ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/317663579

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

Conference Paper · August 2017

DOI: 10.1007/978-3-319-64468-4_36

CITATIONS
4

READS
108

6 authors, including:

Mostafa Bamha
Université d'Orléans
31 PUBLICATIONS **172** CITATIONS

SEE PROFILE

Arnaud Giacometti
University of Tours
71 PUBLICATIONS **610** CITATIONS

SEE PROFILE

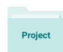
Dominique Li
University of Tours
34 PUBLICATIONS **115** CITATIONS

SEE PROFILE

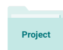
Arnaud Soulet
University of Tours
90 PUBLICATIONS **656** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:



Echantillonnage de motifs sous contrainte View project



Mining medical data View project

All content following this page was uploaded by Arnaud Soulet on 19 October 2018.

The user has requested enhancement of the downloaded file.

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 Francois Rabelais de Tours, LI 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 MapReduce programming models. Many famous FIM algorithms have been parallelized in a MapReduce framework like *Parallel Apriori*, *Parallel FP-Growth* and *Dist-Eclat*. However, most papers focus on work partitioning and/or load balancing but they are not extensible because they require some memory assumptions. A challenge in designing parallel FIM algorithms is thus finding ways to guarantee that data structures used during mining always fit in the local memory of the processing nodes during all computation steps.

In this paper, we propose MapFIM, a two-phase approach for frequent itemset mining in very large datasets relying both on a MapReduce-based distributed Apriori method and a local in-memory method. In our approach, MapReduce is first used to generate local memory-fitted prefix-projected databases from the input dataset benefiting from the Apriori principle. Then an optimized local in-memory mining process is launched to generate all frequent itemsets from each prefix-projected database. Performance evaluation shows 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. This task aims at extracting a set of events (called itemsets) that occur frequently within database entries (called transactions). For more than 20 years, a large number of algorithms have been proposed to mine frequent patterns as efficiently as possible [1]. In big data era, proposing efficient algorithms that handle huge volumes of transactions remains an important challenge due to the memory space required to mine all frequent patterns. To tackle this issue, several proposals have been made to work in distributed environments where the major idea is to distinguish two phases: a global one and a local one. A first global phase uses MapReduce distributed techniques for mining the most frequent patterns whose calculation requires a large part of the data that does not fit in memory. Then, a second local phase mines on a single machine all the supersets of a pattern obtained at the previous phase. Indeed, these supersets can be mined using only a part of the data that can fit in the memory of a single machine. Intuitively the first phase guarantees the possibility of working on a large volume of data while the second phase preserves a reasonable execution time. Unfortunately the current proposals

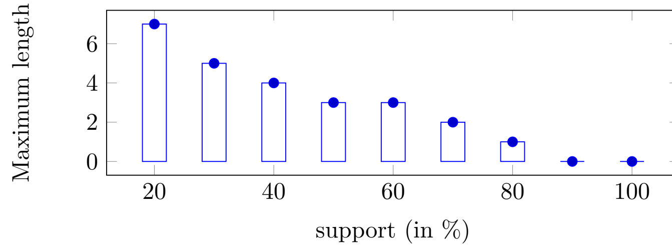


Figure 1: Maximum length of frequent itemsets in dataset Webdocs

fail 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.

Indeed, a major difficulty consists in determining the balance between the global phase and the local phase. If an approach relies too heavily on the local phase, it can only deal with high minimum frequency thresholds where the amount of candidate patterns or the projected database fit in memory. For instance, Parallel FPF algorithm in [7] where the projected databases are distributed cannot deal with low minimum thresholds when at least one projected database does not fit in the memory of a machine. Conversely, if an approach relies too heavily on the global phase, it will be very slow because the cost of communication is high. For instance, Parallel Apriori [8] is quite slow for low thresholds because all patterns are extracted in the global phase. In BigFIM [14], the user sets a minimum length k below which the itemsets are mined globally while the larger itemsets are mined locally as they cover a smaller set of transactions that can fit in memory. The problem is that this length is difficult to determine for the user as it varies, depending on the dataset and on the available memory. To illustrate this issue encountered with the threshold k, Figure 1 plots the maximum length of frequent itemsets with the dataset Webdocs (see Section 5 for details) varying the minimum frequency threshold. In [14], it is suggested to use a global phase with itemsets of size $k = 3$ and for larger itemsets, it is assumed that the conditional databases will fit in the memory. However, from Figure 1, it is easy to see that 3 is not a sufficiently high threshold because there is at least one itemset of size 4 that covers more than 40% of transactions. Moreover, it does not take into account the fact that two patterns of the same size may have very different frequencies. In this paper, we propose a fine-grained method depending on the frequency of each itemset for determining whether it is possible to switch from the global phase to the local phase.

Contributions of the paper. We propose the algorithm MapFIM (Memory aware parallelized Frequent Itemset Mining) 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 prove 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 show how to empirically determine this parameter in practice. Indeed, the higher this threshold, the faster the mining (because more patterns are mined locally). Finally, an experimental section illustrates the extensibility and the efficiency of MapFIM compared to the state-of-the-art algorithms.

Section 2 formulates the problem of frequent itemset mining in an extensible way in order to

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

deal with huge volumes of transactions. Section 3 shows that existing proposals in literature are not extensible. In Section 4, we present how our algorithm MapFIM works and in particular, we detail the two phases (global and local ones) and the switch between the two. In Section 5, we empirically evaluate MapFIM against the state-of-the-art methods by comparing execution times and memory consumption. Section 6 briefly concludes.

2 Problem Formulation

2.1 Frequent Itemset Mining problem

Let $\mathcal{I} = \{i_1 < i_2 < \ldots < i_n\}$ be a set of n ordered 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.**

2.2 The MapReduce Programming Model

MapReduce is a simple yet powerful framework 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.

Google's MapReduce programming model presented in [6] is based on two functions: **map** and **reduce**, that the programmer is supposed to provide to the framework. These two functions should have the following signatures:

$$\textbf{map:} \quad (k_1, v_1) \longrightarrow list(k_2, v_2)\,,$$
$$\textbf{reduce:} \quad (k_2, list(v_2)) \longrightarrow list(v_3)\,.$$

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 $list(v_3)$.

In this paper, we use an open source version of MapReduce, called Hadoop, developed by The Apache Software Foundation. Hadoop framework includes a distributed file system called HDFS[1] designed to store very large files with streaming data access patterns.

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.

2.3 The challenge of extensibility

Guaranteeing the correct execution of a method whatever the volume of input data is a classic challenge in MapReduce framework 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 mapper or reducer). 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 when the set of 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 are added continuously. We then formalize the notion of extensibility with respect to the number of transactions as follows:

Definition 1 (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 number of transactions in $\mathcal{D}$ and the minimum frequent threshold α.

In the remainder of the paper, we aim at proposing the first transaction-extensible FIM method. This goal is clearly a challenge because it is difficult to control the amount of memory required for a frequent itemset mining. The following section reviews the shortcomings of the various literature proposals.

3 Related Work

Due to the explosive growth of data, many parallel methods of frequent pattern mining (FPM) algorithms have been proposed in the literature, mainly to extract frequent itemsets [7–9, 15–

[1] HDFS: Hadoop Distributed File System.

17, 20], but also to extract frequent sequences [5, 13]. In this section, we only consider related work involving the parallelization of FPM algorithms on the MapReduce framework.

A first category of approaches includes works that are specific parallelizations of existing FPM algorithms. For example, different adaptations of Apriori on MapReduce have been proposed [8, 9]. These implementations 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). In Section 4.3, we show how this limitation can be overcome using HDFS to store the set of candidates. Different implementations of FP-Growth on MapReduce also exist [7, 20]. The main idea of these implementations is to distribute the conditional databases of the frequent items to the mappers. However, these proposals do not guarantee that the conditional databases can be stored in the worker nodes, and therefore, these parallelizations of FP-Growth are also not *transaction-extensible*. More recently, Makanju et al. [12] propose to use Parent-Child MapReduce (a new feature of IBM Platform Symphony) to overcome the limitations of the previous implementations of FP-Growth. The authors show that their method can provide significant speed-ups over Parallel FP-Growth [7]. 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 works 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. At a high-level, the methods in the first sub-category carefully partitions the original dataset in such a way that each partition can be mined independently and in parallel [13, 15, 16]. Once partitions have been constructed (in a first global phase), an arbitrary FPM algorithm can be used to mine each partition (in this second local phase, the partition are mined independently and in parallel). In order to maintain completeness, it is important to note that some partitions built by these approaches can overlap and that some interesting pattern can be generated several times. However, these approaches are more efficient than SON Algorithm [17] because locally frequent itemsets are necessarily globally frequent. Thus, compared to SON Algorithm, after the local phase, it is not necessary to compute the supports of the locally frequent itemsets with respect to the whole dataset. Finally, because these approaches cannot guarantee that all the partitions will fit in main memory (of the mappers or reducers), it is important to note 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 interesting pattern is only generated once. We can consider that Parallel FP-Growth (PFP) [8] also belongs to this second sub-category of methods. However, because PFP partitions the search space only considering single frequent items, it is not efficient. In order to overcome this type of limitation, Moens et al. [14] propose to use longer frequent itemsets as prefixes for partitioning the search space. In a first and global phase, their algorithm (called $BigFIM$) mines the frequent k-itemsets using a MapReduce implementation of Apriori. Then, in a second phase, subset of prefixes of length k are passed to worker nodes. These worker nodes use the conditional databases of prefixes to mine interesting patterns that are more specific, assuming that the conditional databases can fit in the main memory of the worker nodes. In practice, note that the choice of the parameter k can be very difficult. Indeed, if the user chooses a value of k that is too low, then BigFIM will not pass (because a conditional database will not fit in main memory). On the other hand, if the user chooses a value of k that is too high, then the first global phase of BigFIM (which computes the frequent k-itemsets) will be time consuming and not efficient. It explains why we propose in this paper a new approach that do not require the involvement of the user to fix a

parameter such as k, and automatically detect when it is possible to switch from a global phase to a local phase.

4 MapFIM: a MapReduce approach for frequent itemset mining

4.1 Overview of the approach

The key idea of our proposal is to enumerate in a breadth-first search manner 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 an itemset has a support sufficiently low to guarantee that the projected database (plus the amount of memory required to enumerate the itemsets) holds in memory. To do this, we introduce a maximum frequency threshold β to indicate when it is possible to switch to the local mining phase. Given a transactional database $\mathcal{D}$ and a maximum support threshold β, an itemset X is said to be *overfrequent* if its support exceeds β: $supp(X, \mathcal{D}) \geq \beta$. In the following, we denote:

- $\mathcal{L}$ the set of frequent itemsets, *e.g.* set of itemsets X such that $supp(X, \mathcal{D}) \geq \alpha$.
- $\mathcal{L}^{>\beta}$ the set of *overfrequent* itemsets, *e.g.* set of itemsets X such that $supp(X, \mathcal{D}) > \beta$.
- $\mathcal{L}^{\leq\beta}$ the set of frequent but not overfrequent itemsets, *e.g.* set of itemsets X such that $\alpha \leq supp(X, \mathcal{D}) \leq \beta$. It is clear that $\mathcal{L}^{\leq\beta} = \mathcal{L} \setminus \mathcal{L}^{>\beta}$.
- $\mathcal{L}_k$, $\mathcal{L}_k^{>\beta}$, $\mathcal{L}_k^{\leq\beta}$ are respectively the set of frequent, overfrequent, frequent but not overfrequent k-itemsets.
- $\mathcal{C}$ the set of candidates. $\mathcal{C}_k$ the set of candidate k-itemsets. $\mathcal{C}_k$ is generated by the join $\mathcal{L}_{k-1} \bowtie \mathcal{L}_{k-1}$.
- $\mathcal{D}'$ the compressed database from $\mathcal{D}$, by removing infrequent items, *e.g.* items that are not in $\mathcal{L}_1$.

More precisely, given a set of items $\mathcal{I}$, a transactional database $\mathcal{D}$, a minimum support threshold α and a maximum support threshold β, the algorithm MapFIM (for Memory aware parallelized Frequent Itemset Mining) enumerates all frequent itemsets by using three phases:

1. **Data preparation:** This phase initializes the process by compressing the transactional database based on frequent 1-itemsets. Let $\alpha = 20\%$ and $\beta = 50\%$. Considering the example given by Table 1, as only a, b, c and d are frequent, the original dataset is compressed as shown by Table 2. Transactions are updated for removing non-frequent items. Transactions t_1 and t_{10} are removed because they cannot contain an itemset of size 2 (or greater). At the end of this phase, we have $\mathcal{L}_1^{>\beta} = \{a, c\}$ and $\mathcal{L}_1^{\leq\beta} = \{b, d\}$.

2. **Global mining:** This phase mines all potentially overfrequent itemsets using Apriori algorithm. An itemset is potentially overfrequent whenever at least one direct subset is overfrequent. For instance, four candidates of size 2 are generated from overfrequent items in $\mathcal{L}_1^{>\beta}$, *e.g.* $\mathcal{C}_2^{>\beta} = \{ab, ac, ad, cd\}$ (these candidate 2-itemsets are potentially overfrequent). The support of all candidates in $\mathcal{C}_2^{>\beta}$ are evaluated during this global phase. But, as their support is greater than α, but below β, $\mathcal{L}_2^{>\beta}$ is empty. No more candidates are generated and MapFIM moves to the next phase.

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 dataset

itemset	Projected datasets
b	$\mathcal{D}_b : \{\{c\}, \{c, d\}, \{c\}\}$
d	$\emptyset$
ab	$\mathcal{D}_{ab} : \{\{c\}, \{c, d\}$
ac	$\mathcal{D}_{ac} : \{\{d\}\}$
ad	$\emptyset$
cd	$\emptyset$

Table 3: Conditional datasets

3. **Local mining:** This phase mines itemsets from non overfrequent itemsets. In our running example, the prefix-based supersets generated from $\mathcal{L}_1^{\leq\beta} = \{b, d\}$ and $\mathcal{L}_2^{\leq\beta} = \{ab, ac, ad, cd\}$ will be evaluated during this phase. Each prefix is considered individually by using a projected database as given in Table 3. Typically, abc will be generated from the prefix ab.

Of course, the maximum support threshold β is a very crucial parameter for balancing the mining process. Section 5.2 will show how to set β in practice. Note that varying this parameter enables us to unify different state-of-the-art methods. By choosing $\beta = \alpha$, MapFIM algorithm is very similar to parallel Apriori [8] as the local mining phase is ignored. At the opposite, with $\beta = 100\%$, MapFIM is similar to parallel FP-Growth algorithm [7] which only relies on a local mining phase after the data preparation. Interestingly, when the maximum support threshold β is between α and 100%, MapFIM benefits from the same ideas as BigFIM with the important difference that we are sure that all prefixes examined by the local mining phase are not overfrequent. Consequently, we are sure that they can be locally processed in-memory.

Sections 4.2, 4.3, and 4.4 detail respectively the three main phases of MapFIM: data preparation, global mining and local mining. Finally, Section 4.5 demonstrates its completeness and its extensibility with respect to the number of transactions.

4.2 Data Preparation

In this phase, frequent items, *e.g.* items in $\mathcal{L}_1$ are found. This can be achieved by adapting the *Word Count* problem [6]. Each item is considered as a word and by using a MapReduce phase for Word Counting problem, we get the support of every item. Then, by using α and β parameters, $\mathcal{L}_1^{\leq\beta}$ and $\mathcal{L}_1^{>\beta}$ are constructed. Finally, the compressed data $\mathcal{D}'$ is generated and put in HDFS. This can be solved by a simple Map phase, where each mapper reads a block of data and removes items which are not in $\mathcal{L}_1$, then emits transactions with at least two frequent items.

4.3 Global mining based on Apriori

This phase is similar to the parallel implementation of Apriori algorithm [8]. The key difference is on the way candidates are generated. In Apriori algorithm, at each iteration, the set of candidates $\mathcal{C}_k$ is generated by the join $\mathcal{L}_{k-1} \bowtie \mathcal{L}_{k-1}$ [2]. From the definition of $\mathcal{L}^{\leq\beta}$ and $\mathcal{L}^{>\beta}$,

[2] In our work, in order to generate each candidate once, we use a prefix-based join operation. More precisely, given two set of k-itemsets $\mathcal{L}_{k-1}$ and $\mathcal{L}'_{k-1}$, the join of $\mathcal{L}_{k-1}$ and $\mathcal{L}'_{k-1}$ is defined by: $\mathcal{L}_{k-1} \bowtie \mathcal{L}'_{k-1} = \{(i_1, \dots, i_k) \mid (i_1, \dots, i_{k-2}, i_{k-1}) \in \mathcal{L}_{k-1} \land (i_1, \dots, i_{k-2}, i_k) \in \mathcal{L}'_{k-1} \land i_1 < \dots < i_{k-1} < i_k\}$.

we have:

$$\begin{aligned}\mathcal{C}_k &= \mathcal{L}_{k-1} \bowtie \mathcal{L}_{k-1} \\ &= (\mathcal{L}_{k-1}^{\leq\beta} \bowtie \mathcal{L}_{k-1}) \cup (\mathcal{L}_{k-1}^{>\beta} \bowtie \mathcal{L}_{k-1})\end{aligned}$$

We define $\mathcal{C}_k^{\leq\beta} = \mathcal{L}_{k-1}^{\leq\beta} \bowtie \mathcal{L}_{k-1}$ and $\mathcal{C}_k^{>\beta} = \mathcal{L}_{k-1}^{>\beta} \bowtie \mathcal{L}_{k-1}$ and thus: $\mathcal{C}_k = \mathcal{C}_k^{\leq\beta} \cup \mathcal{C}_k^{>\beta}$.

The idea developed in this paper is to use MapReduce framework to globally mine candidates in $\mathcal{C}_k^{>\beta}$, and then locally mine $\mathcal{C}_k^{\leq\beta}$ in a local phase. Algorithm 1 presents this algorithm.

At each iteration in Main() function, $\mathcal{C}_k^{>\beta}$ is computed by the join $\mathcal{L}_{k-1}^{>\beta} \bowtie \mathcal{L}_{k-1}$ and sent to all Mappers. The Map function counts the frequency of all the candidates belonging to $\mathcal{C}_k^{>\beta}$ in a parallel way. Let us precise that the counting of the frequency is performed at the end of each Mapper, when the combine function summarizes the emission of 1 for each itemset. Then, the Reduce function sums the frequency obtained by each mapper. If a candidate X is frequent, it is put into $\mathcal{L}_k^{>\beta}$ or $\mathcal{L}_k^{\leq\beta}$ depending on whether it is *overfrequent* or not. In case $\mathcal{C}_k^{>\beta}$ is too large to be handled by Mappers, we partition this set into, for example, l subsets and we run l MapReduce phases instead of one. Finally, the global mining achieves a good load balance because the compressed database $\mathcal{D}'$ is distributed equally among mappers and all mappers handle the same candidate set.

4.4 Local Mining of Frequent Itemsets

As described in the previous sections, the two-phase mining strategy guarantees the efficiency of MapFIM. Indeed, once it is estimated, through the use of the parameter β, that each projected-database with respect to a prefix generated can always be handled by a single node in the cluster, MapFIM switches to the local mining phase.

In the local mining phase, the frequent itemset enumeration is completed by using efficient algorithms (for instance, Eclat or LCM) that fit the memory constraints required by single nodes. This step is still MapReduce driven: local memory-fitted projected-databases are dispatched to each node (as Reducers) that allow to run any local FIM algorithm. The complete local mining process is shown in Algorithm 2.

In the Map phase, we consider frequent itemsets $X \in \mathcal{L}^{\leq\beta}$ as prefixes and construct their projected databases. For each $X \in \mathcal{L}^{\leq\beta}$, let i denote the last item in X. The projected-database $\mathcal{D}'$ is built by: (1) pruning every transaction $t \in \mathcal{D}'$ that does not contain X (2) pruning every item $j \leq i$ since these items cannot expand X due to the prefix-based join. As shown in Algorithm 2, each Mapper reads a block of data, then for each $X \in \mathcal{L}^{\leq\beta}$, it emits every transaction t that contains X after pruning unnecessary items.

In the Reduce phase, 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 $X \in \mathcal{L}^{\leq\beta}$ 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 $\mathcal{L}^{\leq\beta}$ is too large to fit in memory of Mappers, we partition this set, for example, into l subsets and repeat the local mining in l MapReduce phases until every itemset in $\mathcal{L}^{\leq\beta}$ is handled.

An algorithm adapted to the local mining phase must be able to enumerate all the itemsets corresponding to a given prefix in a bounded memory space. 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

Algorithm 1: Global Mining

```
1  Function Main():
2      k = 2;
3      while |L_{k-1}^{>β}| > 0 do
4          C_k^{>β} = L_{k-1}^{>β} ⋈ L_{k-1};
5          Send C_k^{>β} to all Mappers;
6          Map phase;
7          Reduce phase;
8          k = k + 1;
9      return;
10 Function Map(String key, String value):
11     // key: input name, value: input contents;
12     foreach transaction t ∈ value do
13         foreach itemset X ∈ C_k^{>β} do
14             if X ⊆ t then
15                 Emit(X, 1);
16     return;
17 Function Reduce(String key, Iterator values):
18     // key: a candidate X, values: a list of counts;
19     frequency = 0;
20     foreach v ∈ values do
21         frequency = frequency + v;
22     if α * |D| ≤ frequency then
23         L_k = L_k ∪ {X};
24         if frequency ≤ β * |D| then
25             L_k^{≤β} = L_k^{≤β} ∪ {X};
26         else
27             L_k^{>β} = L_k^{>β} ∪ {X};
28     return;
```

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.

Due to the difference in size among projected databases, the local mining can lead to a load imbalance among reducers. In [14], 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. Following this way, we decide in our implementation to assign randomly projected databases to reducers.

4.5 Completeness and Extensibility

Thanks to the complementarity of global and local mining phases, this section demonstrates that MapFIM is correct and complete, but also is transaction-extensible:

Algorithm 2: Local Mining

```
1  Function Map(String key, String value):
2      // key: input name, value: input contents;
3      foreach itemset X ∈ L^{≤β} do
4          Let i = the last item in X;
5          foreach transaction t ∈ value that contains X do
6              Create t' = t;
7              Remove every item j in t' such that j ≤ i;
8              Emit(X, t')
9      return;
10 Function Reduce(String key, Iterator values):
11     // key: an itemset X, values: a list of transactions;
12     Create an empty file f_in in local disk;
13     Save values to f_in;
14     Run a local FIM program with input=f_in, output=f_out, support=α * |values|/|D|;
15     foreach frequent itemset X' ∈ f_out do
16         X'' = X ∪ X';
17         L = L ∪ {X''};
18     return;
```

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.*

Idea of the proof: The algorithm counts the support of each itemset and returns only frequent itemsets, therefore it is correct. We give here an idea of the proof of the completeness. Let I be a k-frequent itemset, $\mathcal{I} = (i_1, \ldots, i_k)$, with $i_1 < i_2 \ldots < i_k$. Let I_j denote $I = (i_1, \ldots, i_j)$.

If $k = 1$, then I is computed during data preparation. If $k > 1$, then we have two cases:

- $supp(I_{k-1}, \mathcal{D}) > \beta$, $I_{k-1} \in \mathcal{L}_{k-1}^{>\beta}$. Then, since $i_k > i_{k-1}$, I is generated and evaluated during the global mining phase.
- $supp(I_{k-1}, \mathcal{D}) \leq \beta$. Let j be the smallest index such that $supp(I_j, \mathcal{D}) \leq \beta$, *i.e.*, $I_j \in \mathcal{L}_j^{\leq\beta}$. Then frequent itemsets starting by I_j will be mined in the local mining step, from the conditional database with respect to I_j. It is built by considering all transactions in $\mathcal{D}$ containing I_j and removing from these transactions all items i with $i \leq i_j$. Since I is ordered, if I is frequent in $\mathcal{D}$ then $\{i_{j+1}, \ldots, i_k\}$ is frequent in the conditional database w.r.t I_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 1:

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

Dataset	# Transactions	# Items	Avg length	FileSize
Webdocs	1,692,082	5,267,656	177	1.48 GB
Synthetic	10,000,000	10,000	50	2.47 GB

Table 4: Characteristic of the two used datasets

Idea of the proof: The first step of data preparation is not a problem as it is similar to a word counting. 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 subsets of candidates. For local mining phase, the mining algorithm for a prefix takes a memory space proportional to the size of its projected database so there is at least one β such that each projected database holds in memory.

5 Experiments

We have chosen the dataset Webdocs [10], 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. It was obtained from the Frequent Itemset Mining Implementations Repository at `http://fimi.ua.ac.be/data/`. We have also generated a synthetic dataset by using the generator from the IBM Almaden Quest research group. Their program can no longer be downloaded and we have used another implementation at `https://github.com/zakimjz/IBMGenerator`. The command used to generate our dataset is: `./gen lit -ntrans 10000 -tlen 50 -nitems 10 -npats 1000 -patlen 4 -fname Synthetic -ascii`

The characteristics of the two datasets are given in Table 4.

5.1 Performance Results

To evaluate the performance of MapFIM presented in this paper, we compared it to Parallel FP-Growth (PFP) [7] and BigFIM algorithms [14]. We believe that PFP and BigFIM are the best approaches for itemset mining in Hadoop MapReduce framework. We implemented MapFIM in Hadoop 2 and for the local mining step, we use a local program based on Eclat/LCM algorithm [18, 19]. The program is implemented in C++ by Borgelt at `http://www.borgelt.net/eclat.html`. PFP implementation is present in the library Apache Mahout 0.8 [11] and BigFIM implementation based on Hadoop 1 is provided by the authors at `https://gitlab.com/adrem/BigFIM-sa`.

All the experiments were performed on a cluster of 3 machines. Each machine has 2 Xeon Cpu E5-2650 @ 2.60 GHz with 32 cores and 64 GB of memory. MapFIM and PFP were tested in Hadoop 2.7.3 while BigFIM was experimented in Hadoop 1.2.1. We configured Hadoop environment to use up to 30 cores and 60 GB of memory for each machine [3]. We have experimented the three approaches with different values of the minimum support threshold α.

In all the experiments, the time was limited to 72 hours and we report the total execution time in seconds. PFP program was tested with its default parameter and BigFIM program was configured with parameter $k = 3$ as suggested by the authors. With this configuration,

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

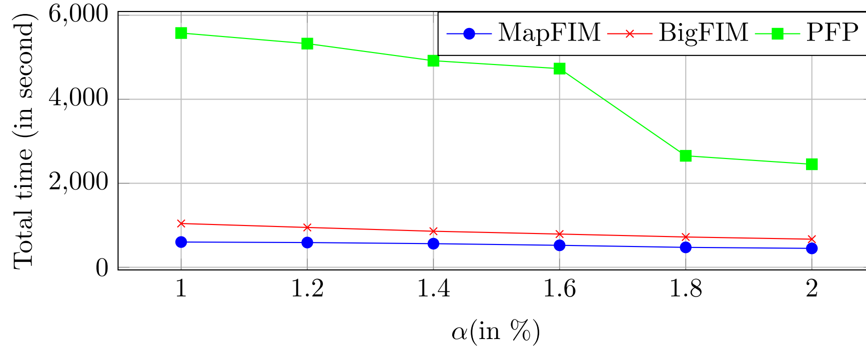


Figure 2: Performance with the Synthetic dataset

α	MapFIM				BigFIM	PFP
	$\beta = 100\%$	$\beta = 50\%$	$\beta = 30\%$	$\beta = \alpha$		
20%	162	360	641	280	421	1278
15%	211	684	1960	466	3370	3882
10%	454	1357	5183	2349	26258	Out of Memory
9%	477	1691	7109	4869	45665	Out of Memory
8%	581	2117	10558	12975	80858	Out of Memory
7%	674	2760	15249	44075	Out of Time	Out of Memory
6%	967	4107	23807	215402	Out of Time	Out of Memory
5%	1804	6705	40832	Out of Time	Out of Time	Out of Memory

Table 5: Performance with the Webdocs dataset

BigFIM uses a parallel Apriori approach to mine all 3-frequent itemsets before switching to global mining. It is shown in [14] that with $k = 3$, BigFIM achieves good performance.

For the dataset Synthetic The value of the minimum support threshold α varies from 1% to 2%. In this dataset, there is no itemset whose support is greater than 30% and for that reason, MapFIM with $\beta = 30\%$ or $\beta = 50\%$ or $\beta = 100\%$ takes the same amount of execution time. All three approaches can enumerate all the frequent itemsets without running out of memory and the results of MapFIM (with $\beta = 30\%$), BigFIM and PFP are shown in Figure 2. It is clear that PFP is the slowest while MapFIM and BigFIM are comparable. This dataset is generated randomly and there is no long frequent itemsets. Indeed, most of the frequent 3-itemset appears in only 9% of transactions. As a consequence, both BigFIM and MapFIM can achieve a good workload balancing and a good performance.

For the dataset Webdocs The three programs were tested with various values of the minimum support threshold α. In MapFIM, we set the value of β to $100\%, 50\%, 30\%$ and $\beta = \alpha$. As shown in Figure 1, this dataset is expected to be hard to mine as it has long frequent itemsets as well as itemsets that are very frequent. 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 with the dataset Webdocs are shown in Table 5. It is surprising that PFP cannot

solve the dataset Webdocs with a support below to 15%, but it requires a huge memory for the Reduce phase: with $\alpha = 15\%$, each reducer in PFP can take as much as 60 GB of memory. On the contrary, both MapFIM and BigFIM are effective in memory and never run out of memory in our setup. The results show that MapFIM outperforms both BigFIM and PFP, especially for low values of support. As expected, our algorithm works better with higher value of β. However, in case when the support α is equal or higher than 9%, MapFIM with $\beta = 30\%$ has a worse performance than with $\beta = \alpha$. Indeed, MapFIM with $\beta = \alpha$ ignores completely the Local Mining step and it is similar to parallel Apriori algorithm. With a high value of support, the Apriori approach is still effective because the number of candidates is not huge. However, when α is lower, applying the Local Mining step is efficient, as shown in our experiment.

5.2 Estimating β Parameter

In our algorithm, a good value of β is important for getting high performance. The higher the value of β is, the better performance we get in general but more memory is required. In this subsection, we present a method for estimating a good value of β.

It is proven in [18] that LCM algorithm requires an amount of memory linear to the input size. As we use a local program based on Eclat/LCM algorithm [18, 19], we expect that the program requires a maximum of $f(input_size)$ of memory, where $f()$ is a linear function. To simplify the estimation, we suppose that the maximum memory needed by the program is $\gamma \times input_size$, where $input_size$ is measured as the total length of all transactions in the dataset. We try to figure out the value of γ by experiments with various datasets. With each dataset, we run the program with support = 0% to report the maximum memory used during one hour by the program. Then we compute $\gamma = \frac{input_size}{max_memory}$ and report the result in Table 6. In these experiments, we use datasets from Frequent Itemset Mining Implementations Repository at `http://fimi.ua.ac.be/data/`.

From experiments, the value of γ varies from 0.017 to 0.043, with an average value of 0.023 and a standard deviation of 0.00785. For instance, the value of γ for dataset Webdocs is 0.018. Next, we test if $\gamma = 0.018$ is a good value to estimate β with dataset Webdocs. We run MapFIM with different values of β from 100% to 20% and the minimum support threshold $\alpha = 10\%$. The approximate memory required is computed by: $\gamma \times \beta \times input_size$, where $input_size$ is the total length of the transactions in the compressed data $\mathcal{D}'$. We report the approximate memory required *w.r.t* $\gamma = 0.018$ and the real value of maximum memory used by the local program during the mining.

The result is expressed in Figure 3 and as expected, the real value of max memory is always lower but not much lower than the approximate memory calculated.

From those observations, we propose to set the value of β in MapFIM by:

$$\beta = \frac{M_{Reduce} - M_{reduce_task}}{\gamma \times input_size} \tag{1}$$

where M_{Reduce} is the limit of memory of a Reducer, M_{reduce_task} is the memory required for a reduce task without running the local mining program[4] and $input_size$ is the total length of transactions in the compressed dataset $\mathcal{D}'$.

[4]In our implementation, M_{reduce_task} is around 300 MB

Dataset	*input_size*	*max_memory* (in Kilobyte)	γ
accidents	11500870	228400	0.020
connect	2904951	58160	0.020
kosarak	8019015	193644	0.024
pumsb	3629404	63332	0.017
retail	908576	25588	0.028
T40I10D100K	3960507	71988	0.018
T10I4D100K	1010228	25916	0.026
chess	118252	5100	0.043
pumsb_star	2475947	47424	0.019
webdocs	299887139	5422024	0.018

Table 6: the γ value with Borgelt's implementation of Eclat/LCM

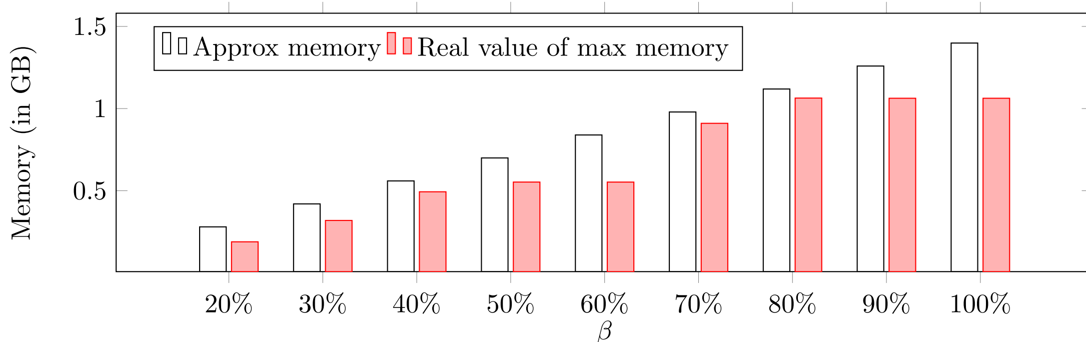


Figure 3: Memory on the Webdocs dataset

6 Conclusion and future work

In this paper, we present MapFIM, 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 phase. Moreover, we show that our method is transaction-extensible, meaning that given a fixed set of 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 to guarantee this property.

Our experimental evaluations show that MapFIM outperforms the best existing MapReduce based frequent itemset mining approaches. Moreover, we show how to calibrate and set the unique parameter β of our algorithm. This point is particularly important, since an optimal value of parameter β guarantees a high performance level.

Future work will be devoted to make MapFIM scalable. This can be achieved by using similar approaches, based on randomized key redistributions introduced in [3, 4] for join processing, allowing to avoid the effects of data skew while guaranteeing perfect balancing properties during

all the stages of join computation in large scale systems even for a highly skewed data.

Acknowledgement

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

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] 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.

[8] 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.

[9] 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.

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

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

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

View publication stats