key-value pairs (k_2, v_2) corresponding to the same key k_2 are sent to the same reducer. This may induce a load imbalance among processing nodes and also can lead to task failures whenever the list of values corresponding to a specific key k_2 cannot fit in processing nodes available memory [3, 4]. For scalability, MapReduce algorithm's design must avoid load imbalance among processing nodes while reducing disks I/O and communication costs during all stages of MapReduce jobs computation.

In this paper, our approach is based on Hadoop, the industrial standard open source implementation of MapReduce as well on its distributed file system HDFS (Hadoop Distributed File System) designed to store very large files with streaming data access patterns.

2.3 The challenge of extensibility

Guaranteeing the correct execution of a method whatever the volume of input data is a classical challenge in MapReduce through the notion of scalability. Scalability refers to the capacity of a method to perform similarly even if there is a change in the order of magnitude of the data volume, in particular by adding new machines (as mappers or reducers). We introduce the notion of *extensibility*, which refers to the capacity of a method to deal with an increase in the data volume but without performance guarantees.

More precisely, our goal is to efficiently process transaction databases whatever the number of transactions while the set of all distinct items remains unchanged. This situation covers many practical use cases. For instance, in a supermarket, the set of products is relatively stable while new transactions will be added continuously. We formalize the notion of extensibility with respect to the number of transactions as follows:

Definition 2 (Transaction-extensible)**.** *Given a set of items $\mathcal{I}$, a FIM method is said to be transaction-extensible iff it manages to mine all frequent itemsets whatever the number of transactions in $\mathcal{D} = \{t_1, \ldots, t_m\}$ (where $t_i \subseteq \mathcal{I}$) and the minimum support threshold α.*

This definition is particularly interesting for a pattern discovery task. Indeed, the transaction-extensible property guarantees that for a given set of items $\mathcal{I}$, the method will always be able to mine all the frequent itemsets whatever the changes of the number of transactions in $\mathcal{D}$ and of the minimum frequent threshold α. In this paper, we aim at proposing the first transaction-extensible FIM method. This goal is clearly a challenge in terms of controlling the amount of memory required by frequent itemset mining.

3 Related Work

Due to the explosive growth of data, many parallel implementations of frequent pattern mining (FPM) algorithms have been proposed in the literature, mainly on mining frequent itemsets [8–10, 16–18, 21], but also to mine frequent sequences [5, 14]. In this section, we only consider

related work involving the parallelization of FPM algorithms via MapReduce and review their important shortcomings.

A first category of approaches includes approaches that are specific parallelizations of existing FPM algorithms, for example, different adaptations of Apriori on MapReduce [9,10]. These adaptations of Apriori are not *transaction-extensible* since they assume that at each level, the set of candidate itemsets can be stored in the main memory of the worker nodes (mappers or reducers). We show in Section 4.3 how this limitation can be overcome by using HDFS to store the set of candidates. Different implementations of FP-Growth on MapReduce [8,21] distribute the conditional databases of the frequent items to the mappers. However, these proposals do not guarantee that the conditional databases can be stored among worker nodes, and therefore, these parallelizations of FP-Growth are also not *transaction-extensible*. More recently, Makanju et al. [13] propose to use Parent-Child MapReduce (a new feature of IBM Platform Symphony) to overcome the limitations of the previous implementations of FP-Growth and show that their method provides significant speed-ups over Parallel FP-Growth [8]. However, their method requires to predict the processing loads of a FP-Tree which is a particularly difficult challenge.

A second category of approaches includes approaches that are independent of a specific FPM algorithm, meaning that after a data preparation and partitioning phase, they can use any existing FPM methods to locally extract patterns. In this category, we can distinguish two sub-categories of approaches as follows.

At a high-level, the methods in the first sub-category carefully partition the original dataset in such a way that each partition can be mined independently and in parallel [14,16,17]. In [18], the authors propose an algorithm that extract frequent itemsets in three phases. In the first phase, their algorithm divides the original dataset $\mathcal{D}$ into a number of non-overlapping partitions $\mathcal{D}_k$ ($k \in [1..K]$). Then, in the second phase, each partition $\mathcal{D}_k$ is mined independently to extract itemsets that are locally frequent, i.e. frequent in $\mathcal{D}_k$. Finally, in the third phase, the sets of locally frequent itemsets are merged, and a scan of the whole dataset is performed to identify the itemsets that are globally frequent, i.e. frequent in $\mathcal{D}$. Note that this third phase is necessary because the sets of locally frequent itemsets may contain false positive, i.e. itemsets that are locally frequent but not globally frequent. In order to overcome this problem and remove the need of the third phase, the algorithms proposed in [14, 16, 17] introduced partition methods such that locally frequent patterns are necessarily globally frequent. However, in order to obtain this property, it is important to note that with these methods, partitions $\mathcal{D}_k$ can overlap and that some frequent patterns can be generated several times. Moreover, all these methods cannot guarantee that all data partitions $\mathcal{D}_k$ will fit in main memory (of the mappers or reducers), which means that they are not *transaction-extensible*.

The approaches in the second sub-category do not initially partition the dataset, but the search space (the pattern language), thereby ensuring that each frequent pattern is only generated once. We can consider that Parallel FP-Growth (PFP) [9] also belongs to this second sub-category of methods. However, because PFP partitions the search space only considering single frequent items, it is less efficient. In order to overcome this type of limitation, Moens et al. [15] propose in BigFIM to use longer frequent itemsets as prefixes for partitioning the search space. In a first and global phase, BigFIM mines the frequent k-itemsets using a MapReduce implementation of Apriori, and then subsets of prefixes of length k are passed to worker nodes in a second phase. These worker nodes use the conditional databases of prefixes to mine frequent patterns that are more specific, assuming that the conditional databases can fit in the main memory of the worker nodes.

Note that the selection of the parameter k for BigFIM can be very difficult in practice. Indeed, if a too low value is chosen for k, BigFIM might not terminate successfully if any

conditional database cannot fit in main memory; on the other hand, if the k value is too high, the first global phase that computes the frequent k-itemsets will be highly time consuming. That is also why we propose our novel approach MapFIM+ that do not require any involvement to fix a parameter such as k, and automatically detect when it is possible to switch from global mining to local mining.

4 MapFIM+: An improved MapReduce approach for Frequent Itemset Mining

4.1 Overview of the approach

The key idea of our proposal is to enumerate using a breadth-first search all itemsets using distributed techniques (global mining phase) until one reaches a point of the search space where all its supersets can be mined on a single machine (local mining phase). This point of the search space is reached as soon as each projected database (plus the amount of memory required to enumerate the itemsets) holds in memory. To do this, for each itemset, we do not only compute its frequency, but also the size of its projected database.

More precisely, in order to evaluate when it is possible to switch to the local mining phase, we introduce a maximum projected database threshold γ and define below the notions of *locally tractable* itemset and *minimally locally tractable* itemset.

Definition 3 (Locally tractable)**.** *Given a database $\mathcal{D}$, a minimum support threshold α and a maximum projected database threshold γ, an itemset X is said to be* locally tractable *if it is frequent and its projected database holds in memory, i.e. $supp(X, \mathcal{D}) \geq \alpha$ and $\|\mathcal{D}_X\| \leq \gamma$. Moreover, an itemset $X = (i_1, \ldots, i_k)$ with $i_1 <_{\mathcal{I}} \ldots <_{\mathcal{I}} i_k$, is said to be* minimally locally tractable *if it is* locally tractable *and $(i_1, \ldots, i_{k-1})$ is not* locally tractable*.*

In the following, we denote $\mathcal{T}^+$ the set of frequent itemsets that are minimally locally tractable*,* e.g. *itemsets X such that $supp(X, \mathcal{D}) \geq \alpha$, $\|\mathcal{D}_X\| \leq \gamma$ (*locally tractable*) and that are minimal among the* locally tractable *itemsets. We also denote $\mathcal{T}^-$ the set of frequent itemsets that are not* locally tractable*,* e.g. *itemsets X such that $supp(X, \mathcal{D}) \geq \alpha$ and $\|\mathcal{D}_X\| > \gamma$. $\mathcal{G}$ is the set of frequent itemsets that are either not* locally tractable *or* minimally locally tractable *($\mathcal{G} = \mathcal{T}^+ \cup \mathcal{T}^-$). Such sets are indexed by k when we want to refer to itemsets of length k.*

For instance, let us consider the database $\mathcal{D}$ presented in Table 1, the minimum support threshold $\alpha = 20\%$ and the maximum projected database threshold $\gamma = 5$. Note that itemset $\{a\}$ is not *locally tractable* as its projected database $\mathcal{D}_a = \{b, bc, bcd, c, d\}$ and $\|\mathcal{D}_a\| = 1 + 2 + 3 + 1 + 1 = 8 > \gamma$. On the other hand, itemset $\{ab\}$ is *locally tractable* as $freq(ab, \mathcal{D}) = 3 \geq \alpha.|\mathcal{D}| = 2$ (ab is frequent), $\mathcal{D}_{ab} = \{c, cd\}$ and $\|\mathcal{D}_{ab}\| = 1 + 2 = 3 < \gamma$ (the projected database of ab holds in memory). Moreover, itemset $\{ab\}$ is *minimally locally tractable* since itemset $\{a\}$ is not *locally tractable*.

Given a transactional database $\mathcal{D}$, a minimum support threshold α and a maximum projected data threshold γ, as shown in Algorithm 1, MapFIM+ enumerates all frequent itemsets by using three main phases:

1. **Initialization and database compression:** This phase initializes the process by compressing the original database $\mathcal{D}$ based on frequent 1-itemsets (see line 3 of Algorithm 1). Considering the database $\mathcal{D}$ presented in Table 1 and $\alpha = 20\%$, the compressed version of

$\mathcal{D}$, denoted $\mathcal{D}'$, is shown in Table 2. $\mathcal{D}$ is first compressed by removing non-frequent items e and f. Then, transactions t_1 and t_{10} in $\mathcal{D}$ are removed because they can not contain an itemset of size 2 (or greater). Next, Algorithm 1 computes the set $\mathcal{T}_1^-$ of 1-itemsets in $\mathcal{G}_1$ that are not locally tractable (see line 4) and the set $\mathcal{T}_1^+$ of 1-itemsets that are *minimally locally tractable* (see line 5). In our example, considering $\alpha = 20\%$ and $\gamma = 5$, we obtain $\mathcal{G}_1 = \{a, b, c, d\}$. Note that itemset $\{a\}$ is not *locally tractable* as its projected database $\mathcal{D}'_a$ consists of 5 transactions (see Table 3) and $\|\mathcal{D}_a\| = 8 > \gamma$. Conversely, itemsets $\{b\}$, $\{c\}$ and $\{d\}$ are *locally tractable* since $\|D'_b\| = 4 < 5$, $\|D'_c\| = 2 < 5$ and $\|D'_d\| = 0 < 5$. Therefore, at the end of this phase, we have $\mathcal{T}_1^- = \{a\}$ and $\mathcal{T}_1^+ = \{b, c, d\}$.

2. **Global mining:** This phase mines all potentially frequent itemsets that are not *locally tractable* using Apriori algorithm. At each iteration k, Algorithm 1 first generates the set of candidate k-itemsets $\mathcal{C}_k$ by joining $\mathcal{T}_{k-1}^-$ and $\mathcal{G}_{k-1}$ (see line 9). Note that we do not join $\mathcal{G}_{k-1}$ with $\mathcal{G}_{k-1}$ because we do not want to generate candidates that are *locally tractable*, except candidates that are potentially *minimally locally tractable*. For instance, at step $k = 2$, three candidates of size 2 will be generated from itemsets in $\mathcal{T}_1^- = \{a\}$ and $\mathcal{G}_1 = \{a, b, c, d\}$, *e.g.* $\mathcal{C}_2 = \{ab, ac, ad\}$. Indeed, these candidate 2-itemsets are potentially not *locally tractable*.

   After the generation of the set of candidate k-itemsets $\mathcal{C}_k$, Algorithm 1 evaluates their frequency and the size of their projected database (see line 11). Using these measures, it computes the set $\mathcal{G}_k$ of frequent k-itemsets (mined during the global phase). Then, it identifies from $\mathcal{G}_k$ the set of frequent k-itemsets that are not *locally tractable*, *e.g.* the set $\mathcal{T}_k^-$ (see line 13). Finally, it computes the set $\mathcal{T}_k^+$ of frequent k-itemsets that are *minimally locally tractable*. We will demonstrate in Section 4.5 that all itemsets in $\mathcal{G}_k \setminus \mathcal{T}_k^-$ are necessarily *minimally locally tractable*. In our example, all candidate 2-itemsets in $\mathcal{C}_2 = \{ab, ac, ad\}$ are frequent. Indeed, we have: $freq(ab, \mathcal{D}') = 3 \geq \alpha.|\mathcal{D}| = 2$, $freq(ac, \mathcal{D}') = 3 \geq 2$ and $freq(ad, \mathcal{D}') = 2 \geq 2$. Moreover, because the size of their projected databases ($\|\mathcal{D}'_{ab}\| = 3$, $\|\mathcal{D}'_{ac}\| = 1$ and $\|\mathcal{D}'_{ad}\| = 0$) is lower than $\gamma = 5$, $\mathcal{T}_2^-$ is empty and $\mathcal{T}_2^+ = \{ab, ac, ad\}$. Finally, because $\mathcal{T}_2^- = \emptyset$, MapFIM+ moves to the *local mining* phase (see the test at line 7).

3. **Local mining:** This phase mines the itemsets from frequent itemsets that are *minimally locally tractable* (see line 17). In our running example, the frequency of the prefix-based supersets generated from $\mathcal{T}^+ = \mathcal{T}_1^+ \cup \mathcal{T}_2^+ = \{b, c, d, ab, ac, ad\}$ will be evaluated during this phase. Each prefix is considered individually by using a projected database as given in Table 3. More precisely, frequent pattern abc will be generated from the prefix ab, frequent pattern bc from the prefix b, frequent pattern cd from c, and we will obtain $\mathcal{L} = \{abc, bc, cd\}$. Finally, Algorithm 1 will return the set of all frequent patterns $\mathcal{G}_1 \cup \mathcal{G}_2 \cup \mathcal{L} = \{a, b, c, d\} \cup \{ab, ac, ad\} \cup \{abc, bc, cd\}$.

In the following, Sections 4.2, 4.3, and 4.4 detail how MapReduce is used to implement efficiently the three main phases of MapFIM+. Then, Section 4.5 demonstrates the completeness and extensibility of MapFIM+ with respect to the number of transactions.

4.2 Database compression and initialization

In this phase, we have first to compute the set of frequent 1-items, *e.g.* the set of items in $\mathcal{G}_1$. Using MapReduce, this goal can be achieved by adapting the *Word Count* routine [6]. Each item is considered as a word and we get the support of every item by MapReduce word

Transaction	items
t_2	a, b
t_3	a, b, c
t_4	a, b, c, d
t_5	a, c
t_6	a, d
t_7	b, c
t_8	c, d

Table 2: Compressed database $\mathcal{D}'$

itemset	Projected databases
a	$\mathcal{D}'_a = \{b, bc, bcd, c, d\}$
b	$\mathcal{D}'_b = \{c, cd, c\}$
c	$\mathcal{D}'_c = \{d, d\}$
d	$\mathcal{D}'_d = \emptyset$
ab	$\mathcal{D}'_{ab} = \{c, cd\}$
ac	$\mathcal{D}'_{ac} = \{d\}$
ad	$\mathcal{D}'_{ad} = \emptyset$
cd	$\mathcal{D}'_{cd} = \emptyset$

Table 3: Projected databases

Algorithm 1: MapFIM+: chaining of MapReduce jobs

1 **Function** `Main(`*Float* α*,Float* γ`):`
2 // Initialization and database compression phase;
3 Computation of $\mathcal{G}_1$ and generation of compressed database $\mathcal{D}'$ from $\mathcal{D}$;
4 $\mathcal{T}_1^- \leftarrow \{X \in \mathcal{G}_1 : \|D'_X\| > \gamma\}$; $\mathcal{T}_1^+ \leftarrow \mathcal{G}_1 \setminus \mathcal{T}_1^-$ and $k \leftarrow 2$;
5 // Global Mining phase;
6 **while** $|\mathcal{T}_{k-1}^-| > 0$ **do**
7 // Using GenMap/GenReduce job;
8 Generation of $\mathcal{C}_k$ from $\mathcal{T}_{k-1}^-$ and $\mathcal{G}_{k-1}$;
9 // Using EvalMap/EvalReduce job;
10 Evaluation of $freq(X, \mathcal{D}')$ and $\|D'_X\|$ for all candidates $X \in \mathcal{C}_k$;
11 $\mathcal{G}_k \leftarrow \{X \in \mathcal{C}_k : freq(X, D') \geq \alpha.|\mathcal{D}|\}$;
12 $\mathcal{T}_k^- \leftarrow \{X \in \mathcal{G}_k : \|D'_X\| > \gamma\}$;
13 $\mathcal{T}_k^+ \leftarrow \mathcal{G}_k \setminus \mathcal{T}_k^-$;
14 $k \leftarrow k + 1$;
15 // Local Mining phase using LocalMap/LocalReduce job;
16 $\mathcal{T}^+ = \bigcup_{i=1}^{k-1} \mathcal{T}_i^+$;
17 Computation of $\mathcal{L} \leftarrow \bigcup_{X \in \mathcal{T}^+} \{Y : X \subseteq Y \wedge freq(Y, \mathcal{D}') \geq \alpha.|\mathcal{D}|\}$;
18 **return** $(\bigcup_{i=1}^{k-1} \mathcal{G}_i\}) \cup \mathcal{L}$;

counting. Then, the compressed database $\mathcal{D}'$ is generated and stored in HDFS. This procedure is solved by a simple Map function, where each mapper reads a block of data, removes items which are not in $\mathcal{G}_1$, and finally emits transactions with at least two frequent items.

Finally, we have to compute the size of the projected databases of all frequent items to determine if they are *locally tractable* or not (see line 4 of Algorithm 1). This goal can also be achieved using MapReduce. In the Map function, for each transaction $t \in \mathcal{D}'$, a pair (i, s) is emitted where i is an item in t and s is the length of t minus the position of i in t. In the Reduce function, for each pair (i, L) received, we just have to sum the values in L to obtain the size of the projected database $\mathcal{D}'_i$ of item i. For example, for transactions t_2, t_3, t_4 and t_7, the mappers will emit the pairs $(b, 0)$, $(b, 1)$, $(b, 2)$ and $(b, 1)$ and a reducer will compute the size of the projected database $\mathcal{D}'_b$ of item b as $0 + 1 + 2 + 1 = 4$. At the same time, using the γ parameter and the size of the projected database of all frequent items, the sets $\mathcal{T}_1^-$ and $\mathcal{T}_1^+$ can be easily constructed.

4.3 Global mining based on Apriori

This phase is similar to the parallel implementation of Apriori algorithm [9]. The key difference is mainly on the way candidates are generated (see Section 4.3.1). Moreover, during the evaluation of the supports of the candidates (see Section 4.3.2), we also evaluate the size of their projected databases, in order to detect whether they are *minimally locally tractable* or not.

4.3.1 Candidate generation step

In Apriori algorithm and its parallel implementation [9], the set $\mathcal{C}_{k+1}$ of candidate $(k+1)$-itemsets is generated by the join $\mathcal{L}_k \bowtie \mathcal{L}_k$ at each iteration, where $\mathcal{L}_k$ denotes the set of all frequent k-itemsets[1].

In our case, a candidate $(k+1)$-itemset is obtained by the join of a frequent but not *locally tractable* k-itemset, i.e. an itemset in $\mathcal{T}_k^-$, with a frequent k-itemset that is *locally tractable* or not, i.e. an itemset in $\mathcal{G}_k$. During the candidate evaluation step (see Section 4.3.2), all candidate itemsets that are frequent are emitted and stored with a flag in $\{+,-\}$ to underline whether they are *locally tractable* ($flag = +$) or not ($flag = -$). Therefore, at iteration $(k+1)$ of the candidate generation step, we join only frequent k-itemset with a negative flag ($flag = -$) with frequent k-itemset (whatever their flag).

We now detail how we implement the candidate generation step using a MapReduce job (see Algorithm 2). In the Map function (see $GenMap$ in Algorithm 2), for each pair $(X, flag) \in value$, where $X = (i_1, \ldots, i_{k-1}, i_k)$ is a frequent k-itemset and $flag \in \{+,-\}$ indicates whether X is *locally tractable* or not, we emit a pair $(prefix, (i_k, flag))$ where $prefix = (i_1, \ldots, i_{k-1})$ is the prefix of X of length $k-1$.

In the Reduce function (see $GenReduce$ in Algorithm 2), we combine candidate k-itemsets with the same prefix $P = (i_1, \ldots, i_{k-1})$. Given two pairs $(i, flag_i) \in values$ and $(j, flag_j) \in values$, we join the k-itemset $X = (i_1, \ldots, i_{k-1}, i)$ with the k-itemset $X' = (i_1, \ldots, i_{k-1}, j)$ if and only if:

- $flag_i = -$ in order to check whether X is not *locally tractable* (see line 14 of Algorithm 2), i.e. $X \in \mathcal{T}_k^-$ and
- $i < j$ in order to generate each candidate once (see line 16 of Algorithm 2).

If the two conditions are fullfilled, the reducer emits a new candidate $(k+1)$-itemset $Y = X \bowtie X' = (i_1, \ldots, i_{k-1}, i, j)$ (see line 17 of Algorithm 2). In Section 4.5, we prove that combining only frequent k-itemsets $X \in \mathcal{T}_k^-$ with frequent k-itemsets $X' \in \mathcal{G}_k$, we are complete w.r.t. $\mathcal{T}^+ \cup \mathcal{T}^-$.

4.3.2 Candidate evaluation

The candidate evaluation step consists in computing both the frequency and the size of its projected database for each candidate in $\mathcal{C}_k$. This is achieved by a MapReduce job, described in Algorithm 3.

In the Map function (see $EvalMap$ in Algorithm 3), for each transaction $t \in \mathcal{D}'$ and for each candidate $X \in \mathcal{C}_k$, if X is a subset of t (see line 8 in Algorithm 3), a pair composed of the value 1 (thus counting the presence of X in t) and the size of the projection of t in $\mathcal{D}'_X$ is emitted

[1] In our work, in order to generate each candidate once, we use a prefix-based join operation. More precisely, given two sets of frequent k-itemsets $\mathcal{L}_k$ and $\mathcal{L}'_k$, the join of $\mathcal{L}_k$ and $\mathcal{L}'_k$ is defined by: $\mathcal{L}_k \bowtie \mathcal{L}'_k = \{(i_1, \ldots, i_k, i_{k+1}) \mid (i_1, \ldots, i_{k-1}, i_k) \in \mathcal{L}_k \wedge (i_1, \ldots, i_{k-1}, i_{k+1}) \in \mathcal{L}'_k \wedge i_1 < \cdots < i_k < i_{k+1}\}$.

Algorithm 2: MapFIM+: Map and Reduce functions to generate candidate itemsets

```
1  Function GenMap(String key, String value):
2      // key: input name;
3      // value: a set of pairs (X, flag) where X ∈ G_k and flag ∈ {+, −};
4      foreach (X, flag) ∈ value do
5          Let X = (i_1, ..., i_{k−1}, i_k) ;
6          prefix ← (i_1, ..., i_{k−1}) ;
7          Emit(prefix, (i_k,flag));
8  Function GenReduce(String key, Iterator values):
9      // key: a prefix P = (i_1, ..., i_{k−1});
10     // values: a list of pairs (i, flag) where i ∈ I and flag ∈ {+, −};
11     foreach (i, flag_i) ∈ values do
12         Let X = (i_1, ..., i_{k−1}, i);
13         // Test if X ∈ T_k^−;
14         if (flag_i == −) then
15             foreach (j, flag_j) ∈ values do
16                 if (i < j) then
17                     Y ← (i_1, ..., i_{k−1}, i, j) // Y belongs to C_{k+1};
18                     Emit(null, Y) ;
```

(see line 11 in Algorithm 3). More precisely, in order to compute the size of the projection of t in $\mathcal{D}'_X$, we first identify the maximal item i_{max} of X w.r.t. $<_{\mathcal{I}}$ (see line 9 in Algorithm 3). Then, we count the number of items j in t greater than i_{max} w.r.t. $<_{\mathcal{I}}$. Note that if the set of candidats $\mathcal{C}_k$ is too large to fit in memory of Mappers, then $\mathcal{C}_k$ is partitioned into blocks $\mathcal{C}_{Block}$ and Mappers process candidates block by block (see line 5 in Algorithm 3). Finally, we can point out that this Map phase achieves a good load balance because the compressed database $\mathcal{D}'$ is distributed equally among mappers and all mappers handle the same candidate set (reading the same number of candidate blocks).

In the Reduce function (see *EvalReduce* in Algorithm 3), each *key* is a candidate itemset $X \in \mathcal{C}_k$, and *value* is a list L including for each transaction $t \in \mathcal{D}'$ covered by X a pair $(freq, size)$ where $freq = 1$ and $size$ is the size of the projection of t in $\mathcal{D}'_X$. Thus, in order to compute the frequency of X and the size of $\mathcal{D}'_X$, a Reducer has just to sum the first and second components of the pairs in L (see lines 18 and 19). Then, if X is frequent in $\mathcal{D}$ w.r.t. the minimum support threshold α, i.e. $freq(X, \mathcal{D}') \geq \alpha \times |\mathcal{D}'|$ (see line 20 in Algorithm 3), a Reducer tests whether X is *locally tractable* or not (see line 21 in Algorithm 3). Finally, if X is *locally tractable*, we emit a pair $(X, +)$; otherwise, we emit a pair $(X, -)$ (see line 22 and 24 in Algorithm 3).

4.4 Local mining of frequent itemsets

As described in the previous sections, the two-phase mining strategy guarantees the efficiency of MapFIM+. Indeed, when each projected-database stemming from a prefix is sufficiently small to be handled by a single node in the cluster (i.e., $\|D'_X\| \leq \gamma$), MapFIM+ switches to the local mining phase. After presenting the local mining method, we will show how to configure γ

Algorithm 3: MapFIM+: Map and Reduce functions to evaluate candidates itemsets

```
1  Function EvalMap(String key, String value):
2      // key: input name;
3      // value: a subset of transactions in D';
4      while there are unsolved candidates do
5          C_Block ← A block of C_k in HDFS;
6          foreach transaction t ∈ value do
7              foreach itemset X ∈ C_Block do
8                  if X ⊆ t then
9                      i_max ← max_{<_I}(X) // Last item of X;
10                     size ← |{j ∈ t : i_max <_I j}|;
11                     Emit(X, (1, size));
12 Function EvalReduce(String key, Iterator values):
13     // key: a candidate X ∈ C_k;
14     // values: a list of (frequency, projectedTransactionSize);
15     totalFrequency ← 0;
16     totalProjectedData ← 0;
17     foreach (frequency, projectedData) ∈ values do
18         totalFrequency ← totalFrequency + frequency;
19         totalProjectedData ← totalProjectedData + projectedTransactionSize;
20     if totalFrequency ≥ α * |D| then
21         if totalProjectedData ≤ γ then
22             Emit(null, (X, +)) // X belongs to T_k^+ ;
23         else
24             Emit(null,(X, −)) // X belongs to T_k^− ;
```

parameter to ensure that this method has sufficient memory space for D'_X processing.

Method In the local mining phase, the frequent itemset enumeration is completed by using a traditional efficient algorithm (for instance, Eclat [20] or LCM [19]) that fits the memory constraints required by single nodes. More precisely, this algorithm has to enumerate all the itemsets corresponding to a given prefix X in a linear memory space with respect to the size of D'_X. Level-wise algorithms will therefore not be adapted since it is difficult to limit themselves to a given prefix and the amount of memory required is very variable. Similarly, approaches based on FP-trees do not guarantee a bounded amount of memory for tree storage. However vertical database layout based approaches such as Eclat or LCM fit well the requirement of bounded memory usage.

Algorithm 4 details this step, which is still MapReduce driven. Local memory-fitted projected-databases are dispatched to each node (as Reducers) that allow to run the selected local FIM algorithm. Due to the difference in size among projected databases, the local mining could lead to a load imbalance among reducers. In [15], the authors of BigFIM algorithm have experimented different strategies to assign the prefixes and it is shown that a random method can achieve a good workload balancing.

Algorithm 4: MapFIM: Local Mining

```
1  Function LocalMap(String key, String value):
2      // key: input name;
3      // value: a subset of transactions in D ;
4      while there are unsolved locally tractable itemsets do
5          T+_Block ← A block of T+ in HDFS;
6          foreach itemset X ∈ T+_Block do
7              i_max ← max_<I(X) // The last item in X;
8              foreach transaction t ∈ value that contains X do
9                  t' ← {j ∈ t : i_max <I j};
10                 if t' ≠ ∅ then
11                     Emit(X, t');
12 Function LocalReduce(String key, Iterator values):
13     // key: an itemset X;
14     // values: the projected database D_X of X;
15     Create an empty file f_in in local disk;
16     Save values into f_in;
17     Run a local FIM program with input=f_in, output=f_out, support=α * |values|/|D|;
18     foreach frequent itemset X' ∈ f_out do
19         X'' = X ∪ X';
20         Emit(null,X'') // X'' belongs to L;
```

In the Map phase (lines 1-11 in Algorithm 4), we consider frequent itemsets $X \in \mathcal{T}^+$ as prefixes and construct their projected databases. For each $X \in \mathcal{T}^+$, i_{max} denotes the maximal item in X w.r.t $<_{\mathcal{I}}$. The projected-database $\mathcal{D}'_X$ is built by: (1) pruning every transaction $t \in \mathcal{D}'$ that does not contain X (2) pruning every item $j \leq_{\mathcal{I}} i_{max}$ since these items cannot expand X due to the prefix-based join. As shown in Algorithm 4, each Mapper reads a block of data, then for each $X \in \mathcal{T}^+$, it emits every transaction t' that contains X after pruning unnecessary items (line 11).

In the Reduce phase (lines 12-20 in Algorithm 4), a local FIM algorithm is independently called to enumerate all the frequent itemsets for each projected-database. More precisely, in the Reduce phase, each *key* is a frequent itemset $X \in \mathcal{T}^+$ and each list of *values* contains all transactions of the projected-database of X. They are saved to a local file so that the local FIM algorithm can work on it. For each itemset X' being frequent in the projected-database, the itemset $X'' = X \cup X'$ is frequent in $\mathcal{D}$. Notice that in the case where $\mathcal{T}^+$ is too large to fit in memory of Mappers, we partition this set into several parts and repeat a local mining (via a MapReduce phase) for each part until that all itemsets in $\mathcal{T}^+$ are processed.

Automatic parameterization of MapFIM+ In our algorithm, a good value of γ is important for getting high performance. The higher the value of γ is, the better performance we get in general but more memory is required. Unfortunately the configuration of this threshold is complex for a user. Indeed, the appropriate choice of this parameter requires a good understanding of the approach in order to anticipate the amount of memory to consume at the level of this local mining phase. Without an automatic configuration system, a novice user could

make the extraction impossible (memory saturation or extraction time too long). Even a more seasoned user might not get the best of the approach.

The automatic procedure for parameterizing the minimum size of the projected dataset γ requires an offline calibration phase which is carried out only once before extractions. The first step is to select a local frequent itemset mining algorithm whose memory is proportional to the size of the projected database as explained above. For instance, in our experiments, we use a local program based on Eclat/LCM algorithm [19, 20] which requires a maximum of $f(projectedData)$ of memory, where $f()$ is a linear function. Then, the maximum memory needed by the program is $K \times projectedData$ where K is the amount of memory required per item of $projectedData$ in the worst case during the extraction. The second step is to calibrate this algorithm by applying it to various datasets and then, figuring out the value of K. Once the constant K is known, it is easy to configure the parameter γ (whatever the dataset and the minimum frequency threshold) by applying a proportionality law, taking care to keep enough memory for performing the reducer. More precisely, we propose to set the threshold γ in MapFIM+ as follows:

$$\gamma = \frac{M_{Reduce} - M_{reduce\_task}}{K} \tag{1}$$

where M_{Reduce} is the limit of memory of a Reducer and M_{reduce_task} is the memory required for a reduce task without running the local mining program.

4.5 Completeness and extensibility

Thanks to the complementarity of global and local mining phases, this section demonstrates that MapFIM+ is not only correct and complete, but also transaction-extensible.

Let us first recall that MapFIM+ is composed of two phases: a global phase computes $\mathcal{G}$, the set of frequent itemsets that are either not *locally tractable* or that are *minimally locally tractable*, then a local mining phase computes the frequent itemsets derived from frequent *minimally locally tractable* itemsets. We show that the global phase is complete w.r.t. $\mathcal{G}$ and that the whole algorithm is complete w.r.t. the frequent itemsets, under the conditions of the *prefix-completeness* of the local algorithm.

Proposition 1. *MapFIM+ is* **correct***, i.e., all itemsets returned by the algorithm are frequent and* **complete***, i.e., all frequent itemsets are returned by the algorithm.*

Proof: The algorithm counts the support of each itemset and returns only frequent itemsets, therefore it is correct. The proof of the completeness is decomposed in two steps: proof of the completeness of the global phase w.r.t. $\mathcal{G}$ and the proof of the completeness of MapFIM+ w.r.t. $\mathcal{L}$. In the following, let $X = (i_1, \ldots, i_k)$ be an itemset with $i_1 < i_2 \ldots < i_k$, then X_j denotes the prefix of X, with length j, i.e., $X_j = (i_1, \ldots, i_j)$.

Proof of the completeness of the global phase w.r.t. $\mathcal{G}$. We first prove by recurrence on k that the algorithm is complete w.r.t. $\mathcal{G}$, i.e. the set of frequent *minimally locally tractable* itemsets and of frequent non *locally tractable* itemsets. We recall that for $k > 1$, we have $\mathcal{C}_k = \mathcal{T}^-_{k-1} \bowtie \mathcal{G}_{k-1}$. Therefore, to prove completeness, we have to prove that $\mathcal{G}_k \subseteq \mathcal{C}_k$.

This is true for $k = 1$, since during data preparation, the support of all items are counted and only non frequent items are discarded. Thus, it allows to compute $\mathcal{G}_1$. Now, let us suppose that the algorithm is complete w.r.t. $\mathcal{G}_{k-1}$ and let us show that the algorithm is complete w.r.t. $\mathcal{G}_k$.

Let $X = (i_1, \ldots, i_k)$, with $i_1 < i_2 \ldots < i_k$ be an element of $\mathcal{G}_k$ and let us show that $X \in \mathcal{C}_k$. X is equal to the join of two $k-1$ itemsets, $(i_1, \ldots, i_{k-1})$ and $(i_1, \ldots, i_{k-2}, i_k)$, i.e. $X = (i_1, \ldots, i_{k-1}) \bowtie (i_1, \ldots, i_{k-2}, i_k)$. We have two cases:

- $X \in \mathcal{T}_k^-$: X is not *locally tractable*, *i.e.* $supp(X, \mathcal{D}) \geq \alpha$ and $\|\mathcal{D}_X\| > \gamma$.
  - $(i_1, \ldots, i_{k-1}) \in \mathcal{T}_{k-1}^-$, since it is frequent and it is not *locally tractable* (as a prefix of an itemset that is not *locally tractable*)
  - $(i_1, \ldots, i_{k-2}, i_k)$ is frequent but it is not a prefix of X. It is either not *locally tractable* or *locally tractable*. If it is *locally tractable*, it is *minimally locally tractable*, since its prefix is not *locally tractable*. Therefore it belongs either to $\mathcal{T}_{k-1}^-$ or to $\mathcal{T}_{k-1}^+$.

  Therefore $(i_1, \ldots, i_{k-1}) \in \mathcal{T}_{k-1}^-$ and $(i_1, \ldots, i_{k-2}, i_k) \in \mathcal{T}_{k-1}^- \cup \mathcal{T}_{k-1}^+ = \mathcal{G}_{k-1}$ and X belongs to $\mathcal{T}_{k-1}^- \bowtie \mathcal{G}_{k-1}$
- $X \in \mathcal{T}_k^+$: X is *minimally locally tractable*, *i.e.* $supp(X, \mathcal{D}) \geq \alpha$, $\|\mathcal{D}_X\| \leq \gamma$, and all its prefixes X_j satisfy $\left\|\mathcal{D}_{X_j}\right\| > \gamma$
  - $(i_1, \ldots, i_{k-1})$ and $(i_1, \ldots, i_{k-2})$ are frequent but not *locally tractable* (otherwise X would not be *minimally locally tractable*), i.e. $(i_1, \ldots, i_{k-1}) \in \mathcal{T}_{k-1}^-$ and $(i_1, \ldots, i_{k-2}) \in \mathcal{T}_{k-2}^-$
  - $(i_1, \ldots, i_{k-2}, i_k)$ is either in $\mathcal{T}_{k-1}^-$ or in $\mathcal{T}_{k-1}^+$ (since $(i_1, \ldots, i_{k-2}) \in \mathcal{T}_{k-1}^-$), i.e. $(i_1, \ldots, i_{k-2}, i_k) \in \mathcal{G}_{k-1}$

  As a consequence, X belongs to $\mathcal{T}_{k-1}^- \bowtie \mathcal{G}_{k-1}$.

Proof of the completeness of MapFIM+ w.r.t. $\mathcal{L}$.

Now we can prove the completeness of the algorithm w.r.t. the set $\mathcal{L}$ of frequent itemsets. Let $X = (i_1, \ldots, i_k)$, with $i_1 < i_2 \ldots < i_k$, be a frequent itemset. We have two cases:

- $X \in \mathcal{T}^-$, therefore $X \in \mathcal{G}$ and we have already shown the completeness of the algorithm w.r.t $\mathcal{G}$.
- $X \notin \mathcal{T}^-$: $supp(X, \mathcal{D}) \geq \alpha$ and $\|\mathcal{D}_X\| \leq \gamma$. Let j be the smallest index such that $X_j \in \mathcal{T}^+$. X_j belongs to $\mathcal{G}$ and therefore it has been generated in the global phase. If $j = k$ then $X_j = X$ has been generated at the global phase, otherwise *under the condition that the local mining algorithm is prefix complete*, X is generated in the local phase. More precisely, the frequent itemsets starting by X_j will be mined in the local mining step, from the conditional database with respect to X_j. It is built by considering all transactions in $\mathcal{D}$ containing X_j and removing from these transactions all items i with $i \leq i_j$. Since X is ordered, if X is frequent in $\mathcal{D}$ then $\{i_{j+1}, \ldots, i_k\}$ is frequent in the conditional database w.r.t X_j and will be found during the local mining phase.

The main challenge faced by MapFIM+ is to deal with a very large number of transactions. This is possible because the preparation and the scanning of this transactional database is distributed on several mappers and the set of generated candidates that is potentially huge is stored on the distributed file system. Therefore, in addition to being complete, MapFIM+ is transaction-extensible as introduced by Definition 2:

Proposition 2 (Transaction-extensible). *Assuming the distributed file system has an infinite storage capacity, MapFIM+ is transaction-extensible when the set of items I holds in memory and the local frequent itemset mining method takes space $O(\gamma)$.*

Proof: The first step of data preparation is not a problem as it is similar to MapReduce *WordCount* problem. The second step is also transaction-extensible because the set of frequent items holds in memory as we make the assumption that the set of all items holds in memory. Global mining phase does not raise any problem because all candidates are stored on the distributed file system (which has an infinite storage capacity) and can be partitioned into independent blocks of candidates (see line 5 of Algorithm 3). For local mining phase, the minimally locally tractable itemsets are also considered block by block (see line 5 of Algorithm 4). In the reduce step, the mining algorithm for a prefix takes a memory space proportional to the size of its projected database so there is at least one γ such that each projected database holds in memory.

5 Experiments

The experimental evaluation mainly focuses on performance and transaction-extensibility of the proposed Memory Aware Parallelized Frequent Itemset method. The research questions are as follows:

- **Q1** How MapFIM+ compare to MapFIM, MapFIM+ being an improved version of our previous MapFIM method?
- **Q2** How transaction-extensible is MapFIM+, i.e. does it manage to mine all frequent itemsets whatever the number of transactions in the dataset and the minimum support threshold?
- **Q3** How MapFIM+ compare to the best approaches for itemset mining using Hadoop MapReduce framework, in particular BigFIM and PFP?

Question Q1 is addressed in Section 5.2, whereas questions Q2 and Q3 are both addressed in Section 5.3. All the experiments were performed on a cluster of 16 virtual machines, where each virtual machine possesses 4 vCPUs, 8 GB RAM, and 300 GB HDD space. Each map/reduce task is allowed to use up to 7 GB of RAM. MapFIM+, MapFIM and PFP are experimented on top of Hadoop 2.7.3 while BigFIM is tested on Hadoop 1.2.1.[2]

5.1 Experimental Setup

Data Sets In our experiments, we have chosen WebDocs dataset [11], one of the largest commonly used datasets in Frequent Itemset Mining. It is derived from real-world data and has a size of 1.48 GB. The copy of the dataset used in our experiments is obtained from the Frequent Itemset Mining Implementations Repository at `http://fimi.ua.ac.be/data/`.

We have also generated various synthetic datasets by using the generator from the IBM Almaden Quest research group. Their program can no longer be downloaded and we have used another repository available at `https://github.com/zakimjz/IBMGenerator`. The command used to generate our synthetic datasets is: `./gen lit -ntrans 50000 -tlen L -nitems 100 -npats 1000 -patlen 4 -ascii` where L is the average length of transactions. We varied

[2]In our configuration, there is no real difference of performance between Hadoop 1.2.1 and Hadoop 2.7.3.

parameter L from 20 to 100 to generate 5 different datasets. The characteristics of the datasets are given in Table 4.

Dataset	Avg length	# Items	# Transactions	FileSize (GB)
WebDocs	177	5,267,656	1,692,082	1.5
T20.I100K.D50M	20	100,000	50,000,000	6.0
T40.I100K.D50M	40	100,000	50,000,000	11.9
T60.I100K.D50M	60	100,000	50,000,000	17.8
T80.I100K.D50M	80	100,000	50,000,000	23.7
T100.I100K.D50M	100	100,000	50,000,000	29.6

Table 4: Characteristic of the two used datasets

Setting of MapFIM+ For setting MapFIM+, we first apply the calibrating protocol described in Section 4.4 on the selected local FIM implementation based on Eclat/LCM algorithm [19,20]. With 10 datasets coming from the FIMI repository at `http://fimi.ua.ac.be/data/`, we run the program with a minimum support threshold equal to 0% to report the maximum memory used during one hour by the program. Then we compute $K_i = \frac{projectedData}{max\_memory}$ and report the result in Table 5 (e.g., the value of K for dataset WebDocs is 0.018). From experiments, the value of K_i varies from 0.017 to 0.043, with an average value K of 0.023 and a standard deviation of 0.00785. For estimating γ, we fixed the value of K equals to 0.05. Finally, as the available memory space of the reducer is M_{Reduce} KB and the reduce task requires around M_{reduce_task} KB, we configure MapFIM+ by setting MapFIM+ such that: $\gamma = (M_{Reduce} - M_{reduce\_task})/0.05$.

5.2 Difference between MapFIM+ and MapFIM

In this subsection, we focus on the first question **Q1**: How MapFIM+ compare to MapFIM? MapFIM+ is implemented similarly to MapFIM algorithm, both are in Java 8 for Hadoop 2. Moreover, for the local mining phase of both MapFIM+ and MapFIM, Eclat/LCM program implemented in C++ by Borgelt at `http://www.borgelt.net/eclat.html` was used to mine local projected database.

Dataset	*projectedData*	*max_memory* (KB)	K (KB)
accidents	11,500,870	228,400	0.020
connect	2,904,951	58,160	0.020
kosarak	8,019,015	193,644	0.024
pumsb	3,629,404	63,332	0.017
retail	908,576	25,588	0.028
T40I10D100K	3,960,507	71,988	0.018
T10I4D100K	1,010,228	25,916	0.026
chess	118,252	5,100	0.043
pumsb_star	2,475,947	47,424	0.019
webdocs	299,887,139	5,422,024	0.018

Table 5: Parameter K setting using Borgelt's implementation of Eclat/LCM

Experimental Setup The main difference between MapFIM+ and MapFIM is how each approach estimates the projected datasets, based on the memory allocation for the local mining phase. Therefore, we experiment the two algorithms using different values of the bound of memory available for the local mining program. Both programs are tested using WebDocs dataset with a value of support equals to 5%. We varied the bound of memory for local mining program from 1024 MB to 4096 MB.

For MapFIM+ algorithm, we recall that γ parameter is defined using the value of K equal to 0.05. For MapFIM algorithm, memory control is based on β parameter which is the number of transactions that can be processed locally. This β parameter is more difficult to set because the size of transactions in a projected database varies. A suboptimal solution is to bound the size of the projected database by using the maximum transaction length l: $\|\mathcal{D}'_X\| \leq \beta \times l$. By injecting this approximation into Equation 1, we obtain for the threshold: $\beta = \gamma/l$. Table 6 presents the estimated value of β parameter in MapFIM.

For example, when memory available for the local program is bounded by 1024 MB, MapFIM estimates that local program can only handle projected datasets with at most 24% of the total transactions. With 4096 MB of available memory or more, the performances of MapFIM+ and MapFIM are identical because every projected datasets generated from frequent items can be mined locally. As a consequence, the number of prefix-projected datasets is the same in the two approaches. However, with limited memory, the difference between MapFIM+ and MapFIM is clear.

Experimental Results Figure 2 shows execution time of the two programs (in seconds). As expected, with 4096 MB of available memory for allocation, there is no difference between the two programs in term of performance. However, for lower memory available for allocation, MapFIM+ performs better than MapFIM. Figure 3 shows the number of projected datasets generated by MapFIM+ and MapFIM for different cases of available memory of local processing nodes. By calculating the exact size of each projected dataset, MapFIM+ achieves a good performance compared to MapFIM by generating a smaller number of projected datasets. Not difficult to see, the gap between MapFIM+ and MapFIM shall be more important in the case of huge candidate sets due to the parallel candidate sets generation in MapFIM+, which is performed by a single processing node in MapFIM.

Memory available for local mining program (MB)	Estimated β in MapFIM
1024	24%
2048	47%
3072	71%
4096	94%

Table 6: Estimated value of β parameter in MapFIM

5.3 Comparison to other existing FIM approaches

In this subsection, we focus on the second and third questions:

- **Q2** How transaction-extensible is MapFIM+?
- **Q3** How MapFIM+ compare to the best existing approaches for itemset mining using Hadoop MapReduce framework?

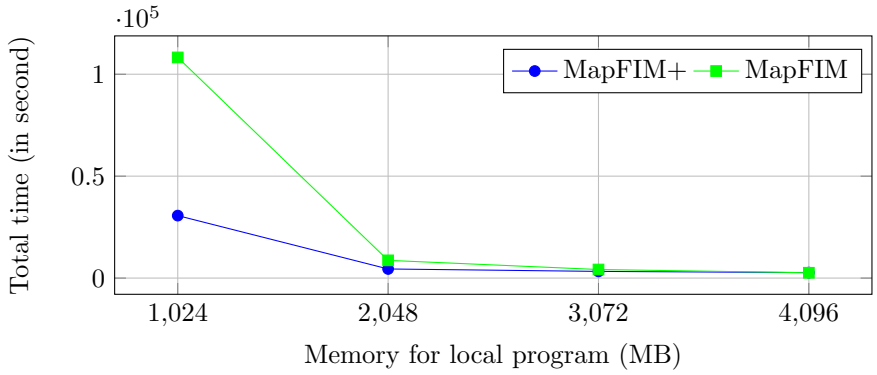

Figure 2: Execution time of MapFIM+ and MapFIM using WebDocs dataset with support=5%

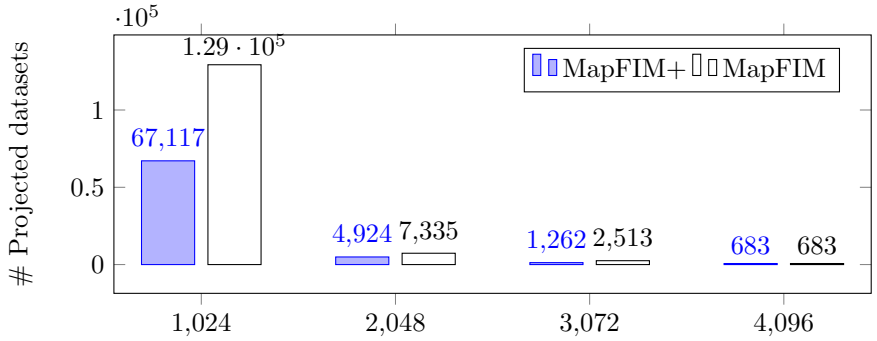

Figure 3: Number of projected datasets using different bounds of local memory (MB)

Experimental Setup Beside MapFIM+, we believe that Parallel FP-Growth (PFP) [8] and BigFIM algorithms [15] are the best approaches for itemset mining using Hadoop MapReduce framework. Thus, we decide to compare the performance of MapFIM+ to BigFIM and PFP algorithms. Moreover, we check that MapFIM+ is transaction-extensible, meaning that it can mine all frequent itemsets whatever the number of transactions in the data set and the minimum support threshold, which is not the case for PFP and BigFIM algorithms. In our experiments, we use PFP implementation available in the library Apache Mahout 0.8 [12] and BigFIM implementation based on Hadoop 1 and provided by the authors at `https://gitlab.com/adrem/BigFIM-sa`.

PFP program was tested with its default parameter and BigFIM program was configured with parameter $k = 3$ as suggested by the authors. Using this configuration, BigFIM uses a parallel Apriori approach to mine all 3-frequent itemsets before switching to local mining phase. It is shown in [15] that with $k = 3$, BigFIM achieves a good performance.

Experimental Results The first experiment consists of testing the real dataset WebDocs. We varied the value of the minimum support threshold from 5% to 15%. This dataset is expected to be hard to mine as it has long frequent itemsets as well asvery frequent itemsets. For example, in this dataset, there exists a frequent 7-itemset that occurs in 20% of the transactions and at least one frequent 3-itemset that appears in more than 60% of transactions.

The results are shown in Table 7. It is surprising that PFP can not solve WebDocs dataset

with values of minimum support threshold between 5% and 15%, showing that PFP is not transaction-extensible. The program requires a huge memory for the Reduce phase for mining projected FP-trees. The results show that MapFIM+ outperforms BigFIM and can solve effectively the dataset for low values of minimum support threshold.

In the second experiment, we compare the performance of approaches with generated synthetic datasets. All the synthetic datasets consists of 100.000 different items and 50.000.000 transactions. However, the average length of transactions varies from 20 to 100. We set the value of minimum support threshold to 0.2%. The result is shown in Table 8. With the average length equal to 20, all three programs can solve the dataset within 20 minutes. However, BigFIM program can not solve other datasets due to memory lack, showing that it is not transaction-extensible. PFP program can solve two further datasets but not the last ones where the average length of transactions is equal to 80 and 100. It is clear that MapFIM+ has a better execution time and is able to solve harder datasets with longer transactions. These results confirm that MapFIM+ is transaction-extensible.

From the papers presenting the different approaches and the implementations of the programs, in our opinion, there are three main reasons that explain why MapFIM+ achieves better performances:

- Our approach generates balanced prefix-projected datasets in an efficient manner while guaranteeing that projected datasets can always be mined locally. This makes local mining of projected datasets more efficient compared to existing approaches.

- Different from BigFIM and PFP, MapFIM+ does not implement the local mining program but is able to use any efficient FIM implementations such as Eclat, so the performance of MapFIM+ could be further improved while applying additional optimizations to local mining process.

- MapFIM+ is transaction-extensible while guaranteeing good balancing properties, among processing nodes, during all computation steps.

Support	MapFIM+	BigFIM	PFP
15	392	4136	Out of Memory
14	401	5583	Out of Memory
13	446	8207	Out of Memory
12	465	12319	Out of Memory
11	514	19748	Out of Memory
10	615	32267	Out of Memory
9	703	Out of Memory	Out of Memory
8	894	Out of Memory	Out of Memory
7	1208	Out of Memory	Out of Memory
6	1798	Out of Memory	Out of Memory
5	2684	Out of Memory	Out of Memory

Table 7: Execution time (in seconds) using WebDocs dataset

Dataset	MapFIM+	BigFIM	PFP
T20.I100K.D50M	788	854	1009
T40.I100K.D50M	3178	Out of Memory	10838
T60.I100K.D50M	7623	Out of Memory	54031
T80.I100K.D50M	15092	Out of Memory	Out of Memory
T100.I100K.D50M	24749	Out of Memory	Out of Memory

Table 8: Execution time (in seconds) using synthetic datasets

6 Conclusion and Future Work

In this paper, we present MapFIM+ an improved version of our previous MapFIM algorithm [7], a MapReduce based two-phase approach to efficiently mine frequent itemsets in very large datasets. In the first global mining phase, MapReduce is used to generate local memory-fitted prefix-projected databases from the input dataset benefiting from the Apriori principle. Then, in a local mining phase, an optimized in-memory mining process is launched to enumerate in parallel all frequent itemsets from each prefix-projected database. Compared to other existing approaches, our algorithm implements a fine-grained method to switch from global phase to the local mining phase. Moreover, we show that our method is transaction-extensible, meaning that given a fixed set of items, it can mine all frequent itemsets whatever the number of transactions and the minimum support threshold. To the best of our knowledge, our algorithm is the first one to guarantee this property.

Our experimental evaluations show that both MapFIM and MapFIM+ outperform the best existing MapReduce based frequent itemset mining approaches. Moreover, MapFIM+ performs better than MapFIM in the case of huge candidate sets by reducing communication and disks I/O costs. We show how to define and set the unique γ parameter of our MapFIM+ algorithm depending only on the available memory for local mining using prefix-projected databases. This point is particularly important, since an optimal value of γ parameter guarantees a high performance level.

Future work will be devoted to make MapFIM+ scalable. This can be achieved by using similar approaches, as those based on randomized key redistributions and introduced in [3,4] for join processing. Indeed, the use of randomized key redistributions prevents the effects of data skew while guaranteeing perfect balancing properties during all the stages of join computation in large scale systems.

Acknowledgement

This work is partly supported by the GIRAFON project funded by *Centre-Val de Loire* region (France).

References

[1] A. C. Aggarwal and J. Han. *Frequent pattern mining.* Springer, 2014.

[2] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *Proc. of VLDB'94*, volume 1215, pages 487–499, 1994.

[3] M. Al Hajj Hassan and M. Bamha. Towards scalability and data skew handling in groupby-joins using mapreduce model. In *Proc. of ICCS'2015*, pages 70–79, 2015.

[4] M. Al Hajj Hassan, M. Bamha, and F. Loulergue. Handling data-skew effects in join operations using mapreduce. In *Proc. of ICCS'2014*, pages 145–158. IEEE, 2014.

[5] K. Beedkar, K. Berberich, R. Gemulla, and I. Miliaraki. Closing the gap: Sequence mining at scale. *ACM Trans. Database Syst.*, 40(2):8:1–8:44, 2015.

[6] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[7] Khanh-Chuong Duong, Mostafa Bamha, Arnaud Giacometti, Haoyuan Li, Arnaud Soulet, and Christel Vrain. MapFIM: Memory Aware Parallelized Frequent Itemset Mining in Very Large Datasets. In *International Conference on Database and Expert Systems Applications (DEXA)*, Lyon, France, August 2017.

[8] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang. Pfp: parallel fp-growth for query recommendation. In *Proc. of RecSys'2008*, pages 107–114. ACM, 2008.

[9] N. Li, L. Zeng, Q. He, and Z. Shi. Parallel implementation of apriori algorithm based on mapreduce. In *Proc. of SNDP'2012*, pages 236–241. IEEE, 2012.

[10] M.-Y. Lin, P-Y. Lee, and S.-C. Hsueh. Apriori-based frequent itemset mining algorithms on mapreduce. In *Proc. of ICUIMC'2012*, pages 76:1–76:8, 2012.

[11] C. Lucchese, S. Orlando, R. Perego, and F. Silvestri. Webdocs: a real-life huge transactional dataset. In *FIMI*, volume 126, 2004.

[12] Apache Mahout. Scalable machine learning and data mining, 2012.

[13] A. Makanju, Z. Farzanyar, A. An, N. Cercone, Hu Z. Z., and Y. Hu. Deep parallelization of parallel fp-growth using parent-child mapreduce. In *Proc. of BigData'2016*, pages 1422–1431. IEEE, 2016.

[14] I. Miliaraki, K. Berberich, R. Gemulla, and S. Zoupanos. Mind the gap: Large-scale frequent sequence mining. In *Proc. of SIGMOD'2013*, pages 797–808. ACM, 2013.

[15] S. Moens, E. Aksehirli, and B. Goethals. Frequent itemset mining for big data. In *Proc. of BigData'2013*, pages 111–118. IEEE, 2013.

[16] S. Salah, R. Akbarinia, and F. Masseglia. Data partitioning for fast mining of frequent itemsets in massively distributed environments. In *Proc. of DEXA'2015*, pages 303–318, 2015.

[17] S. Salah, R. Akbarinia, and F. Masseglia. Optimizing the data-process relationship for fast mining of frequent itemsets in mapreduce. In *Proc. of ICML'2015*, pages 217–231, 2015.

[18] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of VLDB '95*, pages 432–444, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[19] T. Uno, T. Asai, Y. Uchida, and H. Arimura. Lcm: An efficient algorithm for enumerating frequent closed item sets. In *FIMI*, volume 90. Citeseer, 2003.

[20] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, et al. New algorithms for fast discovery of association rules. In *KDD*, volume 97, pages 283–286, 1997.

[21] L. Zhou, Z. Zhong, J. Chang, J. Li, J. Z. Huang, and S. Feng. Balanced parallel fp-growth with mapreduce. In *Proc. of YC-ICT'2010*, pages 243–246, 2010.