Frequent Itemsets Mining for Big Data: an experimental analysis

Daniele Apiletti, Elena Baralis, Tania Cerquitelli, Paolo Garza, Fabio Pulvirenti*

Politecnico di Torino, Dipartimento Automatica e Informatica, Torino, Italy

Abstract

Itemset mining is a well-known exploratory data mining technique used to discover interesting correlations hidden in a data collection. Since it supports different targeted analyses, it is profitability exploited in a wide range of different domains, ranging from network traffic data to medical records. With the increasing amount of generated data, different scalable algorithms have been developed, exploiting the computational advantages of distributed computing frameworks, as Apache Hadoop MapReduce and Apache Spark. However, in most cases no algorithm is universally superior. Several aspects influence which algorithm performs best, including input data cardinality and data distribution, and the algorithm selection is usually manually performed based on analyst expertise.

This paper presents an experimental study to compare the performance of some state-of-the-art implementations of itemset mining algorithms to guide

*Corresponding author

Email addresses: daniele.apiletti@polito.it (Daniele Apiletti), elena.baralis@polito.it (Elena Baralis), tania.cerquitelli@polito.it (Tania Cerquitelli), paolo.garza@polito.it (Paolo Garza), fabio.pulvirenti@polito.it (Fabio Pulvirenti)

the analyst in selecting the most suitable approach based on the outlined lesson learned. Load balancing and communication costs will be included in the analysis dimensions, in order to build a structured comparison through both data mining and distributed environments criteria. Many real and synthetic datasets have been considered in the comparison. Eventually, no algorithm proves to be universally superior and performance is heavily influenced by both data distribution and input parameter setting.
Keywords: Big data, Frequent itemset mining, Hadoop and Spark platforms

---

1. Introduction

In recent years, the increasing capabilities of recent applications to produce and store huge amounts of information has changed dramatically the importance of the intelligent analysis of big data. The interest towards data mining, an important set of techniques useful to extract effective and usable knowledge from data, has risen. This trend is noticeable in both the academic and the industrial domains. For researchers, the big data analytics scenario is very challenging. Often, indeed, the application of traditional data mining techniques to such large volumes of data is not straightforward. It is not only a matter of computational power or memory; some of the most popular techniques had to be redesigned from scratches to fit the new environment. On the other side, companies are interested in the strategic benefits that big data could deliver, even directly. In [1], the authors present a study to illustrate that larger data indeed can be more valuable assets for predictive analytics. The deduction is that institutions with larger collections of data

and, of course, the skill to take advantage of them, can obtain a competitive advantage over institutions without. Data mining, together with machine learning [2], is the main tool on which big data analytics rely on and it includes different types of techniques: (i) clustering algorithms to discover hidden structures in unlabelled data [3], (ii) frequent itemsets mining and association rules analysis to discover interesting correlations and dependencies [4], and finally, (iii) supervised learning techniques to extract a function or a model that best approximates the distribution of the input dataset to map or label new data examples [5]. Existing mining algorithm revealed to be very efficient on typical datasets but very resource intensive when the size of the input dataset grows up. In general, applying data mining techniques to big data collections has often entailed to cope with computational costs that represent a critical bottleneck. Furthermore, the shift towards horizontal scaling in hardware has highlighted the need of distribution/parallelization of data analytics techniques. Distributed and parallel approaches allow sharing the processing of the data into independent tasks to scale mining activities on very large datasets. Different frameworks have been designed and developed to support the parallelization of data mining algorithms (e.g., Hadoop [6], Apache Spark [7], GraphLab [8], Google Pregel [9], Giraph [10]).

Effective and efficient analytics algorithms have been proposed during the last years to better utilize the available hardware resources and distributed computing frameworks. Here we focus on itemset mining algorithms because thery represent exploratory approaches widely used to discover frequently co-occurring items from the data. These algorithms have been widely exploited in different application domains (e.g., network traffic data [11], health-

care [12], biological data [13], energy data [14], images [15], open linked data [16], document and data summarization [17], [18], [19], to support different targeted analyses.

Although different algorithms have been proposed to perform the computationally intensive frequent itemset mining task, also in the distributed frameworks, no algorithm is universally superior. Several aspects influence which algorithm performs best, including input data cardinality and data distribution, adopted strategies to process the data into independent tasks, strategies to reduce the communication costs. The algorithm selection for a given analytics case study is usually manually performed based on analyst expertise and it is very time consuming. To help the analyst in the algorithm selection process, here we present an experimental comparison of different scalable itemset mining algorithms. Specifically, as summarized in Table 2, the contribution of this review includes:

- The discussion of the state-of-the-art itemset mining algorithms dealing with huge data collections to analyze how technological development efficiently support the continuous design of more scalable and more efficient algorithms. We selected the most two widespread and recent distributed frameworks as Hadoop [6], Apache Spark [7] to set the experimental scenario. We selected the five algorithms, released as open source code, to perform the itemset mining discovery on distributed environment. These algorithms (i.e., Mahout PFP [20], Mllib PFP [21], BigFIM [22], DistEclat [22], YAFIM [23]) cover the different search space strategies adopted in the centralized architecure to efficiently address the mining activity by effectively dealing with different data

distribution.

- The definition of four evaluation criteria to characterize both the algorithmic strategies and the distributed implementation as well.

- A detailed comparative analysis of the selected, running on either Spark or Hadoop framework, with a thoroughly discussion on interesting results got by performing a large set of experiments on real and synthetic datasets. Specifically, we run more than 250 experiments on 14 synthetic datasets and 2 real datasets to evaluate the algorithm performance, load balancing and communication cost as well.

- The discussion of the lessons learned to share general advices gained from the experience of performing the in-depth comparative analysis.

- The discussion of some open issues that should be addressed to support a more effective and efficient data mining process on very large datasets.

This paper is organized as follow. Section 2 briefly introduces the Hadoop and Spark frameworks. Section 3 briefly recalls the frequent itemset mining problem, while Section 4 presents the evaluation criteria considered in this study. Section 5 discussed the selected algorithms, while in Section 6 we benchmark the algorithms with a large set of experiments on both real and synthetic datasets. Section 7 summarizes the lessons learned from our evaluation analysis, while Section 8 discusses some research directions to be addressed to support a more effective and efficient data mining process on big data collections. Section 9 provides a brief summary of this revies.

2. Apache Hadoop and Spark

Today's shift towards horizontal scaling in hardware has highlighted the need of distributed algorithms. Thus, we are witnessing the explosion of distributed and parallel approaches, often accompanied with cloud-based services (e.g. Platform-as-a-Service tools) [11]. MapReduce [24] can be considered the most popular approach of the past decade. Designed to cope with very large datasets, Hadoop [6] is the most widely adopted MapReduce implementation. In the last couple of years, instead, Apache Spark [7] has become the favourite platform for large scale data analytics, outperforming Hadoop performance thanks to its distributed memory abstraction. Hadoop and Spark are not the only frameworks supporting the parallelization of Data mining algorithms. GraphLab [8], Google Pregel [9] and its open-source counterpart Giraph [10] are fault-tolerant, graph-based framework while SimSQL [25], for instance, exploits an SQL-based approach. Distributed systems are popular also because they became very easy to use: Message Passing Interface (MPI) [26], one of the most adopted framework in academic environment, works efficiently only on very low level programming such as C.

In this review we analyse and evaluate the most effective approaches developed on top of the Apache Hadoop [6] and Spark [7] frameworks. Thanks to their architecture which allows programmers to focus only on the algorithmic issues, leveraging the high level programming environment, these frameworks represent the current de-facto standard in the Big Data environment. Both of them support the MapReduce paradigm, a distributed programming model introduced by Google [24] to support its data intensive

processing. A MapReduce application consists of two main phases, whose names are map and reduce. The map function is fed with a shard of the input dataset on each node; after the processing it outputs one or more key-value couples. Map results are exchanged among the cluster nodes: this is the optional shuffle phase. Finally, the reduce phase is run for each unique key and iterates through the values that are associated with that key.

Hadoop has become very popular in the last decades; it allows programmers not to concern about inter-process communication and low-level details but to focus on the problem to be solved. The success of Hadoop and MapReduce is due to the paradigm that shifts the computation to the data: thanks to the Hadoop Distributed File System (HDFS), it takes advantages of data locality allowing the nodes to process the data they store. The MapReduce paradigm is designed for batch processing: iterative processes do not fit efficiently since, often, each iteration requires a new reading phase from the disk. This feature is critical when dealing with huge datasets. This issue motivated the improvements introduced by Spark. Apache Spark is a general purpose in-memory distributed platform. It enables machines to cache data and intermediate results in memory, instead of reloading them from the disk at each iteration, through the introduction of Resilient Distributed Datasets (RDD). A RDD is a read-only, partitioned collection of records obtained from another RDD or from HDFS. RDDs can avoid on-disk materialization since their creation process, as a series of transformations (the lineage), is preserved, so that they can be recreated when needed, even if this is an expensive process. Spark supports both graph and streaming processes, overcoming the limitations of the MapReduce batch-oriented paradigm, al-

though maintaining a full compatibility with the latter. Adding flexibility beyond the two-stage model of MapReduce, Spark can provide complex Direct Acyclic Graph (DAG) data flows. Spark also supports different development languages, such as Java, Python and Scala, while Hadoop supports only Java.

2.1. Hadoop and Spark Machine Learning Libraries

In recent years the success of these distributed platforms was supported by the introduction of open source libraries of machine learning algorithms. Mahout [20] for Hadoop has represented one of the most popular collection of Machine Learning algorithms, containing implementations in the areas such as clustering, classification, recommendation systems, etc. All the current implementations are based on Hadoop MapReduce. MADlib [27], instead, provides a SQL toolkit of algorithms that run over Hadoop. Finally, MLLib [21] is the Machine Learning library developed on Spark, and it is rapidly growing up. MLLib allows researchers to exploit Spark special features to implement all those applications that can benefit from them, e.g. fast iterative procedures.

3. Frequent itemset mining

In this section, a basic introduction to frequent itemset mining will be given to the readers. Let $\mathcal{I}$ be a set of items. A transactional dataset $\mathcal{D}$ consists of a set of transactions $\{t_1, \ldots, t_n\}$. Each transaction $t_i \in \mathcal{D}$ is a collection of items (i.e., $t_i \subseteq \mathcal{I}$) and it is identified by a transaction identifier (tid_i). Figure 1a reports an example of a transactional dataset with 5 transactions.

$\mathcal{D}$	
tid	items
1	a,b,c,l,o,s,v
2	a,d,e,h,l,p,r,v
3	a,c,e,h,o,q,t,v
4	a,e,f,h,p,r,v
5	a,b,d,f,g,l,q,s,t

(a) Horizontal representation of $\mathcal{D}$

Vertical Representation	
item	tidlist
a	1,2,3,4,5
b	1,5
c	1,3
d	2,5
e	2,3,4
f	4,5
g	5
h	2,3,4
l	1,2,5
o	1,3
p	2,4
q	3,5
r	2,4
s	1,5
t	3.5
v	1,2,3,4

(b) Vertical representation of $\mathcal{D}$

Figure 1: Running example dataset $\mathcal{D}$

An itemset I is defined as a set of items (i.e., $I \subseteq \mathcal{I}$) and it is characterized by a support value, which is denoted by $sup(I)$ and defined as the ratio between the number of transactions in $\mathcal{D}$ containing I and the total number

of transactions in $\mathcal{D}$. In the example dataset in Figure 1a, for instance, the support of the itemset $\{aco\}$ is 2/5. . This value represents the frequency of occurrence of the itemset in the dataset.

Given a transactional dataset $\mathcal{D}$ and a minimum support threshold $minsup$, the Frequent Itemset Mining [28] problem consists in extracting the complete set of frequent itemsets from $\mathcal{D}$.

Many subsets of frequent itemsets exist. In this paper, we focus on closed itemsets. Closed itemsets [29] are a particular and valuable subset of frequent itemsets, being a concise but complete representation of frequent itemsets. Precisely, an itemset I is closed if none of its supersets (i.e. the set of itemsets which include I) has the same support count as I.

A transactional dataset can also be represented in a vertical format, which is usually a more effective representation for datasets characterised by an average number of items per transaction orders of magnitudes larger than the number of transactions. In this representation, each row consists of an item i and the list of transactions in which it appears, also called $tidlist(\{i\})$. For instance, the tidlist of the itemset $\{aco\}$ in the example dataset $\mathcal{D}$ is $\{1, 3\}$. Figure 1b reports the transposed representation of the running example reported in Figure 1a. The main advantage of the vertical format is the possibility to obtain the tidlist of an itemset just intersecting the tidlists of the included items, without the need of a full scan of the dataset.

4. Evaluation criteria

The main target of this review is to build a structured comparison among the most popular frequent itemset miners in distributed environments. For

this reason, we define a set of criteria which can be divided into two groups, summarized in Table 1.

The first group, named *algorithmic strategy*, is strictly related to the centralized frequent itemset algorithms from which the distributed implementations are derived. Itemset discovery algorithms, proposed for the distributed frameworks, are not designed from scratches to be distributed or parallelized. Often, the main contribution introduced in the domain is the implementation of well-known techniques to distributed environment. Thus, the main research efforts are moved from the algorithm design to the following points:

- The distribution of tools or algorithms that were not designed to be distributed (i.e. splitting the computation load into more than one node). In addition, data mining algorithms are often characterized by the need of a full knowledge of the problem or data. In other words, data mining problems are often not "embarassingly parallelizable". This issue makes the distribution very challenging.

- A well-engineered transposition to distributed frameworks, exploiting the advantages and features of the platforms. For instance, exploiting data locality in MapReduce-based implementations provides a fundamental performance boost. Another example is the optimized "shuffle & sort" phase, which represents the unique phase in which data can be sent to other nodes. Transposing an algorithm into MapReduce can be very challenging because of its limitations, whereas one of the advantages of Apache Spark over Hadoop is a greater flexibility.

Hence, the underlying centralized algorithms are very important to de-

scribe and evaluate the scalable approaches. Some of their features are directly inherited by the distributed algorithms.

Specifically, we have selected two criteria, as reported in Table 1, directly inherited from the underlying centralized approaches, named the candidate itemset generation phase [30].

1. *The search space exploration strategy* allows decomposing the mining task into a set of smaller tasks to dramatically reduce the computation cost. Different strategies have been exploited in performing the itemset mining as divide-and-conquer, depth-first methods or level-wise, breadth-first generation methods. Each strategy can yield good performance when dealing with a given data distribution.
2. *The data distribution.* Each collection of data is characterized by a given distribution varying based on the number of transactions, average transaction length (average number of objects in a given transaction, and the cardinality of different objects. Datasets are usually characterized by an inherent sparseness when a large number of transactions appear with a limited/shorter average transaction length, and a large variety of different objects/items. The sparseness in data distribution increases with data volume and cardinality of different objects. Although a formal and universal definition of data distribution is not yet available in the domain of itemset mining, it is well-known that a given algorithm is typically suited for a given data distribution, thus its performance are the best for some datasets or under some input parameter values and the worst in other cases. In general, the execution cost of a given algorithm tends to increase when dealing with dense datasets

or large data volume with an inherent sparseness but with low support thresholds. In these both conditions, a large number of itemsets have to be generated.

The performance of the algorithms that adopt a level-wise or breadth-first exploration (i.e. algorithms that generate candidate itemsets of length k from itemsets of length $k-1$) is negatively affected by a large average transaction width, because more candidate itemsets must be examined [28]. Since average transaction width is strongly related to the input data distribution, there exists a relationship between the exploration strategy and the input dataset distribution. For example, Apriori-based algorithms [31], detailed in Subsection 5.2, with their breadth-first exploration approach, better fit datasets characterized by sparse distributions, i.e. low correlation among patterns and high item cardinality.

The second set of evaluation criteria is related to the distributed nature of the processing. They are often undervalued in the data mining context but represent critical issues [32], [33].

1. Communication costs: this issue is often underestimated in distributed algorithms, but they represent the most likely bottleneck of a distributed system [34]. In the design phase, most of the researchers focus only on the computational costs and the need to split them among the nodes. The result is that a great amount of data is sent through the network, making communication costs much higher than computational costs.
2. Load balancing: since one of the main goals of a distributed approach is to decrease the overall execution time, load balancing is required

Table 1: Overview of valution criteria.

Class Name	Property of	Criterion name	Domain
Algorithmic strategy	Centralized approaches	The search space exploration strategy	{ Depth First, Breadth First}
		Data distribution	{ dense, sparse }
Distributed processing	Distributed approaches	Communication cost handling	{ Yes, No }
		Load balancing handling	{ Yes, No }

to efficiently reach such objective. An unbalanced load undermines the advantages of a parallel environment: the overall execution time is that of the slowest, most loaded node. In a fully unbalanced environment, the worst case scenario leads to no benefits from parallelization while still incurring all the overheads of coordinating a rather complex distributed system.

A review of the evaluation criteria is presented in Table 3. After a qualitative review of the algorithms in Section 5, in Section 6 an experimental performance evaluation is provided.

5. Selected algorithms

This section describes the algorithms representing the state of the art in frequent itemset mining, summarized in Table 3: FP-growth [38], BigFIM and DistEclat [22], and YAFIM [23]. They have been selected based on popularity and distributed implementation availability for Apache Hadoop and/or Spark, to allow a real experimental comparison. The only algorithm which is missing a publicly available implementation is YAFIM.

Table 2: Resume of the theoretical and experimental analysis

Distributed frameworks	Apache Spark [7]	**Datasets**	**14 Synthetic datasets**
	Apache Hadoop [6]		Datasets generated through IBM generator [35]
Frequent itemset mining algorithms	Mahout PFP [20]		**2 Real datasets**
	Mllib PFP [21]		URL tagging of the Delicious dataset [36]
	BigFIM [22]		Network Traffic Traces collected through TSTAT [37]
	DistEclat [22]		
	YAFIM [23]		
Evaluation criteria	Search space exploration strategy	**Addressed issues in the evaluation comparison**	Performance
	Data distribution		Load Balancing
	Communication cost		Communication cost
	Load Balancing		

5.1. FP-Growth based algorithms

FP-growth [38] is among the most popular approaches for frequent pattern mining (FP stands for frequent pattern). It is based on a transposition of the whole dataset into a main memory compressed representation of the database called FP-tree. The algorithm is based on a recursive visit of the tree with a divide and conquer, partitioning-based approach. In the first phase the support of the items is counted to build the "header table". Then, the FP-tree is built exploiting the header table and the input dataset: each transaction is included adding or extending a path on the tree, exploiting common prefixes. Finally, for each item, it extracts the frequent itemsets from the items conditional FP-tree, in a recursive, depth first fashion. For the nature of the FP-tree, the data distribution which best fits FP-Growth is

Table 3: Algorithm comparison summary.

Name	**Framework**	**Underlying algorithm**	**Data distrib.**	**Search Strategy**	**Comm. cost handling**	**Load balance handling**
Mahout PFP	Hadoop	FP-Growth	dense	Depth First	Yes	No
MLlib PFP	Spark	FP-Growth	dense	Depth First	Yes	No
Dist-Eclat	Hadoop	Eclat	dense	Depth First	Yes (tradeoff with) load balancing)	Yes
BigFIM	Hadoop Hadoop	Apriori and Eclat	dense and sparse	Breadth First and Depth First	Yes (tradeoff with load balancing)	Yes
YAFIM	Spark	Apriori	sparse	Breadth First	Yes	No

dense. With a sparse dataset, the benefits of the FP-tree transposition would be reduced because there would be a higher number of branches and paths [28] (i.e. a large number of subproblems to generate and results to merge).

Parallel FP-growth [39] is a distributed FP-growth implementation which exploits the MapReduce paradigm to extract the k most frequent closed itemsets. It is included in the Mahout machine learning Library (version 0.9) and it is developed on Apache Hadoop. The main idea behind the distribution is to build independent FP-trees that can be processed separately over different nodes, splitting data-intensive mining tasks into independent subtasks. The algorithm consists of 3 MapReduce jobs: the first is the construction of the Header Table in a MapReduce "Word Count" manner. In the second job, the transactions are transformed into a group dependent set of transactions and distributed among the nodes: in this way, each node builds its independent FP-tree and extracts the frequent itemsets. Finally, the last MapReduce job consists of grouping and merging the top k frequent itemsets found.

The independent FP-trees can have different characteristics and this factor has a significant impact on the execution time of the mining tasks. When the FP-Trees have different sizes, the tasks are unbalanced and hence the whole mining process is unbalanced. This problem could be potentially solved by splitting complex trees in sub-trees: however, defining a metric to split a tree is not easy. The paper takes into account communication cost even if, in the worst case, they can be very high: the shards of the datasets that are sent to the nodes overlap significantly, depending on the dataset characteristics.

Spark PFP [21] represents a pure transposition of FP-growth to Spark; it is included in MLlib, the Spark machine learning library. The algorithm implementation in Spark is very close to the Hadoop sibling, i.e., it first builds independent FP-trees and then invokes the mining step on each tree (one independent task for each FP-tree). It is characterized by dynamic and smooth handling of the different stages of the algorithm, without a strict division in phases. Its main advantage over the Hadoop sibling is the low I/O cost, potentially leading to a single read of the dataset from disk, by loading the transactions in an RDD and processing the data in main memory, whereas the Hadoop-based implementation of PFP performs many more I/O operations.

Both the implementations, being strongly inspired from FP-growth, keeps from the underlying algorithm the features related to the search space exploration (depth-first) and to the data distribution (dense).

5.2. BigFIM and DistEclat

BigFIM and DistEclat [22] are two Hadoop-based frequent itemsets algorithms inspired, instead, from Apriori and Eclat algorithms respectively.

Apriori [31] is a very popular technique. It uses a bottom up approach in which frequent itemset are extended on item at a time (candidate generation) in a level-wise, breadth-first fashion, and groups of candidates are tested against the dataset. The search space is reduced through the downward-closure property, which guarantees that all the supersets of an infrequent itemset are infrequent too. Hence, each iteration consists of two steps: candidate generation and support count. The algorithm ends when no further frequent extension are found.

The data distribution which better fits Apriori is sparse. In fact, with dense datasets the average transaction width can be very large, affecting the algorithms complexity. In this case, the candidates length starts to increase, increasing the number of candidates that should be generated, stored in main memory and tested. In general, Apriori is very efficient at the first steps, when the candidates are not long, but starts to be computationally intensive as soon as long candidates have to be kept in memory.

The Eclat [40] algorithm performs the mining from a vertical transposition of the dataset: in this format, each transaction includes an item and the transaction identifiers (tid) in which it appears ($tidlist$). After the initial dataset transposition, the search space is explored in a depth-first manner similar to FP-growth. The algorithm is based on equivalence classes (groups of itemsets sharing a common prefix), which are smartly merged to obtain all the candidates. Prefix-based equivalence classes are mined independently, in a "divide and conquer' strategy', still taking advantage of downward closure property, even if the depth-first fashion of tree expansion reduces the pruning benefits. The support of a $(k+1)$-candidate is obtained intersecting the

tidlists of the k-itemsets from which it has been obtained, with no need of rescanning the whole dataset.

Eclat better fits dense datasets: the depth-first search strategy may require more infrequent itemsets generated and tested than, for instance, Apriori does. As a result, Eclat efficiency reduces for sparse data with short patterns where most itemsets are infrequent [41].

DistEclat is a frequent itemset miner developed on Apache Hadoop. It exploits Eclat algorithm to extract a superset of closed itemsets. The algorithm mainly consists of two steps: the first aims at finding k-sized prefixes on which, in the second step, the algorithm builds independent subtrees. Even in this case, the main idea is to mine these indepedent prefix trees in different nodes. The algorithm is organized in 3 phases. In the first one, a MapReduce job transposes the dataset into a vertical representation. In the second MapReduce job, k-sized prefixes are obtained from the 1-item prefixes. In the last phase, finally, each node compute independent prefix trees from a set of prefixes. DistEclat is designed to be very fast but, increasing the length of the prefixes, it assumes that the whole initial dataset (transposed in a vertical format) should be stored in the nodes main memory. Specifically, in the worst case, one mapper needs the complete dataset to build all the 2-prefixes [22]. The algorithm inherits from the centralized version the depth-first strategy to explore the search space and the preference for dense datasets.

The BigFIM implementation is very similar to DistEclat. The only difference lies in the prefix extraction phase, where BigFIM exploits the Apriori algorithm: BigFIMs structure makes it more scalable when dealing with very

large datasets. Even if it is slower than DistEclat, which is focused on speed, BigFIM is designed to run on larger datasets, where DistEclat runs out of memory. The reason is related to the first phase in which, exploiting the Apriori strategy, the k-prefixes are extracted in a breadth-first fashion. Consequently, the nodes do not have to keep large transaction lists in memory but only itemsets to be counted. One of the most critical issues of the application of Apriori to large datasets is that, depending on their density, the set of candidates may not fit in main memory. This does not happen for lower values of prefix length (in [22] the authors experimented with a prefix length up to 3). DistEclat, instead, in the worst case is limited by the need of communicating and storing the whole dataset in each node. Finally, because of the differences in the extraction technique used in each phase, in the first one BigFIM achieves the best performance with sparse datasets, while in the second phase it better fits dense ones: overall it does not show a data-distribution preference.

As reported in Table 3, from an analytical point of view, DistEclat and BigFIM are the only algorithms for which an evaluation of the communication costs and load balancing is presented in their respective experimental sections. In particular, the choice of the length of the prefixes generated during the first step affects both communication costs and load balancing. The former would benefit from shorter prefixes while the latter would improve with a deeper level of the mining phase before the redistribution of the prefixes. Hence, depending on the data distribution and the characteristics of the Hadoop cluster, DistEclat and BigFIM can be tuned to optimize communication costs or load balancing.

5.3. YAFIM

YAFIM [23] is an Apriori distributed implementation developed in Spark. Apriori works best with sparse datasets and it is characterized by a different behavior with respect to Eclat and FP-growth: the iterative nature of the algorithm has always represented a challenge for its application in MapReduce-based big data frameworks. The reasons are the overhead caused by the launch of new MapReduce jobs and the requirement to read the input dataset from the disk at each iteration. YAFIM exploits Spark RDDs to cope with these issues. Precisely, it assumes that all the dataset can be loaded into RDDs in order to speed up the counting operations. Hence, after the first phase in which all the transactions are loaded, the algorithm starts the iterative merging and pruning, organizing the candidates in a hash tree to speed up the search. Being strongly Apriori-based, it inherits the breadth-first strategy to explore the search space and the preference towards sparse data distributions. YAFIM exploits the Spark "broadcast variables abstraction" feature, which allows programmers to send subsets of transactional data to each slave only once, rather than with every job that uses those subset of data. This implementation mitigates communication costs (reducing the inter job communication), while load balancing is not addressed.

6. Experimental evaluation

In this section, the results of the experimental comparison are presented. The purpose of the experiments is to compare the behaviours of the algorithms and their distributed implementations, by considering different data distributions and use cases, and to highlight pros and cons of each algorithm.

Specifically, experiments analyze the algorithms in terms of (i) efficiency (i.e., execution time) under different conditions (Sections 6.1-6.4), (ii) load balancing (Section 6.5), and (iii) communication costs (Section 6.6). Finally, Section 7 presents a summary of the "lessons learned".

We evaluated the implementations of four of the algorithms described in Section 5. Specifically, we considered the Parallel FP-Growth implementations of Mahout 0.9 (called Mahout PFP) [20] and MLlib for Spark 1.3.0 (called MLlib PFP) [21], the June 2015 implementation of BigFIM [42], and the version of DistEclat downloaded from [42] on September 2015. Unfortunately, the implementation of the YAFIM algorithm is not available. Therefore, YAFIM is not included in this experimental comparison.

We recall that Mahout PFP extracts the top k frequent closed itemsets, BigFIM and DistEclat extract a superset of the frequent closed itemsets, while MLlib PFP extracts all the frequent itemsets. To perform a fair comparison, Mahout PFP is forced to output all the closed itemsets. Since the extraction of the complete set of frequent itemsets is usually more resource intensive than dealing with only the set of frequent closed itemsets[1], the execution times of Mahout PFP, BigFIM and DistEclat may increase with respect to MLlib PFP. In our experiments, we took care to verify that the numbers of frequent itemsets and closed itemsets are the same. Therefore, the disadvantages related to the more intensive task performed by MLlib are mitigated.

We defined a common set of default parameter values for all experiments,

[1]We recall that the complete set of frequent itemsets can be obtained expanding and combining the closed itemsets by means of a post-processing step.

different settings are explicitly indicated. The default setting of each algorithm was chosen by taking into consideration the physical characteristics of the Hadoop cluster, to allow each approach to exploit the hardware and software configuration at its best. The following default configuration settings for the four algorithms have been considered. For Mahout PFP, the default value of k is set to the lowest value forcing Mahout PFP to mine all frequent closed itemsets for each dataset and *minsup*, while for MLlib PFP the number of partitions is set to 6,000. This value has shown to be the best tradeoff among performance and the capacity to complete the task without memory issues. In particular, with lower values the increase in performance is limited whereas some algorithms cannot scale to very long transactions or very low *minsup*. Higher values, instead, do not lead to better scalability, while affecting performance. Finally, the default value of the prefix parameter of both BigFIM and DistEclat is set to 2, as the result of the following strategy: with a value of 1, BigFIM and DistEclat become too similar, since their only difference is in the first phase of the mining process. On the contrary, with prefix lengths higher than 2, many experiments complete the extraction in the first phase, without execution of the second phase.

We did not define a default value of *minsup*, which is a common parameter of all algorithms, because it is highly related to the data distribution and the use case, so this value is discussed in each set of experiments.

We considered both synthetic and real datasets. The synthetic ones have been generated by means of the IBM dataset generator [35], a very well known generator that is commonly used for performance benchmarking in the itemset mining context. We tuned the following parameters of the IBM

dataset generator to analyze the impact of different data distributions on the performance of the mining algorithms: T = average size of transactions, P = average length of maximal patterns, I = number of different items, C = correlation grade among patterns, and D = number of transactions). The full list of synthetic datasets is reported in Table 4, where the name of each dataset consists of pairs <parameter,value>. Finally, the two real datasets have been used to simulate real life use cases, they are described in Section 6.4.

All the experiments were performed on a cluster of 5 nodes running the Cloudera Distribution of Apache Hadoop (CDH5.3.1) [43]. Each cluster node is a 2.67 GHz six-core Intel(R) Xeon(R) X5650 machine with 32 Gigabytes of main memory, SATA 7200-rpm hard disks, and running Ubuntu 12.04 server with the 3.5.0-23-generic kernel. The dimension of Yarn containers is set to 6 GB. This value leads to a full exploitation of the resources of our hardware, representing a good tradeoff between the amount of memory assigned to each task and the level of parallelism. Lower values would have increased the level of parallelism at the expense of the task completion, whereas higher values would have affected the parallelism, with very few distributed tasks.

6.1. Impact of the minsup support threshold

The minimum support threshold (*minsup*) has a high impact on the complexity of the itemset mining task. Specifically, the lower the value of *minsup*, the higher the execution time of the algorithm. In this section, we analyze how this parameter impacts on the execution time of the each algorithm and implementation.

To avoid the bias due to a specific data distribution, experiments have

Table 4: Synthetic datasets

ID	Name/IBM Generator parameter setting	Num. of different items	Avg. # items per transaction	Size (GB)
1	**T10**-P5-I100k-C0.25-**D10M**	18001	10.2	0.5
2	**T20**-P5-I100k-C0.25-D10M	18011	19.9	1.2
3	**T30**-P5-I100k-C0.25-D10M	18011	29.9	1.8
4	**T40**-P5-I100k-C0.25-D10M	18010	39.9	2.4
5	**T50**-P5-I100k-C0.25-D10M	18014	49.9	3.0
6	**T60**-P5-I100k-C0.25-D10M	18010	59.9	3.5
7	**T70**-P5-I100k-C0.25-D10M	18016	69.9	4.1
8	**T80**-P5-I100k-C0.25-D10M	18012	79.9	4.7
9	**T90**-P5-I100k-C0.25-D10M	18014	89.9	5.3
10	**T100**-P5-I100k-C0.25-D10M	18015	99.9	5.9
11	T10-P5-I100k-C0.25-**D50M**	18015	10.2	3.0
12	T10-P5-I100k-C0.25-**D100M**	18016	10.2	6.0
13	T10-P5-I100k-C0.25-**D500M**	18017	10.2	30.4
14	T10-P5-I100k-C0.25-**D1000M**	18017	10.2	60.9

been executed on two different datasets: Dataset #1 and Dataset #3. They both share the same average length of maximal patterns (5), the number of different items (100 thousands), the correlation grade among patterns (0.25), and the number of transactions (10 millions). The difference is in the average transaction length: 10 items for Dataset #1 and 30 items for Dataset #3 (see Table 4). Being constant the rest of the characteristics, longer transactions lead to a higher dataset density, which results into a larger number of frequent itemsets. Furthermore, we can test the performance degradation when dealing with high-dimensional datasets.

Figure 2 reports the execution time of the itemset mining algorithms when varying the *minsup* threshold from 0.002% to 0.4% and considering

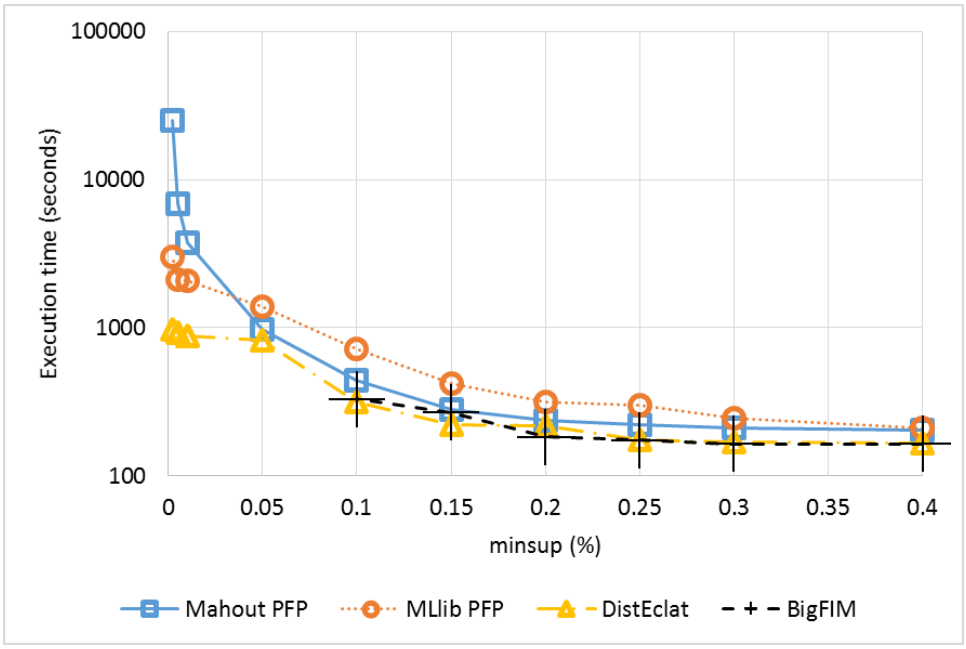

Figure 2: Execution time for different *minsup* values (Dataset #1), average transaction length 10.

Dataset #1. DistEclat is the fastest algorithm for all the considered *minsup* values. However, the improvement with respect to the other algorithms depends on the value of *minsup*. When *minsup* is greater than or equal to 0.2%, all the implementations show similar performance, with BigFIM and DistEclat being slightly faster than Mahout PFP and MLlib PFP. The performance gap largely increases with *minsup* values lower than 0.05%: Mahout PFP becomes orders of magnitude slower than both DistEclat and MLlib PFP. BigFIM is as fast as DistEclat when *minsup* is higher than 0.1%, but below this threshold BigFIM runs out of memory during the k-Frequent Itemset generation phase, specifically when generating 2-FIs. MLlib PFP is generally slower than Mahout PFP, becoming faster only for *minsup* values below

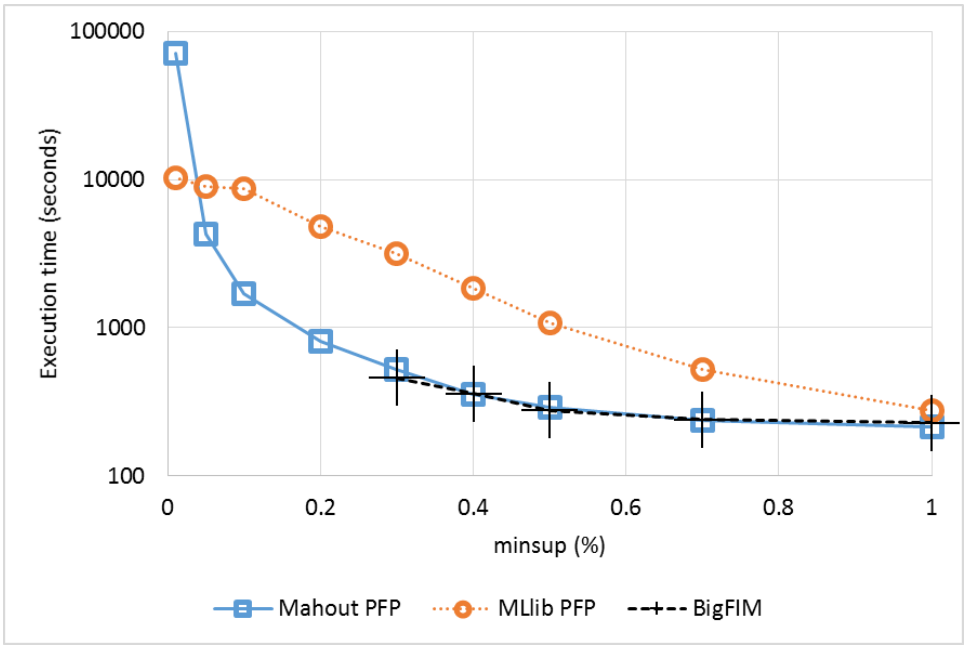

Figure 3: Execution time for different *minsup* values (Dataset #3), average transaction length 30.

0.05%. Based on this first set of experiments, DistEclat seems to be the most appropriate choice, with excellent results when very low values of *minsup* are considered, followed by MLlib PFP, which can reach very low *minsup* values with good performance, at the expense of slightly slower performance for higher *minsup* values.

In the second set of experiments, we analyze the execution time of the algorithms for different minimum support values on Dataset #3, which is characterised by a higher average transaction length (3 times longer than Dataset #1), and a larger data size on disk (4 times bigger), with the same number of transactions (10 millions). Since the mining task is more computationally intensive, *minsup* values lower than 0.01% were not considered in

this set of experiments, as this has proven to be a limit for most algorithms due to memory exhaustion or too long experimental duration (days). Results are reported in Figure 3: DistEclat runs immediately out of memory because of the size of the input dataset, which is transposed in a vertical format during the first phase: the longer transactions prevent it from completing such step. MLlib PFP is much slower than Mahout PFP for most *minsup* values (0.7% and below), and BigFIM, as in the previous experiment, achieves top-level performance, but cannot scale to low *minsup* values (the lowest is 0.3%), due to memory constraints during the k-FI generation phase. Mahout PFP and MLlib PFP are the only algorithms to complete the mining tasks for all *minsup* values. Precisely, Mahout PFP is the most suitable technique when dealing with minsup values over 0.05%, with a perfomance 6 to 8 times faster than the MLlib sibling. However, with lower *minsup* values, Spark MLlib becomes the fastest approach with an order of magnitude gap. We identified the cause of the different performance between the two PFP implementations in the different pruning strategies. The algorithms which extract closed itemsets, such as Mahout PFP, can apply more effective pruning techniques that are not applicable when all frequent itemsets must be extracted, which is the case for MLlib PFP.

Overall, DistEclat is the fastest approach when it does not run out of memory. Mahout PFP is the most reliable implementation across almost all *minsup* values, even if it is not always the fastest, sometimes with large gaps behind the top performers. MLlib is a reasonable tradeoff choice, as it is constantly able to complete all the tasks in a reasonable time. Finally, BigFIM does not present advantages over the other approaches, being unable

to reach low *minsup* values and to provide fast executions.

6.2. Impact of the average transaction length

This section compares the execution times of the selected approaches on datasets with different average transaction lengths, from 10 to 100 items per transaction, and fixed values of 10 million transactions, and 1% *minsup*. To this aim, Datasets #1–10 were used (see Table 4). Longer transactions often lead to more dense datasets and a larger number of longer frequent itemsets. This generally corresponds to more computationally intensive tasks. The execution times obtained are reported in Figure 4. BigFIM and DistEclat execution times for transaction length of 10 and 20 are not reported because, for these configurations, not enough 3-itemsets are extracted. For higher transaction lengths, DistEclat is not included since it runs out of memory for values beyond 20 items per transaction. The other algorithms have similar execution times for short transactions, up to 30 items. For longer transactions, a clear trend is shown: (i) MLlib PFP is much slower than the others and it is not able to scale for longer transactions, as its execution times abruptly increase until it runs out of memory beyond 60 items per transaction; (ii) Mahout PFP and BigFIM have a similar trend until 70 items per transactions, when Mahout PFP becomes slower than BigFIM. Despite the Apriori-based initial phase, BigFIM proved to be the best scaling approach for very long transactions. The FP-growth based approaches, instead, are affected by the increasing length of the transactions.

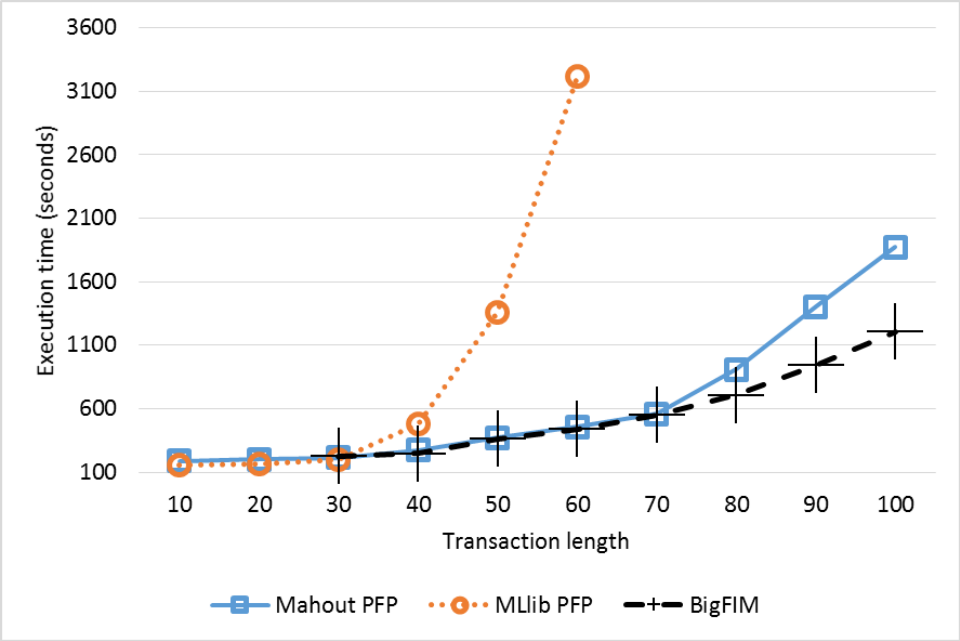

Figure 4: Execution time with different average transaction lengths (Datasets #1–10, *minsup* 1%).

6.3. Impact of the number of transactions

This section evaluates the effect of varying the number of transactions, i.e., increasing the dataset size without changing intrinsic data characteristics, such as the transaction length or the data distribution. To this aim, Datasets #1, #11–14 have been used (see Table 4), which have a number of transactions ranging from 10 millions to 1 billion. The average transaction length is fixed to 10, and the *minsup* is 0.4%, which is the highest value for which the mining leverages both the two phases of BigFIM and DistEclat, and it corresponds to the highest value used in the experiments of Section 6.1.

As shown in Figure 5, all the considered algorithms scale linearly with respect to the dataset cardinality, with BigFIM being the slowest, closely

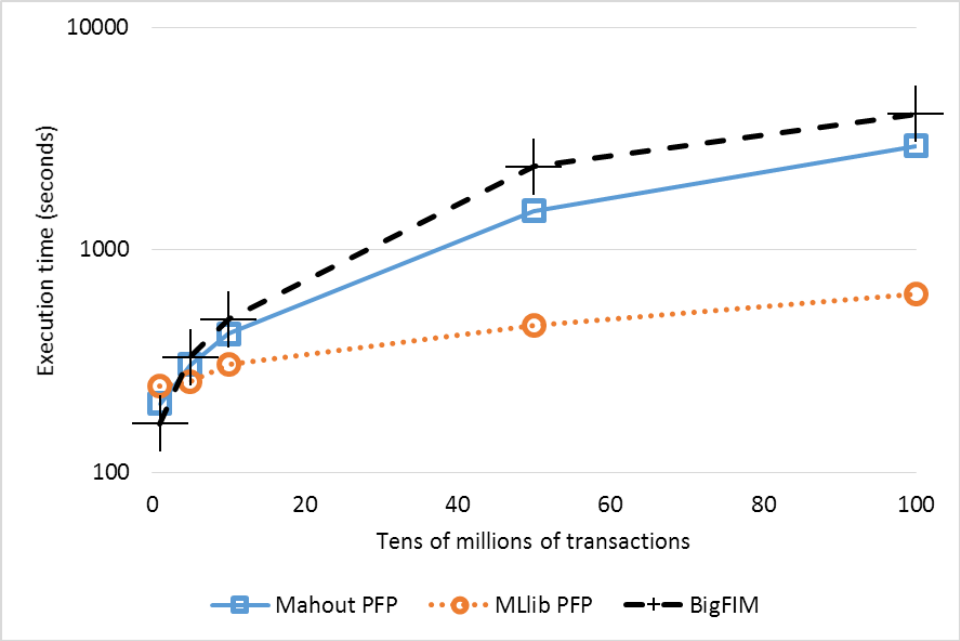

Figure 5: Execution time with different numbers of transactions (Datasets #1, #11–14, *minsup* 0.4%, average transaction length 10).

followed by Mahout PFP, and with MLlib PFP being by far the fastest approach, with execution times reduced by almost an order of magnitude. BigFIM is paying the iterative disk reading activities during its initial Apriori phase. PFP implementations, instead, read from the disk only twice. Finally, DistEclat fails under its assumption that the entire dataset should be stored in each node, and it is not able to complete the extraction beyond 10 million transactions.

6.4. Real use cases

In the following, we analyze the performance of the mining algorithms in two real-life scenarios: (i) URL tagging of the Delicious dataset and (ii)

network traffic flow analysis. The characteristics of the two datasets are reported in Table 5.

Table 5: Real-life use-cases dataset characteristics

ID	Name	Num. of different items	Avg. # items per transaction	Size (GB)
15	Delicious	57,372,977	4	44.5
16	Netlogs	160,941,600	15	0.61

6.4.1. URL tagging

In this experiment, we evaluate the selected algorithms in a real-life use case by using the Delicious dataset [36], which is a collection of web tags. Each record represents the tag assigned by a user to a URL and it consists of 4 attributes: date, user id (anonymized), tagged URL, and tag value. The transactional representation of the Delicious dataset includes one transaction for each record, where each transaction is a set of four pairs (attribute, value), i.e., one pair for each attribute. The dataset stores more than 3 years of web tags. The dataset is very sparse because it has a huge number of different URLs and tags. Additional characteristics of the dataset are reported in Table 6.

This experiment simulates the environment of a service provider that periodically analyses the web tag data to extract frequent patterns: they represent the most frequent correlations among tags, URLs, users, and dates. Many different use cases can fit this description: tag prediction, topic classification, trend evolution, etc. Their evolution over time is also interesting. To this aim, the frequent itemset extraction has been executed cumulatively

on temporally adjacent subsets of data, whose length is a quarter of year (i.e., first quarter, then first and second quarter, then first, second, and third quarter, and so on, as if the data were being colleted quarterly and analyzed as a whole at the end of each quarter). The setting of *minsup* in a realistic use-case proved to be a critical choice. Too low values lead to millions of itemsets, which become useless as they exceed the human capacity to understand the results. However, too high *minsup* values would discard longer itemsets, which are more meaningful as they better highlight more complex correlations among the different attributes and values. Furthermore, we wanted to obtain at least 3-itemsets to assess the BigFIM two phases, without reducing it to a pure DistEclat. As a result of these constraints and of the high sparsity of the dataset, we identified the setting *minsup*=0.01% as the best tradeoff.

Table 6: Delicious dataset: cumulative number of transactions and frequent itemsets with *minsup* 0.01%.

Up to year, month, quarter	**Number of transactions**	**Number of frequent itemsets**
2003 Dec, Q4	153,375	7197
2004 Mar, Q1	489,556	6013
2004 Jun, Q2	977,515	5268
2004 Sep, Q3	2,021,261	5084
2004 Dec, Q4	4,349,209	4714
2005 Mar, Q1	9,110,195	4099
2005 Jun, Q2	15,388,516	3766
2005 Sep, Q3	24,974,689	3402
2005 Dec, Q4	41,949,956	3090

Table 6 reports the cumulative number of transactions for the different

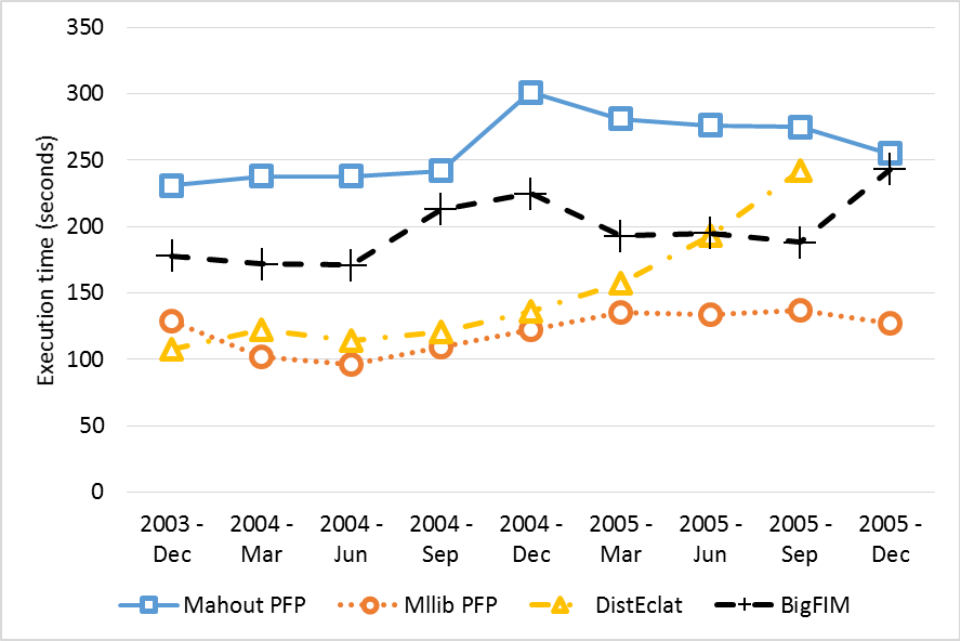

Figure 6: Execution time for different periods of time on the Delicious dataset (*minsup*=0.01%)

periods of time (i.e., the cardinality of the input dataset) and the number of frequent itemsets extracted with a fixed *minsup* of 0.01%, while the execution times of the different algorithms are shown in Figure 6.

MLlib consistently proves to be the fastest approach, with DistEclat following. However, while DistEclat is slightly faster than MLlib only with the first, smallest dataset (up to Dec 2003, with 150 thousands transactions), when the dataset size increases, DistEclat execution times do not scale, and it eventually fails for the final 40-million-transaction dataset of Dec 2005, due to memory exhaustion. BigFIM and Mahout PFP consistently provide 2 to 3 times longer execution times. Apart from DistEclat, all algorithms complete the task with similar performance despite increasing the dataset cardinality

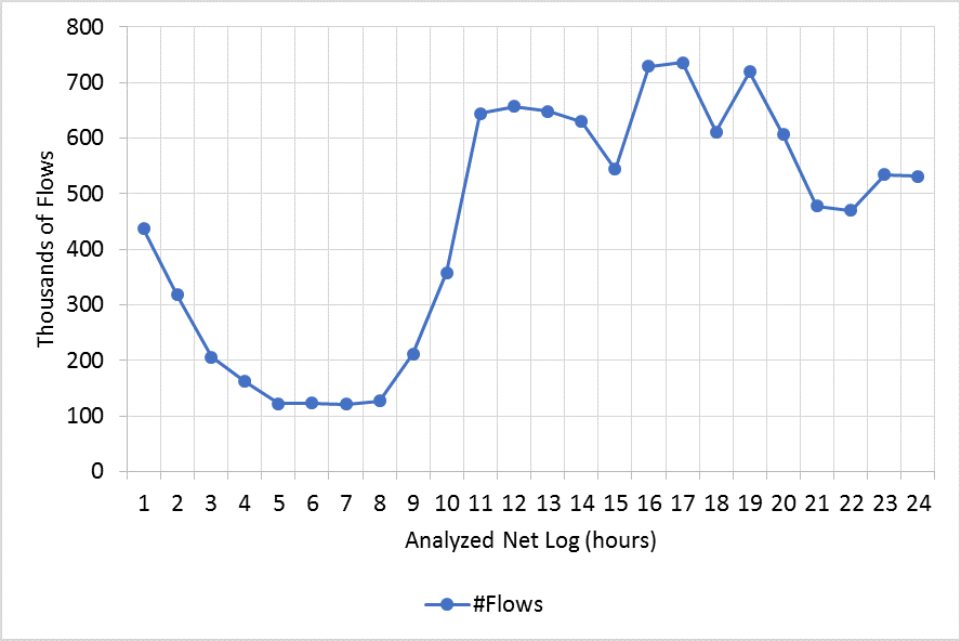

Figure 7: Number of flows for each hour of the day.

from 150 thousand transactions to 41 millions, thanks to the constant relative *minsup* threshold which reduces the number of frequent itemsets for decreasing density of the dataset. To sum up, MLlib is the best choice for short transactions (this dataset length is 4), independently of the dataset cardinality and density.

6.4.2. Network traffic flows

This use case evaluates the approaches in a network environment by using a network traffic log dataset, where each transaction represents a TCP flow. A network flow is a bidirectional communication between a client and a server. The dataset has been gathered through Tstat [37, 44], a popular internet traffic sniffer broadly used in literature [45, 11], by performing a one

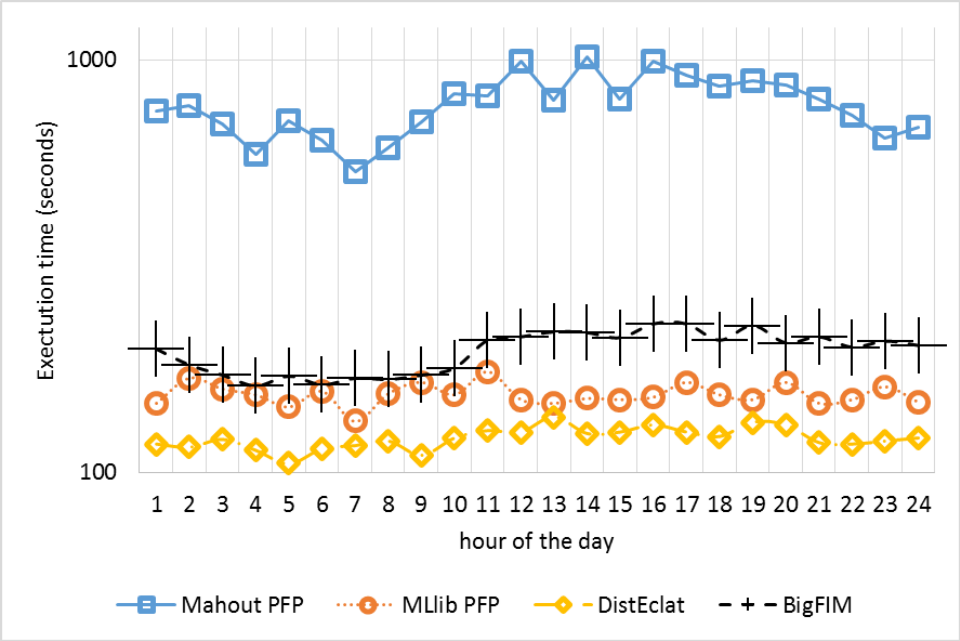

Figure 8: Execution time of different hours of the day. (dataset 31, $minsup$=1%)

day capture in three different vantage points of a nation-wide Internet Service Provider in Italy. Each transaction of the dataset is associated with a flow and consists of pairs $(flow\ feature, value)$. These features can be categorical (e.g., TCP Port, Window Scale) or numerical (e.g., RTT, Number of packets, Number of bytes). We applied a proper preprocessing step in which numerical attributes have been discretized by using the same approach adopted in [11]. Finally, we have divided the set of flows (i.e., the set of transactions) in 1-hour slots, generating 24 sub-datasets. The number of flows in each sub-dataset is reported in Figure 7.

In this use case, the network administrator is interested in performing hourly analysis to shape the hourly network traffic. Hence, we evaluated the performance of the four algorithms, comparing their execution time, on

Table 7: Network traffic flows: number of transactions and frequent itemsets with *minsup* 0.1%.

Hour of the day	**Number of transactions**	**Number of frequent itemsets**
0.00	437,417	166,217
1.00	318,289	173,960
2.00	205,930	163,266
3.00	162,593	166,344
4.00	122,102	157,069
5.00	123,683	164,493
6.00	121,346	170,129
7.00	127,056	159,921
8.00	211,641	169,751
9.00	357,838	187,912
10.00	644,408	191,867
11.00	656,965	183,021
12.00	648,206	184,279
13.00	630,434	180,384
14.00	544,572	175,252
15.00	729,518	192,992
16.00	735,850	189,160
17.00	611,582	177,808
18.00	719,537	179,228
19.00	607,043	174,783
20.00	477,760	161,153
21.00	470,291	159,065
22.00	534,103	144,212
23.00	531,276	164,516

the 24 hourly sub-datasets. For all the 24 experiments *minsup* was set to 1%, which was the tradeoff value allowing all the algorithms to complete the extraction, leveraging both the mining phases of BigFIM and DistEclat, and extracting a reasonable number of frequent itemsets (details in Table 7).

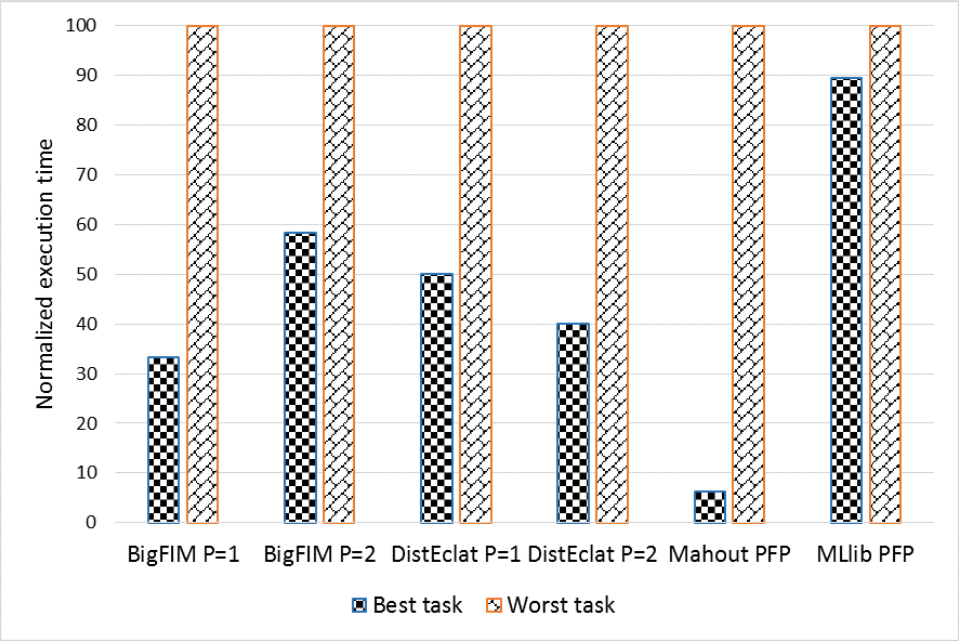

Figure 9: Normalized execution time of the most unbalanced tasks.

Results are reported in Figure 8, where the performance of the different approaches show a clear trend: DistEclat always achieves the lowest execution time, followed by MLlib PFP and BigFIM. Mahout PFP is the slowest. The execution time is almost independent of the dataset cardinality, as it slightly changes throughout the day. The low dataset size (less than 1 Gigabyte overall) and cardinality (less than 1 million transactions) make this the ideal use case for DistEclat, which strongly exploits in-memory computation.

6.5. Load balancing

In this section, experiments address the evaluation of the load balancing strategies of the different approaches, an often underestimated issue in the distributed data mining bibliography (see Section 4).

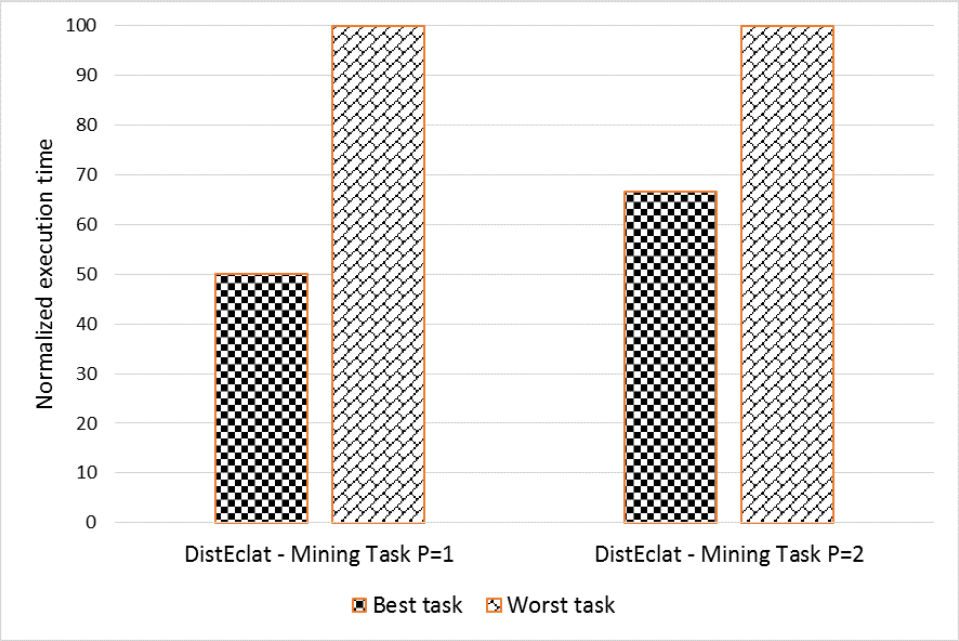

Figure 10: Normalized execution time of the most unbalanced tasks of DistEclat.

Experiments measure the load balancing on a 1-hour-long subset of the network log dataset (Table 5) with a fixed *minsup* of 1%. We consider the most unbalanced jobs of each algorithm and compare the execution times of the fastest and the slowest tasks. To this aim, we are not interested in the absolute execution time, but rather in the normalized execution times, where the slowest task is assigned a value of 100, and the fastest task is compared to such value, as reported in Figure 9.

MLlib PFP achieves the best load balancing, with comparable execution times for all tasks throughout all nodes, whose difference is in the order of 10%. Mahout PFP, instead, shows the worst load balancing issues, with differences as high as 90%.

We included BigFIM and DistEclat with 2 different first-phase prefix sizes. For BigFIM, the experiment confirms that a configuration with longer prefixes leads to a more balanced mining tasks than a configuration with short-sized prefixes, as mentioned in Subsection 5.2. Regarding DistEclat, instead, the behavior is the opposite, but the values are misleading: considering only the second phase of DistEclat, where the trees are mined, as shown in Figure 10, the longer the prefixes, the more balanced the mining tasks. Hence, DistEclat shows a medium-balanced overall behavior with 1-sized prefixes (50% difference between fastest and slowest tasks), a good load balancing for the second phase with 2-sized prefixes (30% difference), and a bad load balancing for the first phase, which affects the overall behavior, leading to a final 60% difference, due to the distribution of the prefixes to the different nodes, their expansion and pruning.

6.6. Communication costs

Finally, this section presents the experiments addressing the evaluation of the communication costs. Specifically, we measure the amount of data transmitted and received through the nodes network interfaces. This information has been retrieved by means of the utilities provided by the Cloudera Manager tool.

Experiments have been performed on Dataset #1 with a fixed *minsup* value of 0.1%, which was the lowest value for which all algorithms completed the extraction.

Figure 11 reports, for each algorithm, the average value among transmitted and received traffic, compared to the total execution time. Firstly, the two measures are not correlated: higher communication costs are associated

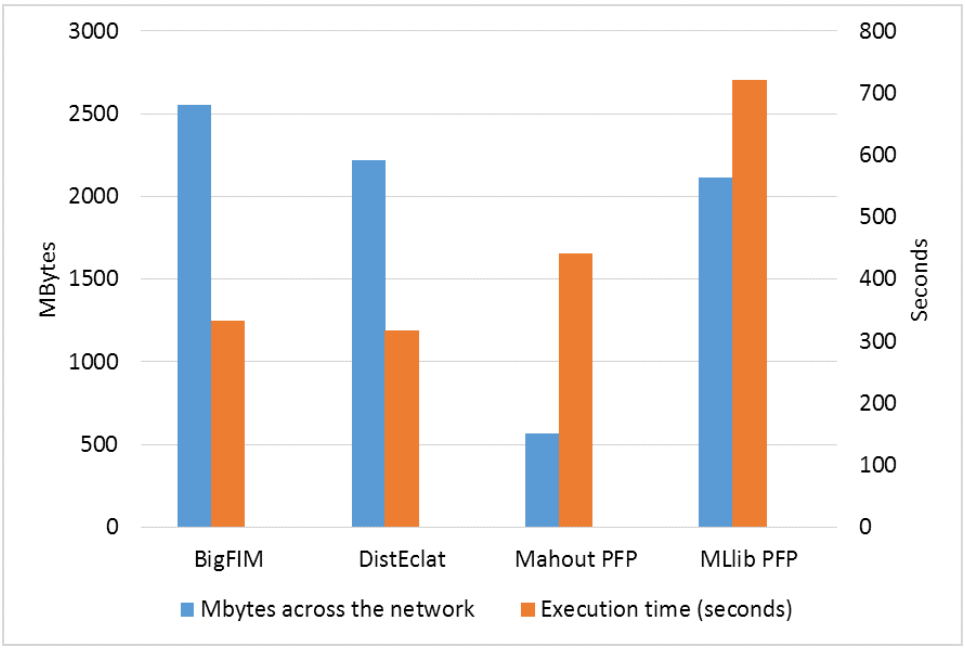

Figure 11: Communication costs and performance for each algorithm, Datasets #1, *minsup* 0.1%. The graph reports an average between transmitted and reveiced data.

with low execution times for BigFIM and DistEclat, whereas MLlib reports both measures with high values. Mahout PFP has a communication cost 4 to 5 times lower than all the others, which exchange an average of 2 Gigabytes of data. Mahout PFP average communication cost is around 0.5 Gigabytes, which is approximately the dataset size. Even though Mahout PFP is the most communication-cost optimized implementation, the very low amount of data sent through the network is related to the adoption of compression techniques, which lead to higher execution times. To address this issue, we measured the communication costs of the first phase of BigFIM and Mahout PFP in word-count toy application. BigFIM generates into the network an

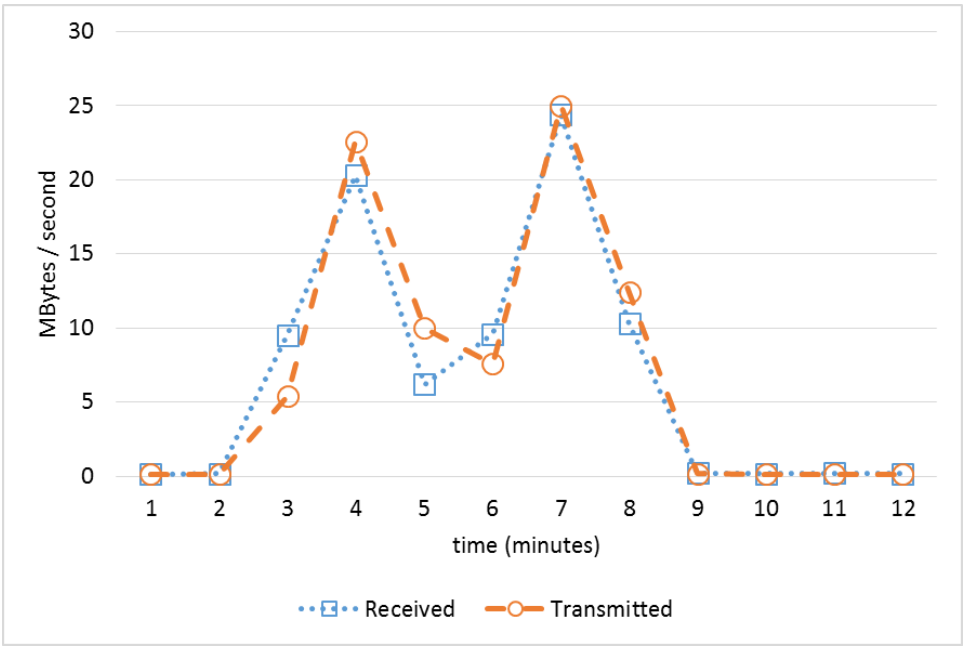

Figure 12: Communication costs of BigFIM, Datasets #1, *minsup* 0.1%. The graph reports both transmitted and reveiced data.

amount of data more than 5 times larger than Mahout PFP. However, at the same time, the execution time of BigFIM initial phase is almost 3 times faster than those of Mahout PFP.

Investigating deeper into the specific algorithm communication-cost analisys, Figures 12, 13, 15, 14 show the data exchanged throughout the whole execution time of each algorithm.

BigFIM and DistEclat, in Figure 12, 13, have a similar behaviour, with the communication costs grouped into two main phases. The first phase is related to the prefix extraction, while the second one matches the prefix preparation and prefix tree mining. BigFIM saturates the network link

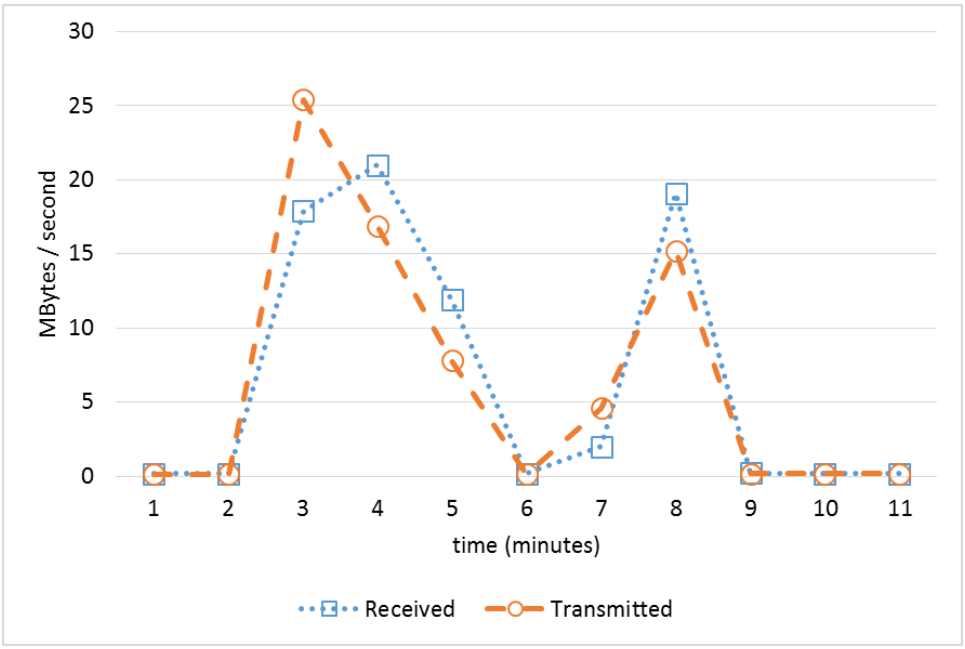

Figure 13: Communication costs of DistEclat, Datasets #1, *minsup* 0.1%. The graph reports both transmitted and reveiced data.

capacity (25 Megabytes/s, corresponding to a 100 Mbit/s full duplex Ethernet interface) during the second phase peak at minute 7 for both sent and received data, whereas DistEclat reaches such point during the first phase at minute 3, in trasmitted data only. BigFIM is also continuously exchanging data through the network during the mining process, whereas DistEclat pauses the network data exchange at minute 6. Also the MLlib PFP implementation (Figure 14) presents two phases: differently from BigFIM and DistEclat, the middle "pause", dividing the first shuffling phase and the second materialization phase, is very long and lasts from minute 4 to minute 11, it is immediately followed by a peak reaching the network capacity at minute

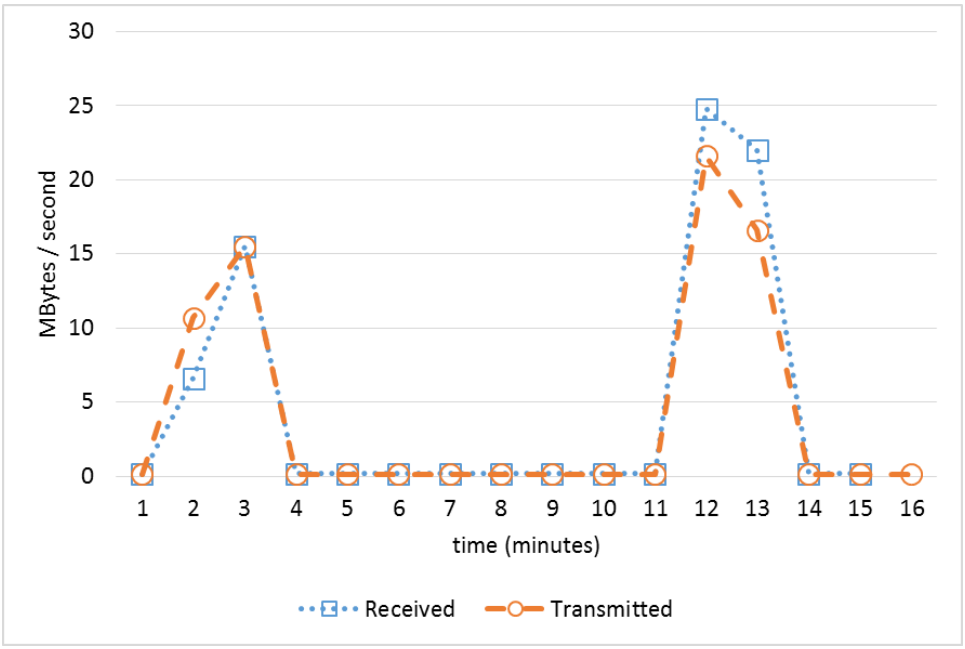

Figure 14: Communication costs of MLlib PFP, Datasets #1, *minsup* 0.1%. The graph reports both transmitted and reveiced data.

12. Mahout PFP communication costs, instead, (Figure 15) consists of three phases: the first is related to the counting part, the second one corresponds to the mining phase, while the last one is related to the aggregation of the results. The peak in network data exchange is as low as 5 Mbytes/s.

7. Lessons learned

The reported experiments provide a wide view of the different behaviours of the algorithms, dealing with diverse types of problems. With this section, we aim at supporting the reader in a conscious choice of the most suitable approach, depending on the use case at hand. Pursuing this target, we measured

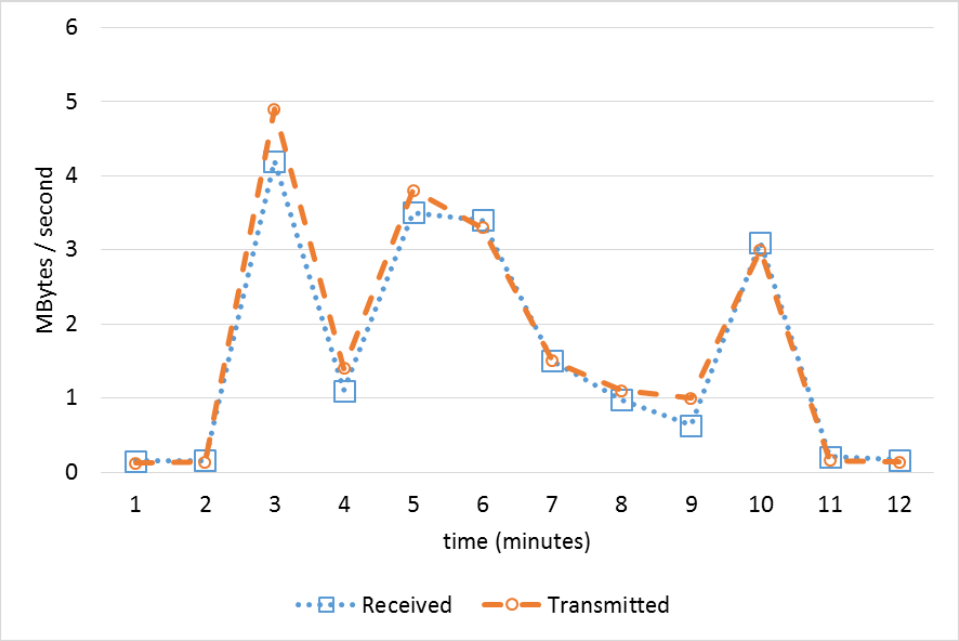

Figure 15: Communication costs of Mahout PFP, Datasets #1, *minsup* 0.1%. The graph reports both transmitted and reveiced data.

the real-life performance of the openly-available frequent-pattern mining implementations for the most popular distributed platforms (i.e., Hadoop and Spark). They have been tested on many different datasets characterised by different values of minimum support (*minsup*), transaction length (dimensionality), number of transactions (cardinality), and dataset density, besides two real-life use cases. Perfomance in terms of execution time, load balancing, and communication cost have been evaluated: a one-table summary of the results is reported in Table 8. As a result of the described experience, the following general suggestions emerge:

- Without prior knowledge of dataset density, dimensionality (average

transaction length), and cardinality (number of transactions), **Mahout PFP** is the algorithm that best guarantees the mining task completion, at the expense of longer execution times. Mahout PFP is the only algorithm able to always reach the experimental limits. Furthermore, it exchanges very few data across the network of the cluster.

- When the dataset size is small with respect to the available memory, **DistEclat** has proven to be among the fastest approaches, and also to be able to reach the lowest experimental *minsup* values. DistEclat experiments showed that it cannot scale for large or high-dimensional datasets, but when it can complete the itemset extraction, it is very fast.

- On most real-world use cases, with limited dimensionality (up to 60 items per transaction on average), **MLlib PFP** has proven to be the most reasonable tradeoff choice, with fast execution times and optimal scalability to very large datasets.

- Finally, for high-dimensional datasets, **BigFIM** resulted the fastest approach, but it cannot cope with *minsup* values as low as the others.

8. Open issues

Following the analysis of the state-of-the-art in frequent itemset mining algorithms for distributed computing frameworks, and the in-depth experimental evaluation discussed in Section 6, we can deduce that different efficient and scalable algorithms have been designed and developed during the

Table 8: Summary of the limits identified by the experimental evaluation of the algorithms (lowest *minsup*, maximum transaction length, largest dataset cardinality). The faster algorithm for each experiment is marked in bold.

	Section 6.1	Section 6.1	Section 6.2	Section 6.3
	minsup	*minsup*	transaction length	millions of transactions
Mahout PFP	0.002%	0.01%	100	100
MLlib PFP	0.002%	**0.01%**	60	**100**
BigFIM	0.1%	0.3%	**100**	100
DistEclat	**0.002%**	-	-	1

last years. However, despite the technological advancements, there is still room for improvements. Specifically, some open problems, summarized below, should be addressed to support a more effective and efficient data mining process on Big Data collections.

Algorithm selection. Many algorithms have been proposed in literature to efficiently extract correlations among data in the form of frequent itemsets, as discussed in this review. However, to apply one of the above algorithms for the analysis of a given dataset, the analyst needs to identify the best algorithm suitable for her use case, able to efficiently deal with the underlying data characteristics. The selection process is mainly based on the analyst expertise and must be handpicked for a given dataset. Thus, innovative and effective techniques that can intelligently and automatically support the analyst in the identification of the best algorithm for the current use case analysis are needed.

Parameter setting. The performance of the available algorithms to extract frequent itemsets depends on the choice of the input parameters,

like the support threshold, which dramatically impacts the execution time based on the data distribution characteristics. The optimal trade-off between execution time and result accuracy must be manually selected for any given application, based on the analysts expertise. To extract meaningful and interesting itemsets while maintaining the number of extracted results within manageable limits, a large number of experiments should be performed and the results manually evaluated by domain experts. The whole process is time consuming and requires a considerable amount of effort and skills. Thus, new scalable approaches capable of self-configuring to automatically extract actionable knowledge from massive data repositories are needed.

Full exploitation of computational capabilities of distributed frameworks. Up to now, data mining algorithms have been mainly designed to be optimized when running on centralized architectures. Furthermore, recursive primitives cannot be easily translated into distributed approaches, thus the efficiency of the current distributed implementations are limited. There is room for novel approaches natively designed to be distributed, able to efficiently address the itemset mining discovery and to fully exploit computational capabilities of distributed frameworks.

9. Summary

In this paper, we presented an overview of the state-of-the-art frequent itemset mining algorithms whose source code is available for the most widespread distributed enviroments. As summarized in Table 2, we selected the two most popular distributed frameworks, Hadoop [6] and Apache Spark [7], and 6 algorithms to address the itemset mining extraction on top of them. The

selected algorithms have been discussed along four interesting dimensions, typical of both the algorithmic and the distributed computation worlds. The comparison has been performed on 14 synthetic datasets with different data distributions and 2 real datasets representing two interesting use-case scenarios. As throughly detailed in Section 7, there is no clear winner: the most suitable approach varies based on the considered use case, data distribution, and parameter setting. Experimental results highlight that DistEclat is very efficient when the analysis can be effectively completed in main memory. BigFIM is very scalable with the number of dimensions, whereas MLlib PFP, instead, is affected by the average transaction length. Mahout PFP is less efficient but more reliable, as it scales to high dimensionality and cardinality, and low support thresholds.

Although many critical limitations related to frequent itemset mining on distributed environments have been efficiently addressed by researchers in recent years, there are still several open issues that need to be addressed to allow frequent itemset mining become a cornerstone approach in big-data mining applications. Our vision of future research directions is summarized in four major open issues, from approaches able to automatically support the analyst in selecting the best algorithm and parameters, to the design of innovative and native distributed approaches.

Acknowledgement

The research leading to these results has received funding from the European Union under the FP7 Grant Agreement n. 619633 (Project "ONTIC").

[1] E. Junqué de Fortuny, D. Martens, F. Provost, Predictive modeling with big data: is bigger really better?, Big Data 1 (4) (2013) 215–226.

[2] O. Y. Al-Jarrah, P. D. Yoo, S. Muhaidat, G. K. Karagiannidis, K. Taha, Efficient machine learning for big data: A review, Big Data Research 2 (3) (2015) 87–93. doi:10.1016/j.bdr.2015.04.001.
URL http://dx.doi.org/10.1016/j.bdr.2015.04.001

[3] R. Xu, D. Wunsch, II, Survey of clustering algorithms, Trans. Neur. Netw. 16 (3) (2005) 645–678.

[4] J. Han, H. Cheng, D. Xin, X. Yan, Frequent pattern mining: Current status and future directions, Data Min. Knowl. Discov. 15 (1) (2007) 55–86.

[5] C. C. Aggarwal, Data Classification: Algorithms and Applications, 1st Edition, Chapman & Hall/CRC, 2014.

[6] D. Borthakur, The hadoop distributed file system: Architecture and design, Hadoop Project 11 (2007) 21.

[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, I. Stoica, Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, in: NSDI'12, 2012, pp. 2–2.

[8] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, J. M. Hellerstein, Graphlab: A new framework for parallel machine learning.

[9] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, G. Czajkowski, Pregel: A system for large-scale graph processing, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, ACM, New York, NY, USA, 2010, pp. 135–146. doi:10.1145/1807167.1807184.
URL http://doi.acm.org/10.1145/1807167.1807184

[10] Apache Giraph, last Accessed: 16/10/2015 (2012).
URL http://giraph.apache.org/

[11] D. Apiletti, E. Baralis, T. Cerquitelli, S. Chiusano, L. Grimaudo, Searum: A cloud-based service for association rule mining, in: 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2013 / 11th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA-13 / 12th IEEE International Conference on Ubiquitous Computing and Communications, IUCC-2013, Melbourne, Australia, July 16-18, 2013, 2013, pp. 1283–1290.

[12] D. Antonelli, E. Baralis, G. Bruno, L. Cagliero, T. Cerquitelli, S. Chiusano, P. Garza, N. A. Mahoto, MeTA: Characterization of Medical Treatments at Different Abstraction Levels, ACM TIST 6 (4) (2015) 57.

[13] G. Cong, A. K. H. Tung, X. Xu, F. Pan, J. Yang, FARMER: finding interesting rule groups in microarray datasets, in: G. Weikum, A. C. König, S. Deßloch (Eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18,

2004, ACM, 2004, pp. 143–154. doi:10.1145/1007568.1007587.
URL http://doi.acm.org/10.1145/1007568.1007587

[14] T. Cerquitelli, E. D. Corso, Characterizing thermal energy consumption through exploratory data mining algorithms, in: Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference, EDBT/ICDT Workshops 2016, Bordeaux, France, March 15, 2016., 2016.
URL http://ceur-ws.org/Vol-1558/paper15.pdf

[15] M. L. Antonie, O. R. Zaiane, A. Coman, Application of data mining techniques for medical image classification, In MDM/KDD.

[16] E. Baralis, G. Bruno, T. Cerquitelli, S. Chiusano, A. Fiori, A. Grand, Semi-automatic knowledge extraction to enrich open linked data, Cases on Open-Linked Data and Semantic Web Applications / Patricia Ordoez de Pablos.

[17] E. Baralis, L. Cagliero, A. Fiori, P. Garza, Mwi-sum: A multilingual summarizer based on frequent weighted itemsets, ACM Trans. Inf. Syst. 34 (1) (2015) 5.

[18] A. de Andrade Lopes, R. Pinho, F. V. Paulovich, R. Minghim, Visual text mining using association rules, Computers & Graphics 31 (3) (2007) 316–326. doi:10.1016/j.cag.2007.01.023.
URL http://dx.doi.org/10.1016/j.cag.2007.01.023

[19] M. Mampaey, N. Tatti, J. Vreeken, Tell me what i need to know: Succinctly summarizing data with itemsets, in: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and

Data Mining, KDD '11, ACM, New York, NY, USA, 2011, pp. 573–581. doi:10.1145/2020408.2020499.
URL http://doi.acm.org/10.1145/2020408.2020499

[20] The Apache Mahout machine learning library, last Accessed: 16/10/2015 (2013).
URL http://mahout.apache.org/

[21] The Apache Spark scalable machine learning library, last Accessed: 16/10/2015 (2015).
URL https://spark.apache.org/mllib/

[22] S. Moens, E. Aksehirli, B. Goethals, Frequent itemset mining for big data, in: SML: BigData 2013 Workshop on Scalable Machine Learning, IEEE, 2013.

[23] H. Qiu, R. Gu, C. Yuan, Y. Huang, YAFIM: A parallel frequent itemset mining algorithm with spark, in: IPDPSW'14, 2014, pp. 1664–1671.

[24] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, in: OSDI'04, 2004, pp. 10–10.

[25] simSQL, a system for stochastic analytics, last Accessed: 16/10/2015 (2013).
URL http://cmj4.web.rice.edu/SimSQL/SimSQL.html

[26] M. P. Forum, Mpi: A message-passing interface standard, Tech. rep., Knoxville, TN, USA (1994).

[27] MADlib: Big Data Machine Learning in SQL, last Accessed: 16/10/2015.
URL http://madlib.net/

[28] Pang-Ning T. and Steinbach M. and Kumar V., Introduction to Data Mining, Addison-Wesley, 2006.

[29] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Discovering frequent closed itemsets for association rules, in: Proceedings of the 7th International Conference on Database Theory, ICDT '99, Springer-Verlag, London, UK, UK, 1999, pp. 398–416.
URL http://dl.acm.org/citation.cfm?id=645503.656256

[30] B. Goethals, Survey on frequent pattern mining, Univ. of Helsinki.

[31] R. Agrawal, R. Srikant, Fast algorithms for mining association rules in large databases, in: VLDB '94, 1994, pp. 487–499.

[32] F. N. Afrati, M. Balazinska, A. D. Sarma, B. Howe, S. Salihoglu, J. D. Ullman, Designing good algorithms for mapreduce and beyond, in: Proceedings of the Third ACM Symposium on Cloud Computing, ACM, 2012, p. 26.

[33] J. Leskovec, A. Rajaraman, J. D. Ullman, Mining of massive datasets, Cambridge University Press, 2014.

[34] A. D. Sarma, F. N. Afrati, S. Salihoglu, J. D. Ullman, Upper and lower bounds on the cost of a map-reduce computation, in: Proceedings of the VLDB Endowment, Vol. 6, VLDB Endowment, 2013, pp. 277–288.

[35] N. Agrawal, T. Imielinski, A. Swami, Database mining: A performance perspective, In IEEE TKDE 5 (6).

[36] R. Wetzker, C. Zimmermann, C. Bauckhage, Analyzing social bookmarking systems: A del.icio.us cookbook, in: Mining Social Data (MSoDa) Workshop Proceedings, ECAI 2008, 2008, pp. 26–30.

[37] A. Finamore, M. Mellia, M. Meo, M. Munafò, D. Rossi, Experiences of internet traffic monitoring with tstat, IEEE Network 25 (3) (2011) 8–14.

[38] J. Han, J. Pei, Y. Yin, Mining frequent patterns without candidate generation, in: SIGMOD '00, 2000, pp. 1–12.

[39] H. Li, Y. Wang, D. Zhang, M. Zhang, E. Y. Chang, PFP: parallel fp-growth for query recommendation, in: RecSys'08, 2008, pp. 107–114.

[40] M. J. Zaki, S. Parthasarathy, M. Ogihara, W. Li, New algorithms for fast discovery of association rules, in: KDD'97, AAAI Press, 1997, pp. 283–286.

[41] L. Vu, G. Alaghband, Mining frequent patterns based on data characteristics, in: Proceedings of 2012 International Conference on Information and Knowledge Engineering, 2012, pp. 369–375.

[42] S. Moens, E. Aksehirli, , B. Goethals, Dist-eclat and bigfim, `https://github.com/ua-adrem/bigfim` (2013).

[43] Cloudera, last Accessed: 16/10/2015.
URL `http://www.cloudera.com`

[44] M. Mellia, M. Meo, L. Muscariello, D. Rossi, Passive analysis of tcp anomalies, Computer Networks 52 (14) (2008) 2663–2676.

[45] D. Giordano, S. Traverso, L. Grimaudo, M. Mellia, E. Baralis, A. Tongaonkar, S. Saha, Youlighter: An unsupervised methodology to unveil youtube cdn changes, arXiv preprint arXiv:1503.05426.