# Frequent Itemset Mining for Big Data

**Conference Paper** · October 2013

DOI: 10.1109/BigData.2013.6691742



**3 authors**, including:

Bart Goethals
University of Antwerp
**168** PUBLICATIONS **5,099** CITATIONS

SEE PROFILE

# Frequent Itemset Mining for Big Data

Sandy Moens, Emin Aksehirli and Bart Goethals
Universiteit Antwerpen, Belgium Email: firstname.lastname@uantwerpen.be

***Abstract*—Frequent Itemset Mining (FIM) is one of the most well known techniques to extract knowledge from data. The combinatorial explosion of FIM methods become even more problematic when they are applied to Big Data. Fortunately, recent improvements in the field of parallel programming already provide good tools to tackle this problem. However, these tools come with their own technical challenges, e.g. balanced data distribution and inter-communication costs. In this paper, we investigate the applicability of FIM techniques on the MapReduce platform. We introduce two new methods for mining large datasets: Dist-Eclat focuses on speed while BigFIM is optimized to run on really large datasets. In our experiments we show the scalability of our methods.**

***Index Terms*—distributed data mining; mapreduce; hadoop; eclat**

## I. Introduction

Since recent developments (in technology, science, user habits, businesses, etc.) gave rise to production and storage of massive amounts of data, not surprisingly, the intelligent analysis of big data has become more important for both businesses and academics.

Already from the start, Frequent Itemset Mining (FIM) has been an essential part of data analysis and data mining. FIM tries to extract information from databases based on frequently occurring events, i.e., an event, or a set of events, is interesting if it occurs frequently in the data, according to a user given minimum frequency threshold. Many techniques have been invented to mine databases for frequent events [5], [7], [31]. These techniques work well in practice on typical datasets, but they are not suitable for truly Big Data.

Applying frequent itemset mining to large databases is problematic. First of all, very large databases do not fit into main memory. In such cases, one solution is to use levelwise breadth first search based algorithms, such as the well known Apriori algorithm [5], where frequency counting is achieved by reading the dataset over and over again for each size of candidate itemsets. Unfortunately, the memory requirements for handling the complete set of candidate itemsets blows up fast and renders Apriori based schemes very inefficient to use on single machines. Secondly, current approaches tend to keep the output and runtime under control by increasing the minimum frequency threshold, automatically reducing the number of candidate and frequent itemsets. However, studies in recommendation systems have shown that itemsets with lower frequencies are more interesting [23]. Therefore, we still see a clear need for methods that can deal with low frequency thresholds in Big Data.

Parallel programming is becoming a necessity to deal with the massive amounts of data, which is produced and consumed more and more everyday. Parallel programming architectures, and hence the algorithms, can be grouped into two major subcategories: shared memory and distributed (share nothing). On shared memory systems, all processing units can concurrently access a shared memory area. On the other hand, distributed systems are composed of processors that have their own internal memories and communicate with each other by passing messages [6]. It is easier to adapt algorithms to shared memory parallelism in general, but they are typically not scalable enough [6]. Distributed systems, in theory, allow quasi linear scalability for well adapted programs. However, it is not always easy to write or even adapt the programs for distributed systems, i.e., algorithmical solutions to common problems may have to be reinvented. Thanks to the scalability of distributed systems, not only in terms of technical availability but also cost, they are becoming more and more common.

While distributed systems are becoming more common, they are also becoming easier to use. In contrast, Message Passing Interface (MPI), one of the most common frameworks for scientific distributed computing, works efficiently only on low level programming languages, i.e., C and Fortran. It is known, however, that higher level languages are more popular in businesses [12]. Although they are not the most efficient in terms of computation or resources, they are easily accessible.

Fortunately, thanks to the MapReduce framework proposed by Google [10], which simplifies the programming for distributed data processing, and the Hadoop implementation by Apache Foundation [2], which makes the framework freely available for everyone, distributed programming is becoming more and more popular. Moreover, recent commercial and non-commercial systems and services improve the usability and availability for anyone. For example; the Mahout framework by Apache Foundation [3], which provides a straight forward usability for the most common machine learning techniques, can be set up and run on full-blown clusters on the cloud, having more than hundreds of processors in total, in less than 1 hour without prior experience with the system.

We believe that, although the initial design principles of the MapReduce framework do not fit well for the Frequent Itemset Mining problem, it is important to provide MapReduce with efficient and easy to use data mining methods, considering its availability and wide spread usage in industry.

In this paper, we introduce two algorithms that exploit the MapReduce framework to deal with two aspects of the challenges of FIM on Big Data: (1) Dist-Eclat is a MapReduce implementation of the well known Eclat algorithm [31], optimized for speed in case a specific encoding of the data fits into memory. (2) BigFIM is optimized to deal with truly Big

Data by using a hybrid algorithm, combining principles from both Apriori and Eclat, also on MapReduce. Implementations are freely available at http://adrem.ua.ac.be/bigfim.

In this paper, we assume familiarity with the most well-known FIM techniques [5], [31], [17] and with the MapReduce framework. In the next Section, we will shortly revise the basic notions, but we refer the reader to [15], [10] for good, short surveys on these topics.

## II. Preliminaries

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_n\}$ be a set items, a transaction is defined as $\mathcal{T} = (\mathit{tid}, X)$ where *tid* is a transaction identifier and $X$ is a set of items over $\mathcal{I}$. A transaction database $\mathcal{D}$ is a set of transactions. Its vertical database $\mathcal{D}'$ is a set of pairs that are composed of an item and the set of transactions $\mathcal{C}_j$ that contain the item:

$$\mathcal{D}' = \{(i_j, \mathcal{C}_{i_j} = \{\mathit{tid} \,|\, i_j \in X, (\mathit{tid}, X) \in \mathcal{D}\})\}.$$

$\mathcal{C}_{i_j}$ is also called the *cover* or *tid-list* of $i_j$.

The support of an itemset $Y$ is the number of transactions that contain the itemset. Formally,

$$\mathit{support}(Y) = |\{\mathit{tid} \,|\, Y \subseteq X, (\mathit{tid}, X) \in \mathcal{D}\}|$$

or, in vertical database format

$$\mathit{support}(Y) = |\bigcap_{i_j \in Y} \mathcal{C}_{i_j}|.$$

An itemset is said to be *frequent* if its support is greater than a given threshold $\sigma$, which is called *minimum support* or *MinSup* in short. Frequency is a monotonic property w.r.t. set inclusion, meaning that if an itemset is not frequent, none of its supersets are frequent. Similarly, if an itemset is frequent, all of its subsets are frequent.

Items in an itemset can be represented in a fixed order, without loss of generality let us assume that items in itemsets are always in the same order as they are in $\mathcal{I}$, i.e., $Y = \{i_a, i_b, \ldots, i_c\} \Leftrightarrow a < b < c$. Then, the common *k-prefix* of two itemsets are the first $k$ elements in those sets that are the same, e.g., sets $Y = \{i_1, i_2, i_4\}$ and $X = \{i_1, i_2, i_5, i_6\}$ have a common 2-*prefix* of $\{i_1, i_2\}$. A *prefix tree* is a tree structure where each path represents an itemset, which is exactly the path from the root to the node, and sibling nodes share the same prefix. Note that all the possible itemsets in $\mathcal{I}$, i.e., the power set of $\mathcal{I}$, can be expressed as a prefix tree.

*Projected* or *conditional* database of an item $i_a$ is the set of transactions in $\mathcal{D}$ that includes $i_a$.

Apriori [5] starts by finding the frequent items by counting them, making a pass over $\mathcal{D}$. Then, it combines these frequent items to generate *candidate itemset*s of length 2, and counts their supports by making another pass over $\mathcal{D}$, removing infrequent candidates. The algorithm iteratively continues to extend $k$-length candidates by one item and counts their supports by making another pass over $\mathcal{D}$ to check whether they are frequent. Exploiting the monotonic property, Apriori prunes those candidates for which a subset is known to be infrequent. Depending on the minimum support threshold used, this greatly reduces the search space of candidate itemsets.

Eclat [31] traverses the prefix tree in a depth first manner to find the frequent itemsets. The monotonic property states that if an itemset, a path in the prefix tree, is infrequent then all of its subtrees are infrequent. Thus, if an itemset is found to be infrequent, its complete subtree is immediately pruned. If an itemset is frequent then it is treated as a prefix and extended by its immediate siblings to form new itemsets. This process continues until the complete tree has been traversed. Eclat uses a vertical database format for fast support computation. Therefore, it needs to store $\mathcal{D}'$ in main memory.

MapReduce [10] is a parallel programming framework that provides a relatively simple programming interface together with a robust computation architecture. MapReduce programs are composed of two main phases. In the *map* phase, each mapper processes a distinct chunk of the data and produces key-value pairs. In the *reduce* phase, key-value pairs from different mappers are combined by the framework and fed to reducers as pairs of key and value lists. Reducers further process these intermediate parts of information and output the final results.

## III. Related Work

Data mining literature employs parallel methods already since its very early days [4], and many novel parallel mining methods as well as proposals that parallelize existing sequential mining techniques exist. However, the number of algorithms that are adapted to the MapReduce framework is rather limited. In this section we will give an overview of the data mining algorithms on MapReduce. For an overview of parallel FIM methods in general, readers are kindly referred to [32], [19], [24], [25], [34].

Lin et al. propose three algorithms that are adaptations of Apriori on MapReduce [21]. These algorithms all distribute the dataset to mappers and do the counting step in parallel. *Single Pass Counting* (*SPC*) utilizes a MapReduce phase for each candidate generation and frequency counting steps. *Fixed Passes Combined-Counting* (*FPC*) starts to generate candidates with $n$ different lengths after $p$ phases and counts their frequencies in one database scan, where $n$ and $p$ are given as parameters. *Dynamic Passes Counting* (*DPC*) is similar to *FPC*, however $n$ and $p$ is determined dynamically at each phase by the number of generated candidates.

The *PApriori* algorithm by Li et al. [20] works very similar to *SPC*, although they differ on minor implementation details.

MRApriori [16] iteratively switches between vertical and horizontal database layouts to mine all frequent itemsets. At each iteration the database is partitioned and distributed across mappers for frequency counting.

The iterative, level-wise structure of Apriori based algorithms does not fit well into the MapReduce framework because of the high overhead of starting new MapReduce cycles. Furthermore, although thanks to breadth-first search Apriori can quickly produce short frequent itemsets, because

of the combinatorial explosion it can not handle long frequent itemsets efficiently.

*Parallel FP-Growth* (*PFP*) [18] is a parallel version of the well-known FP-Growth [17]. *PFP* groups the items and distributes their conditional databases to the mappers. Each mapper builds its corresponding FP-tree and mines it independently. Zhou et al. [35] propose to use frequencies of frequent items to balance the groups of PFP. The grouping strategy of PFP is not efficient neither in terms of memory nor speed. It is possible for some of the nodes to read almost the complete database into memory, which is very prohibitive in the field of Big Data. Zhou et al. propose to balance distribution for faster execution using singletons, however as we discuss further in the paper, partitioning the search space using single items is not the most efficient way.

Malek and Kadima propose an aproximate FIM method that uses $k$-medoids to cluster transactions and uses the clusters' representative transactions as candidate itemsets [22]. The authors implemented a MapReduce version that parallelizes the support counting step.

The PARMA algorithm by Riondato et al. [27] finds approximate collections of frequent itemsets. The authors guarantee the quality of the frequent itemsets that are being found, through analytical results.

Some work exists that aims to improve the applicability of the MapReduce framework to data mining. For example, TWISTER [11] improves the performance between MapReduce cycles, or NIMBLE [14] provides better programming tools for data mining jobs. Unfortunately, none of these frameworks are as widely available as the original MapReduce. We therefore focus on an implementation that uses only the core MapReduce framework.

From a practical point of view, not many options are available to mine exact frequent itemsets on the MapReduce framework. To the best of our knowledge, PFP is the best, if not only, available implementation [3]. However, our experiments show that it has serious scalability problems. We discuss these issues in more detail in Section VI-B.

## IV. Search Space Distribution

The main challenge in adapting algorithms to the MapReduce framework is the limitation of the communication between tasks, which run as batches in parallel. All the tasks that have to be executed should be defined at start-up, such that nodes do not have to communicate with each other during task execution. Fortunately, the prefix tree that is used by Eclat can be partitioned into independent groups. Each one of these independent groups can be mined separately on different machines. However, since the total execution time of an algorithm highly depends on the running time of the longest running sub task, balancing the running times is a crucial aspect for decreasing the total computation time [35].

In the case of Eclat, running time is a function of the number of frequent itemsets (FIs) in the partition. However, this can only be estimated with some boundaries [8], [13].

To estimate the computation time of a sub-tree, it has been proposed to use the total number of itemsets that can be generated from a prefix [31] [26], the logarithm of the order of the frequency of items [35], and the total frequency of items in a partition [4] [24] as estimators.

The total number of itemsets is not a good estimator for partitioning: since the monotonic property efficiently prunes the search space, large portions will never have to be generated or explored. On the other hand, assigning the prefixes to worker nodes using weights can provide a much better load balancing. However, the benefit of complicated techniques is not significant compared to a simple Round-Robin assignment.

Our experiments, shown in Section VI-A, suggest that the most important factor for obtaining a balanced partitioning is the length of the prefixes. In fact we can start Eclat at any depth $k$ of the prefix tree using all frequent $k$-FIs as seeds. These seeds can be partitioned among the available worker nodes. Partitioning the tree at lower depths (using longer FIs) provides better balancing. This behaviour is expected because longer FIs not only give more information about the data but they also avail the finer partitioning of the tree.

Our proposed algorithms exploit the inter-independence of sub-trees of a prefix tree and uses longer FIs as prefixes for a better load balancing. Both algorithms mine the frequent itemsets, in parallel, up to a certain length to find frequent $k$-length prefixes, $\mathcal{P}_k = \{p_1, p_2, \ldots, p_m\}$. Then, $\mathcal{P}_k$ is partitioned into $n$ groups $(\mathcal{P}_k^1, \mathcal{P}_k^2, \ldots, \mathcal{P}_k^n)$, where $n$ is the number of distributed workers.

Each group of prefixes, $\mathcal{P}_k^j$, is passed to a worker node and is used as a conditional database. Since each sub-tree for a distinct prefix is unique and not dependent on other sub-trees, each node can work independently without causing any overlaps. Frequent itemsets that are discovered by individual workers require no further post-processing.

Prefix trees are formed in a way that siblings are sorted by their individual frequency in ascending order. Formally, $I = (i_1, i_2, \ldots, i_n)$ where $\mathit{support}(i_a) \leq \mathit{support}(i_b) \Leftrightarrow a < b$. This ordering helps to prune the prefix tree at lower depths and provides shorter run times. The benefits of this ordering are discussed by Goethals [15].

Details of the algorithms are explained in the following section.

## V. Frequent Itemset Mining on MapReduce

We propose two new methods for mining frequent itemsets in parallel on the MapReduce framework where frequency thresholds can be set low. Our first method, called Dist-Eclat, is a pure Eclat method that distributes the search space as evenly as possible among mappers. This technique is able to mine large datasets, but can be prohibitive when dealing with massive amounts of data. Therefore, we introduce a second, hybrid method that first uses an Apriori based method to extract frequent itemsets of length $k$ and later on switches to Eclat when the projected databases fit in memory. We call this algorithm BigFIM.

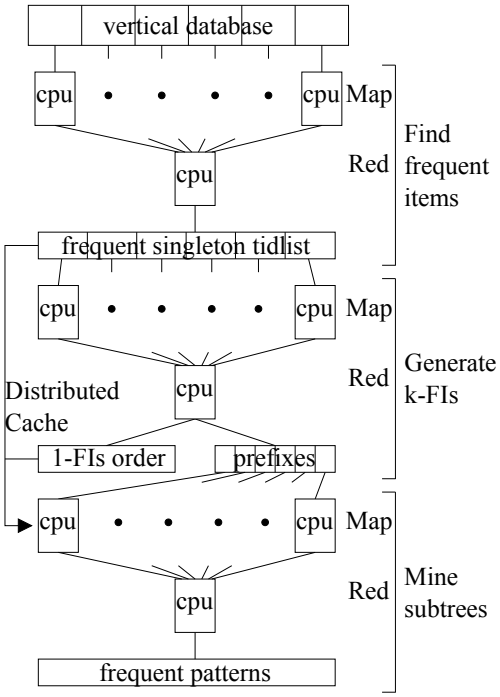



Figure 1: Eclat on MapReduce framework

### A. Dist-Eclat

Our first method is a distributed version of Eclat that partitions the search space more evenly among different processing units. Current techniques often distribute the workload by partitioning the transaction database into equally sized sub databases, also called shards, e.g., the Partition algorithm [28]. Each of the sub databases can be mined separately and the results can then be combined. However, all local frequent itemsets should be combined and counted again to prune the globally infrequent ones, which is expensive.

First of all, this approach comes with a large communication cost, i.e., the number of sets to be mined can be very large, moreover, the number of sets that have to be recounted can be very large as well. Implementing such partitioning technique in Hadoop is therefore prohibitive. A possible solution for the recounting part, is to mine the sub databases with a lower threshold, hence, decreasing the number of itemsets that might have been missed. However, another problem then occurs: indeed, each shard defines a local sub database for which the local structure can be very different from the rest of the data. As a result, computation of the frequent itemsets can blow up tremendously for some shards, although many of the sets are actually local structures and far from interesting globally.

We argue that our method does not have to deal with such problems, since we are dividing the search space rather than the data space. Therefore, no extra communication between mappers is necessary and no checking of overlapping mining results has to be accounted for. We also claim that Eclat, more specifically Eclat using diffsets [33], memory-wise is the best fit for mining large datasets.

First of all, Eclat uses the depth-first approach, such that only a limited number of candidates have to be kept in memory even while finding long frequent patterns. In contrast, Apriori, the breadth-first approach, has to keep all $k$-sized frequent sets in memory when computing $k$+1-sized candidates. Secondly, using diffsets limits the memory overhead approximately to the size of the original tid-list root of the current branch [15]. This is easy to see: a diffset represents the tids that have to be subtracted from the tid-list of the parent to obtain the tid-list for this node, so, at most the tids that can be subtracted on a complete branch extension is the size of the original tid-list.

Dist-Eclat operates in a three step approach as shown in Figure 1. Each of the steps can be distributed among multiple mappers to maximally benefit from the cluster environment. For this implementation we do not start with a typical transaction database, rather we utilize immediately the vertical database format.

1) **Finding the Frequent Items:** During the first step, the vertical database is divided into equally sized blocks (shards) and distributed to available mappers. Each mapper extracts the frequent singletons from its shard. In the reduce phase, all frequent items are gathered without further processing.

2) **$k$-FIs Generation:** In this second step, $\mathcal{P}_k$, the set of frequent itemsets of size $k$, is generated. First, frequent singletons are distributed across $m$ mappers. Each of the mappers finds the frequent $k$-sized supersets of the items by running Eclat to level $k$. Finally, a reducer assigns $\mathcal{P}_k$ to a new batch of $m$ mappers. Distribution is done using Round-Robin.

3) **Subtree Mining:** The last step consists of mining the prefix tree starting at a prefix from the assigned batch using Eclat. Each mapper can complete this step independently since sub-trees do not require mutual information.

### B. BigFIM

Our second method overcomes two problems inherent to Dist-Eclat. First, mining for $k$-FIs can already be infeasible. Indeed, in the worst case, one mapper needs the complete dataset to construct all 2-FIs pairs. Considering Big Data, the tid-list of even a single item may not fit into memory. Secondly, most of the mappers require the whole dataset in memory in order to mine the sub-trees (cf. Section VI-A). Therefore the complete dataset has to be communicated to different mappers, which can be prohibitive for the given network infrastructure.

1) **Generating $k$-FIs:** BigFIM covers the problem of large tid-lists by generating $k$-FIs using the breadth-first method. This can be achieved by adapting the Word Counting problem for documents [10], i.e., each mapper receives part of the database (a document) and reports the items/itemsets (the words) for which we want to know the support (the count). A reducer combines all local frequencies and reports only the globally frequent items/itemsets. These frequent itemsets can be redistributed to all mappers to act as candidates for the next step of breadth-first search. These steps can be repeated $k$ times, to obtain the set of $k$-FIs.

Computing the $k$-FIs in a level-wise fashion is the most logical choice: we do not have to keep large tid-lists in memory, rather we need only the itemsets that have to be counted. In Apriori, for the first few levels, keeping the candidates in memory is still possible—as opposed to continuing this process to greater depths. Alternatively, when a set of candidates does not fit into memory, partitioning the set of candidates across mappers can resolve the problem.

2) **Finding Potential Extensions:** After computing the prefixes, the next step is computing the possible extensions, i.e., obtaining tid-lists for $(k+1)$-FIs. This can be done similar to Word Counting, however, now instead of local support counts we report the local tid-lists. A reducer combines the local tid-lists from all mappers to a single global tid-list and assigns complete prefix groups to different mappers.

3) **Subtree Mining:** Finally, the mappers work on individual prefix groups. A prefix group defines a conditional database that completely fits into memory. The mining part then utilizes diffsets to mine the conditional database for frequent itemsets using depth-first search.

Using 3-FIs as prefixes generally yields well-balanced distributions (cf. Section VI-A). Unfortunately, when dealing with large datasets, a set of 3-FIs extensions can still be too large to fit into memory. In such cases we can continue the iterative process until we reach a set of $k$-FIs that are small enough. To make an estimate on the size of the prefix extensions, the following heuristic can be used. Given a prefix $p$ with support $s$ and order $r$ out of $n$ items, the size of the conditional database described by $p$ is at most $s*(n-(r+1))$. Recall that when a set of candidates does not fit in to memory, the set of candidates can also be distributed and then the diffsets does not incur an exponential blow up in memory usage.

### C. Implementation details

In our methods frequent itemsets are mined in step 3 by the mappers and then communicated to the reducer. To reduce network traffic, we encoded the mined itemsets using a compressed trie string representation for each batch of patterns, similar to the representation introduced by Zaki [30]. Basically, delimiters indicate if the tree is traversed downwards or upwards, and if a support is specified. As an example:

| | | |
|---|---|---|
| a (300)<br>a b c (100)<br>a b d (200)<br>b d (50) | → | a(300)\|b\|c(100)$d(200)$$$b\|d(50) |

As a second remark, we point out that our algorithm computes a superset of the closed itemsets found in a transaction dataset. We can easily do so by letting the individual mappers report only the closed sets in their subtree. Although it is perfectly possible to mine the correct set of closed itemsets, we omitted this post-processing step in our method.

## VI. Experiments

### A. Load Balancing

We examine the load balancing in two ways: the relation between the workload and (1) the length of the distributed prefixes, (2) the assignment scheme.

Let the set of $k$-prefixes $\mathcal{P}_k = \{p_1, p_2, \ldots, p_m\}$ be partitioned to $n$ workers, $\mathcal{P}_k^1, \mathcal{P}_k^2, \ldots, \mathcal{P}_k^n$, where $\mathcal{P}_k^j$ is the set of prefixes assigned to worker $j$, then the prefixes are assigned to worker nodes using the following methods:

- **Round-Robin:** $p_i$ is assigned to the worker $\mathcal{P}_k^{(i \bmod n)}$.
- **Equal Weight:** When $p_i$ is assigned to a worker, *support*$(p_i)$ is added the score of that worker. $p_{i+1}$ is assigned to a worker with the lowest score. Assignment is order dependent.
- **Block partitioning:** $\{p_1, \ldots, p_{\lceil m/n \rceil}\}$ are assigned to $\mathcal{P}_k^1$, $\{p_{(\lceil m/n \rceil+1)}, \ldots, p_{(2\times\lceil m/n \rceil)}\}$ are assigned to $\mathcal{P}_k^2$, and so on.
- **Random:** Each $p_i$ is assigned to a random worker.

For this set of experiments we use 4 different datasets: Abstracts provided by De Bie [9], T10I4D100K, Mushroom and Pumsb are from FIMI repository [1]. Properties of these datasets are given in Table I.

We first mine the datasets to find prefixes of length 1, 2, and 3, then we distribute these prefixes to 128 workers using the assignment methods above. The number of FIs that are generated by each worker is used to estimate the amount of work done by that worker. Because of the high pruning ratio at lower depths, this is an accurate estimate of total work of a worker. Note that, since the number of FIs that are mined in the first step increases with the prefix length, the amount of distributed load decreases.

Table IV shows the standard deviation (StDev), average (Avg) and the difference between the maximum and the minimum (Max-Min) of the workloads. Since we are trying to decrease the total running time by balancing the workloads, the most important indicator of the balancing is the Max-Min.

Figure 2 shows the number of frequent itemsets generated by 8 workers when the prefixes of length 1, 2, 3 are assigned using Round-Robin.

Our experiments suggest independence of the assignment method: using longer prefixes results in a better balancing of the load.

Generating the longer prefixes requires additional initial computation. However, for large databases this computation is negligible compared to the entire mining process. Table II shows the 1, 2 and 3 length frequent itemsets along with the total number of frequent itemsets in different databases. For example in Abstracts, a small dataset, finding 3-FIs is half of the total work, on the other hand; in Pumsb, a larger dataset, the number of 3-FIs is negligible to the total number of FIs, thus, it is still viable to distribute the remaining work after finding long prefixes.

The allocation schemes that have been used in our experiments (c.f. Section VI-A) are able to distribute the search space

Table IV: Prefix assignments with different methods and prefix lengths.

| | | Round-Robin | | | Equal Weight | | | Block | | | Random | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $\mathcal{P}_1$ | $\mathcal{P}_2$ | $\mathcal{P}_3$ | $\mathcal{P}_1$ | $\mathcal{P}_2$ | $\mathcal{P}_3$ | $\mathcal{P}_1$ | $\mathcal{P}_2$ | $\mathcal{P}_3$ | $\mathcal{P}_1$ | $\mathcal{P}_2$ | $\mathcal{P}_3$ |
| Abstracts | StDev | 1071 | 311 | 107 | 1071 | 371 | 118 | 7235 | 4850 | 1835 | 2260 | 432 | 150 |
| | Average | 3025 | 2744 | 1696 | 3025 | 2744 | 1696 | 3025 | 2744 | 1696 | 3025 | 2744 | 1696 |
| | Max-Min | 4488 | 1695 | 482 | 4488 | 2223 | 675 | 35314 | 29606 | 9840 | 11540 | 2756 | 686 |
| T10I4D100K | StDev | 103 | 57 | 34 | 103 | 64 | 32 | 138 | 137 | 68 | 130 | 62 | 32 |
| | Average | 213 | 142 | 85 | 213 | 142 | 85 | 215 | 142 | 87 | 213 | 140 | 85 |
| | Max-Min | 666 | 384 | 183 | 666 | 499 | 179 | 723 | 699 | 471 | 1002 | 333 | 239 |
| Mushroom | StDev | 13287 | 6449 | 4096 | 13287 | 6052 | 3596 | 13287 | 10026 | 7518 | 13287 | 5736 | 3679 |
| | Average | 4488 | 4482 | 4446 | 4488 | 4482 | 4446 | 4488 | 4482 | 4446 | 4488 | 4482 | 4446 |
| | Max-Min | 98303 | 36348 | 23814 | 98303 | 33453 | 24711 | 98303 | 69626 | 40406 | 98303 | 30991 | 24830 |
| Pumsb | StDev | 3897683 | 1955503 | 1112237 | 3897683 | 2077845 | 1003724 | 3897683 | 3442159 | 2598825 | 3897683 | 2065441 | 1191945 |
| | Average | 1296121 | 1296113 | 1296037 | 1296121 | 1296113 | 1296037 | 1296121 | 1296113 | 1296037 | 1296121 | 1296113 | 1296037 |
| | Max-Min | 21342943 | 9809612 | 6167970 | 21342943 | 10089432 | 4815391 | 21342943 | 20059553 | 14507567 | 21342943 | 10534931 | 5827088 |

Table I: Properties of datasets for our experiments

| Dataset | Number of Items | Number of Transactions |
|---|---|---|
| Abstracts | 4,976 | 859 |
| T10I4D100K | 870 | 100,000 |
| Mushroom | 119 | 8,124 |
| Pumsb | 2,113 | 49,046 |
| Tag | 45,446,863 | 6,201,207 |

Table II: Number of total and 1, 2 and 3 length frequent itemsets for datasets.

| Dataset | MinSup | Total | 1-FIs | 2-FIs | 3-FIs |
|---|---|---|---|---|---|
| Abstracts | 5 | 388,631 | 1,393 | 37,363 | 171,553 |
| T10I4D100K | 100 | 27,169 | 797 | 9,627 | 16,742 |
| Mushroom | 812 | 11,242 | 56 | 664 | 2,816 |
| Pumsb | 24,523 | 22,402,411 | 52 | 968 | 9,416 |

Table III: Data ratio needed for different computation units

| | FIs | Avg | Max | Min | StDev |
|---|---|---|---|---|---|
| Abstracts | 1 | 0.29 | 0.66 | 0.20 | 0.09 |
| | 2 | 0.91 | 0.93 | 0.86 | 0.01 |
| | 3 | 0.95 | 0.96 | 0.93 | 0.01 |
| Pumsb | 1 | 0.78 | 0.99 | 0.51 | 0.17 |
| | 2 | 0.99 | 1.00 | 0.99 | 0.00 |
| | 3 | 1.00 | 1.00 | 1.00 | 0.00 |
| Mushroom | 1 | 0.38 | 1.00 | 0.10 | 0.23 |
| | 2 | 0.82 | 0.99 | 0.57 | 0.09 |
| | 3 | 0.99 | 1.00 | 0.97 | 0.01 |

well among different computation units w.r.t. the workload balancing. However, we found that such schemes rendered almost all nodes to be depending on the complete dataset. Hence, the complete data should also be communicated to all nodes. Table III shows statistics on the percentage of the data that is required by a node when using Equal Weight assignment. The table shows that for 3-FIs, the average percentage is almost 1, meaning that almost all nodes need the full dataset. This property can be infeasible when dealing with Big Data.

### *B. Execution Time*

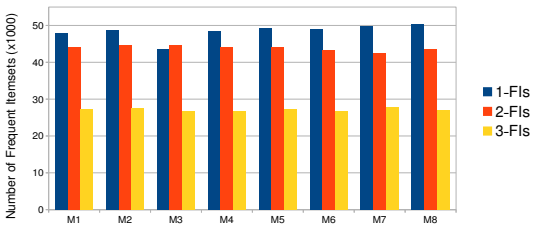
For our runtime experiments we used a scrape of the delicious dataset provided by DAI-Labor [29]. The original dataset contains tagging content between 2004 and 2007. The content is identified by a timestamp, a user, a url and a given tagname. We created a transaction dataset where transactions represent tags. Properties of the dataset is shown in Table I. For the first few experiments we used our local cluster consisting of 2 machines. Each machine containing 32 Intel(R) Xeon(R) processing units and 32GB of RAM. However, we restricted each machine to use up to 6 mappers each. Both machines are running on Ubuntu 12.04 and Hadoop 1.1.2.

We compared running times for our Hadoop implementations of Dist-Eclat and BigFIM and the PFP implementation provided by the Mahout library [3]. Note that PFP is designed as a top-k miner. Therefore, we forced the algorithm to output the same number of patterns found by our implementations to obtain a more fair comparison. The result is shown in Figure 3, the x-axis gives the MinSup threshold for different runs and the y-axis shows the computation time in seconds. As expected, Dist-Eclat is the fastest method and BigFIM is second. However, for lower MinSup values it is an order of magnitude slower. Remember, however, that the goal of BigFIM is to be able to mine enormous databases. More importantly, PFP is always much slower than our methods and while decreasing the minimum support, we observed that PFP either takes an enormous amount of time (we stopped execution after approximately one week) or runs out of memory.

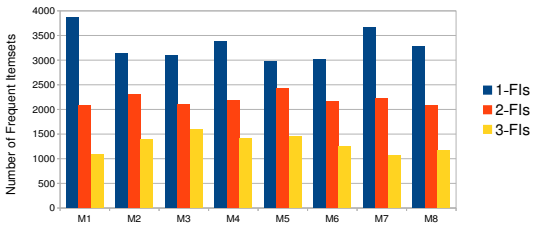
The rest of our experiments are conducted without generating the frequent sets as output. We focus on mining speed rather than on shipping of data.

We analysed the speed of our algorithm using seeds of length 1, 2 and 3. Figure 4 shows timing results when mining the Tag dataset with a MinSup of 15.5K. The x-axis shows the number of mappers used for each run and the y-axis shows the time in seconds. The figure shows that using 2- and 3-FIs give lowest running times and that the two are in fact similar for this dataset in terms of complete running time. Figure 5 gives an in depth view on the time distribution for individual mappers for Tag. The x-axis shows 8 mappers with unique IDs and the y-axis shows the computation time in seconds. We see a clear inbalance for the 1-FIs: mapper M8 finishes 10 times faster than M1. 2-FIs is much better due to a better load balancing, however, the variance is still large. Using 3-FIs yields equally divided mining times.

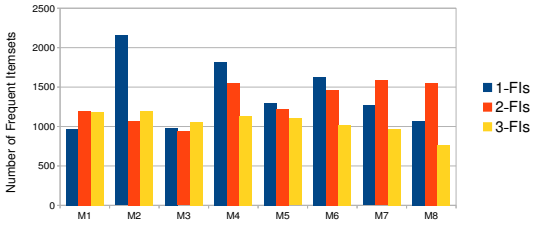
In another experiment we tested the Pumsb dataset with a minimum support of 12K. We used only 3-FIs as seeds, but varied the number of mappers between 1 and 120. Statistics for



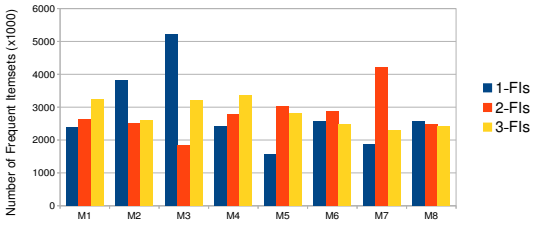

(a) Abstracts




(b) T10I4D100K




(c) Mushroom




(d) Pumsb

Figure 2: Number of Frequent Itemsets generated by partitions

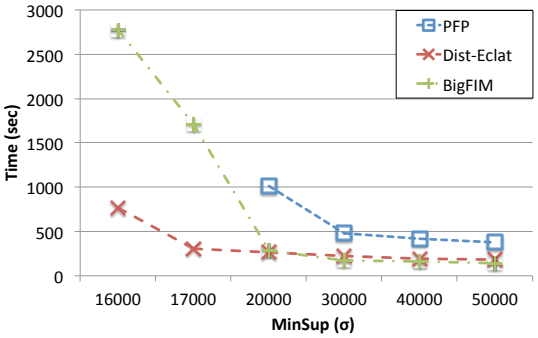



Figure 3: Timing comparison for different methods on Tag

this experiment are shown in Table V. We see that not only the average runtime per node decreases drastically, also the maximum mining time decreases almost linearly. The latter, in the end, influences the running time the most. One peculiarity

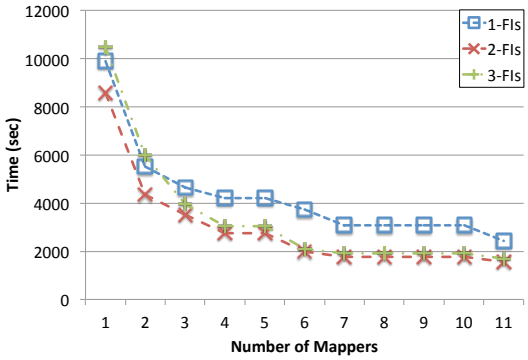



Figure 4: Total execution time on Tag with $\sigma = 15.5K$

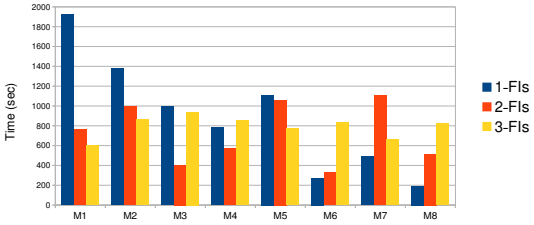



Figure 5: Execution time of mappers for Tag with $\sigma = 15.5K$

Table V: Mining time stats Pumsb with 3-FIs and $\sigma = 12K$

| #Mappers | Avg | Max | Min | StDev | Avg FIs |
|---|---|---|---|---|---|
| 1 | 36,546.00 | 36,546 | 36,546 | 0.00 | 18,885 |
| 5 | 7,775.00 | 8,147 | 7,042 | 400.17 | 3,777 |
| 10 | 3,810.90 | 4,616 | 2,591 | 710.14 | 1,889 |
| 15 | 2,516.47 | 3,726 | 1,733 | 616.26 | 1,259 |
| 20 | 1,912.85 | 3,533 | 875 | 754.05 | 944 |
| 40 | 943.63 | 2,016 | 278 | 456.56 | 472 |
| 60 | 629.80 | 1,579 | 146 | 344.30 | 315 |
| 80 | 472.35 | 1,422 | 69 | 326.37 | 236 |
| 100 | 376.45 | 2,177 | 60 | 313.57 | 189 |
| 120 | 316.15 | 1,542 | 29 | 264.72 | 157 |

Table VI: Mining time stats Tag with 3-FIs and $\sigma = 15K$

| #Mappers | Avg | Max | Min | StDev | Avg FIs |
|---|---|---|---|---|---|
| 20 | 25,851 | 42,900 | 11,160 | 8,724.88 | 844 |
| 40 | 13,596 | 28,620 | 4,320 | 6,439.29 | 422 |
| 60 | 8,508 | 24,240 | 1,440 | 5,195.64 | 281 |
| 80 | 6,548 | 23,760 | 1,320 | 4,066.11 | 211 |

in the result is the sudden increase in time when using 100 mappers. This behaviour is due to coincidence; indeed, some mappers may get many large seeds because of our simple allocation scheme, resulting in an increased runtime. Figure 6 shows the decrease in running time compared to the average number of prefixes per node. The red line in fact shows the perfect scalability behaviour. Its shows that the scalability is finite in the sense that after a while adding only a few nodes is not productive. Because the number of seeds to be divided reaches a saturation point. We also have to remark that during initialisation of the mappers, the data has to be distributed to all of them, resulting in a larger total computation time, in contrast to using less nodes.

At last, we also conducted the previous experiment for Tag on a real cluster, using Amazon Elastic MapReduce [1]. We

[1] http://aws.amazon.com/elasticmapreduce

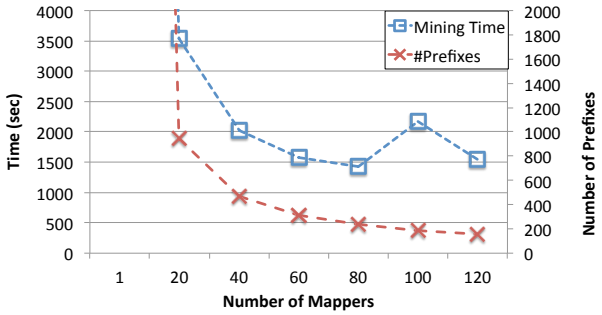



Figure 6: Timing results Pumsb, $\sigma = 12K$

used clusters of sizes between 20 and 80 mappers, using the *m1.xlarge* high memory and high I/O performance instances. Each instance consists of multiple cores performing one map task. The instances are running Red Hat Enterprise Linux and the Amazon Distribution of MapReduce. Table VI shows the statistics for this experiment. We see that the scaling up is less drastic compared to Pumsb. One reason is the blow up of the subtree for some 3-FIs which can not be distributed any longer. The only solution is to increase the length of the prefixes.

## VII. Conclusion

In this paper we studied and implemented two frequent itemset mining algorithms for MapReduce. Dist-Eclat focuses on speed by using a simple load balancing scheme based on $k$-FIs. A second algorithm, BigFIM, focuses on mining very large databases by utilizing a hybrid approach. $k$-FIs are mined by an Apriori variant and then the found frequent itemsets are distributed to the mappers. At the mappers frequent itemsets are mined using Eclat.

We studied several techniques for balancing the load of the $k$-FIs. Our results show that using 3-FIs in combination with even a basic Round-Robin allocation scheme results in a good workload distribution. Progressing further down the prefix tree results in even better workload distributions, however the number of intermediate frequent itemsets blow up.

Assignment methods that will result in a better workload distribution is a further research issue.

At last, the experiments show that our methods outperform state-of-the-art FIM methods on Big Data using MapReduce.

## References


[1] Frequent itemset mining dataset repository. http://fimi.ua.ac.be/data, 2004.

[2] Apache hadoop. http://hadoop.apache.org/, 2013.

[3] Apache mahout. http://mahout.apache.org/, 2013.

[4] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Trans. Knowl. Data Eng.*, pages 962–969, 1996.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. VLDB*, pages 487–499, 1994.

[6] G. A. Andrews. *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley, 2000.

[7] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. *SIGMOD Rec.*, pages 85–93, 1998.

[8] M. Boley and H. Grosskreutz. Approximating the number of frequent sets in dense data. *Knowl. Inf. Syst.*, pages 65–89, 2009.

[9] T. De Bie. An information theoretic framework for data mining. In *Proc. ACM SIGKDD*, pages 564–572, 2011.

[10] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*. USENIX Association, 2004.

[11] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: A runtime for iterative MapReduce. In *Proc. HPDC*, pages 810–818. ACM, 2010.

[12] K. Finley. 5 ways to tell which programming languages are most popular ReadWrite. http://readwrite.com/2012/06/05/5-ways-to-tell-which-programming-lanugages-are-most-popular, 2012.

[13] F. Geerts, B. Goethals, and J. V. D. Bussche. Tight upper bounds on the number of candidate patterns. *ACM Trans. Database Syst.*, pages 333–363, 2005.

[14] A. Ghoting, P. Kambadur, E. Pednault, and R. Kannan. NIMBLE: a toolkit for the implementation of parallel data mining and machine learning algorithms on mapreduce. In *Proc. ACM SIGKDD*, pages 334–342. ACM, 2011.

[15] B. Goethals. Survey on frequent pattern mining. *Univ. of Helsinki*, 2003.

[16] S. Hammoud. *MapReduce Network Enabled Algorithms for Classification Based on Association Rules*. Thesis, 2011.

[17] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. *SIGMOD Rec.*, pages 1–12, 2000.

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

[19] J. Li, Y. Liu, W.-k. Liao, and A. Choudhary. Parallel data mining algorithms for association rules and clustering. In *Intl. Conf. on Management of Data*, 2008.

[20] N. Li, L. Zeng, Q. He, and Z. Shi. Parallel implementation of apriori algorithm based on MapReduce. In *Proc. SNPD*, pages 236–241, 2012.

[21] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh. Apriori-based frequent itemset mining algorithms on MapReduce. In *Proc. ICUIMC*, pages 26–30. ACM, 2012.

[22] M. Malek and H. Kadima. Searching frequent itemsets by clustering data: Towards a parallel approach using mapreduce. In *Proc. WISE 2011 and 2012 Workshops*, pages 251–258. Springer Berlin Heidelberg, 2013.

[23] B. Mobasher, H. Dai, T. Luo, and M. Nakagawa. Effective personalization based on association rule discovery from web usage data. In *Proc. WIDM*, pages 9–15. ACM, 2001.

[24] E. Ozkural, B. Ucar, and C. Aykanat. Parallel frequent item set mining with selective item replication. *IEEE Trans. Parallel Distrib. Syst.*, pages 1632–1640, 2011.

[25] B.-H. Park and H. Kargupta. Distributed data mining: Algorithms, systems, and applications. 2002.

[26] S. Parthasarathy, M. J. Zaki, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory systems. *Knowl. Inf. Syst.*, pages 1–29, 2001.

[27] M. Riondato, J. A. DeBrabant, R. Fonseca, and E. Upfal. PARMA: a parallel randomized algorithm for approximate association rules mining in MapReduce. In *Proc. CIKM*, pages 85–94. ACM, 2012.

[28] A. Savasere, E. Omiecinski, and S. B. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. VLDB*, pages 432–444, 1995.

[29] R. Wetzker, C. Zimmermann, and C. Bauckhage. Analyzing Social Bookmarking Systems: A del.icio.us Cookbook. In *Proc. ECAI MSoDa*, pages 26–30, 2008.

[30] M. Zaki. Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Trans. Knowl. Data Eng.*, pages 1021–1035, 2005.

[31] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel algorithms for discovery of association rules. *Data Min. and Knowl. Disc.*, pages 343–373, 1997.

[32] M. J. Zaki. Parallel and distributed association mining: A survey. *IEEE Concurrency*, pages 14–25, 1999.

[33] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. ACM SIGKDD*, pages 326–335, 2003.

[34] L. Zeng, L. Li, L. Duan, K. Lu, Z. Shi, M. Wang, W. Wu, and P. Luo. Distributed data mining: a survey. *Information Technology and Management*, pages 403–409, 2012.

[35] L. Zhou, Z. Zhong, J. Chang, J. Li, J. Huang, and S. Feng. Balanced parallel FP-Growth with MapReduce. In *Proc. YC-ICT*, pages 243–246, 2010.