Politecnico di Torino

COMPUTER AND CONTROL ENGINEERING
XXIX CYCLE

PhD Thesis

Frequent Itemset Mining for Big Data

Supervisor:
Prof. Elena Baralis
Prof. Pietro Michiardi

Author:
Fabio Pulvirenti
Matr. 210504

December 2016

Politecnico di Torino

Abstract

Computer and Control Engineering
XXIX Cycle

PhD

Frequent Itemset Mining for Big Data

by Fabio Pulvirenti
Matr. 210504

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, data stream from every day life. From social applications to cities infrastructures, from wearable devices to sensor-equipped buildings and vehicles, there have been very strong advances related to the data generation. Furthermore, the advances related to data collection came together with the possibility of storing data which we would have trashed in the past. The reasons behind this new trend (i.e. collecting as much data as possible) concern the new value that is given to such data. Indeed, the "big data revolution" is not meant to describe just the quantity of data. The revolution strongly affects the new value that is given to such data.

The value of these data is directly correlated to the knowledge which can be extracted from it, related to the specific use case. The common story follows a pattern in which human qualitative domain experts are outperformed by the results of a modern data analysis algorithm which can leverage a sufficient amount of data (often, this algorithm was developed by a researcher who does not know anything about that specific domain).

Thanks to big data analysis, currently, many companies are able to develop predictive models which target customers. This is possible thanks to the analysis of huge amount of customer attributes. By means of municipal data collections, in urban scenarios, crimes are predicted or interesting correlations between health and air quality are extracted. The information collected by sensors in the automotive environment, instead, is leveraged in many research domains: from self-driving algorithms training to predictive component replacement. Finally, big data analysis has already been very useful in human genome's researches.

The branch of computer science whose analytic tools are used to transform these huge collections of data into effective and useful knowledge is called *data mining*. In the last

years, the interest towards data mining has risen. The trend is noticeable in both industrial and academic environments. For companies, as already discussed, it represents a very powerful source of information. 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. On the other hand, from the academic point of view, the design of big data algorithms represent a very stimulant challenge and research opportunity. Indeed, the application of traditional data mining techniques to such large collection of data is very challenging. As the amount of data increases, the proportion of it that people is able to interpret decreases [2]. For this reason, there is a concrete need of a new generation of scalable tools which, often, require to be redesigned from scratches to cope with such an extreme and complex environment.

In this dissertation, we focus on one of the most popular data mining techniques, frequent itemset mining. Frequent itemset mining is an exploratory data analysis method used to discover frequent co-occurrence among the items of a transactional dataset (attribute-value pairs). Frequent itemsets are very useful for data summarization and correlation analysis. Frequent co-occurent patterns are commonly used to identify the most relevant insights from large collection of data which cannot be manually examined because of their size. Itemsets are also used to generate association rules, which highlights and analyze relations between objects.

Frequent itemset extraction is a very challenging problem in the big data domain. The reason is related to the nature of the problem which requires a full knowledge of the input data. In the last years several scalable techniques have been introduced. All of them relies on different search space exploration strategy and this leads to different performances related to the use case.

Thesis statement: *The target of this dissertation is to thoroughly analyze the scalable frequent itemset mining environment and, eventually, making a step forward to fill in the discovered gap.*

In the final part of this Chapter, we resume this dissertation plan highlighting our research contribution.

1.1 Dissertation plan and research contribution

The first contribution of this dissertation is the deep analysis the current state of the art of frequent itemset mining algorithms. After the exploration of the domain and the discovery of possible lacks or issues, the thesis will describe our attempt to enrich the former with new solutions and algorithms.

The aforementioned dissertation target is achieved through three main steps, useful to cluster together and label the research contributions behind them:

1. A deep analysis of the most reliable frequent itemset mining tools for big data, trying to discover possible lacks or issues
2. The enrichment of the scalable frequent pattern mining environment with a new distributed high-dimensional algorithm
3. The development of a big data mining framework in which, through distributed frequent itemset mining, a new type of itemsets are extracted

The remainder part of this section will briefly introduce each phase in order to deliver a clear idea of the structure of the dissertation work.

1.1.1 Frequent Itemset Mining for Big Data: a comparative analysis

As already mentioned, with the increasing amount of generated data, different distributed and scalable frequent itemset algorithms have been developed. They have been designed exploiting the computational advantages of distributed computing platforms, such as Apache Hadoop and Apache Spark.

This Section reviews scalable algorithms addressing the frequent itemset mining problem in the Big Data frameworks through both theoretical and experimental comparative analyses [3],[4]. Since the itemset mining task is computationally expensive, its distribution and parallelization strategies heavily affect memory usage, load balancing, and communication costs. A detailed discussion of the algorithmic choices of the distributed methods for frequent itemset mining is followed by an experimental analysis comparing the performance of state-of-the-art distributed implementations on both synthetic and real datasets. The strengths and weaknesses of the algorithms are thoroughly discussed with respect to the dataset features (e.g., data distribution, average transaction length, number of records), and specific parameter settings. Finally, based on theoretical and experimental analyses, open research directions for the parallelization of the itemset

mining problem are presented. The takeaways of the review is that no algorithm is universally superior and performances are heavily skewed by the use cases and the relative input data. Additionally, the experiments have highlighted the fundamental importance of Load Balancing, even sacrificing Communication Costs, which, in this scenario, can be considered as a price worth paying. Finally, all of the algorithms focus on being able to deal with a huge number of transactions. None of them has been designed to cope with a huge number of attributes, i.e. high-dimensional data. As shown in the next subsection, we have tried to fill in this gap.

1.1.2 A Parallel Map-Reduce Algorithm to Efficiently Support Itemset Mining on High Dimensional Data

Thanks to the spread of distributed and parallel frameworks, the development of scalable approaches able to deal with the so called Big Data has been extended to frequent itemset mining. Unfortunately, as mentioned in the previous Subsection (and clearly shown in Chapter 3), most of the current algorithms are designed to cope with low-dimensional datasets, delivering poor performances in those use cases characterized by high-dimensional data. Chapter 4 introduces PaMPa-HD [5],[6], a MapReduce-based frequent closed itemset mining algorithm for high dimensional datasets. An efficient solution has been pro- posed to parallelize and speed up the mining process. Furthermore, different strategies have been proposed to easily tune-up the algorithm parameters. The experimental results, performed on real-life high-dimensional use cases, show the efficiency of the proposed approach in terms of execution time, load balancing and robustness to memory issues.

1.1.3 Big Data Mining frameworks and Misleading Generalized Itemsets

The analysis of data often includes a large number of steps; frequent itemset extraction is often just one of the required processes to extract the desired knowledge from raw data.

The availability of distributed and parallel platforms has allowed not only the spread of scalable algorithms and tools, but also the design of complete big data mining systems like the one presented in [7]. These systems design fits large amount of data starting from the very first data preparation steps.

In these frameworks, distributed frequent itemset mining is just one of the possible steps or 'modules'. It can be replaced by other data analyses or used to support further data

mining processes. In Chapter 5 we will introduce a big data mining framework effective for a wide range of analyses. We will specifically focus on the extraction of 'misleading generalized itemset' [8], a particular type of itemsets obtained from frequent itemsets and a taxonomy of the input data. In this dissertation, two real life use cases will be analyzed. The first is related to smart cities [8],[9] while the second will analyze network traffic logs [10].

1.1.4 Dissertation Plan

This dissertation is organized in the following way. Chapter 2 introduce the background related to frequent itemset mining and the distributed platforms involved. Most of all, it will motivate the problem statement and explain the challenges related to scalable implementations of new frequent itemset mining algorithms. In Chapter 3 a thorough review of the most affirmed solutions will be introduced. The performance of the best-in-class implementation will be evaluated through the utilization of synthetic and real datasets, evidencing the current limitation of the academic state of the art. Then, in Chapter 4 an innovative distributed algorithm will be presented and evaluated, demonstrating its effectiveness in the context of high-dimensional pattern mining. In Chapter 5 a big data mining framework will be introduced and exploited to obtain a special type of frequent itemsets from network traffic and smart cities datasets. Finally, Chapter 6 summarizes the main results we achieved and provides some future possible work directions.

Chapter 2

Background and Motivations

2.1 Data Mining and Frequent Itemset Mining

As already introduced, data mining represents a family of tools and techniques aimed at extracting usable and effective knowledge from collections of data. It is possible to distinguish 3 main groups of techniques:

- Unsupervised Learning (Clustering) [11]
- Supervised Learning [12]
- Frequent Itemset Mining and Correlation Discovery [13]

The goal of clustering and, more in general, unsupervised learning is to discover hidden structures in unlabeled data. Specifically, the aim of this set of techniques is grouping sets of objects in such a way that objects grouped together (in the same cluster) are more similar to each other than to those in other groups (clusters). The greater the homogeneity inside a group and the dissimilarity among different groups, the better the clustering results can be considered. The division into groups can be seen as an attempt to get the natural structure of the data.

Supervised Learning, instead, starting from a set of labeled input data, aims at building a predictive model from it. This model, which is an inferred function, should approximate the distribution of the input dataset, called training set, with respect to the class labels. The built model is then used to classify new unlabeled samples.

Frequent Itemset Mining is an exploratory data analysis method used to discover frequent co-occurrence among the items of a transactional dataset (attribute-value pairs). The support of an itemset, a set of items, is the number of transactions in which it

appears. A set of items is considered frequent if its support is over a user-provided frequency threshold (minimum support). Frequent Itemsets are very used also to summarize large collection of data since they output the most frequent patterns, which can be interpret as the most representative [14]. In a similar way, they can be leveraged to highlight patterns which do not respect the most common trend [15],[16]. They can hide interesting outliers which could worth to be investigated and deepened [17],[18]. Frequent itemsets are often used as input to Association rules mining, a method to discover interesting relations between objects. They were first introduced analyzing retail transactions data from supermarkets. Each rule is organized on two members, respectively called antecedent and consequent. The rule concept is very straightforward and an example rule is: $\{bread, butter\} \rightarrow \{milk\}$. This rule means that customers who buy bread and butter usually buy also milk. Of course, the rules should be considered statistically significant just if supported by a sufficient support and confidence (i.e. how often the rule has been found to be true). Association rules and, in general, the extracted knowledge in terms of correlations, could be considered very valuable information. For instance, a whole category of classifier or recommendation systems are based on rules [19],[20].

In the following Subsection, a preliminary background, useful to better understands the content of this work, will be introduced.

2.1.1 Frequent Itemset Mining - Preliminaries

Let $\mathcal{I}$ be a set of items. A transactional dataset $\mathcal{D}$ consists of a set of transactions $\{t_1, \ldots, t_n\}$, where each transaction $t_i \in \mathcal{D}$ is a set of items (i.e., $t_i \subseteq \mathcal{I}$) and it is identified by a transaction identifier (tid_i). Figure 2.1a reports an example of a transactional dataset with 4 transactions.

An itemset I is defined as a set of items (i.e., $I \subseteq \mathcal{I}$) and it is characterized by a tidlist and a support value. The tidlist of an itemset I, denoted by $tidlist(I)$, is defined as the set of tids of the transactions in $\mathcal{D}$ containing I, while the support of I in $\mathcal{D}$, denoted by $sup(I)$, is defined as the ratio between the number of transactions in $\mathcal{D}$ containing I and the total number of transactions in $\mathcal{D}$ (i.e., $|tidlist(I)|/|\mathcal{D}|$). For instance, the support of the itemset *{acd}* in the running example dataset $\mathcal{D}$ is $2/4$ and its tidlist is $\{1, 2\}$. An itemset I is considered frequent if its support is greater than a user-provided minimum support threshold $minsup$. Figure 2.1c reports the frequent itemset extracted from $\mathcal{D}$ with a minsup value equal to 2.

Given a transactional dataset $\mathcal{D}$ and a minimum support threshold $minsup$, the Frequent Itemset Mining [21] problem consists in extracting the complete set of frequent itemsets

$\mathcal{D}$	
tid	items
1	a b c d
2	a c d e
3	b c d e
4	a d e

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

TT	
item	tidlist
a	1,2,4
b	1,3
c	1,2,3
d	1,2,3,4
e	2,3,4

(B) Transposed representation of $\mathcal{D}$

Frequent Itemsets	
itemsets	Support
a	3
b	2
c	3
d	4
e	3
a c	2
a d	3
a e	2
b c	2
b d	2
c d	3
c e	2
d e	3
a c d	2
a d e	2
b c d	2
c d e	2

(C) Frequent itemset extracted from $\mathcal{D}$ with a minsup=2

FIGURE 2.1: Running example dataset $\mathcal{D}$

from $\mathcal{D}$. The dimension of the search space , which can be represented as a lattice with an empty set at the top and an itemset containing all the possible itemset at the bottom, scales exponentially with the number of items [22]. In Figure 2.2, it is shown the lattice related to our running example.

In this work, we focus also on a valuable subset of frequent itemsets called frequent closed itemsets [23]. Closed itemsets allow representing the same information of traditional frequent itemsets in a more compact form. In addition, an item or itemset I is closed in $\mathcal{D}$ if there exists no superset that has the same support count as I.
For instance, in our running example, given a $minsup = 2$, the itemset $\{ac\}$ is a frequent itemset (support=2), but it is not closed for the presence of the itemset $\{acd\}$ (support=2).
A transactional dataset can also be represented in a vertical format, which is usually a more effective representation of the dataset when the average number of items per transactions is orders of magnitudes larger than the number of transactions. In this representation, also called *transposed table* TT, each row consists of an item i and its list of transactions, i.e., $tidlist(\{i\})$. Figure 2.1b reports the transposed representation of the running example reported in Figure 2.1a.

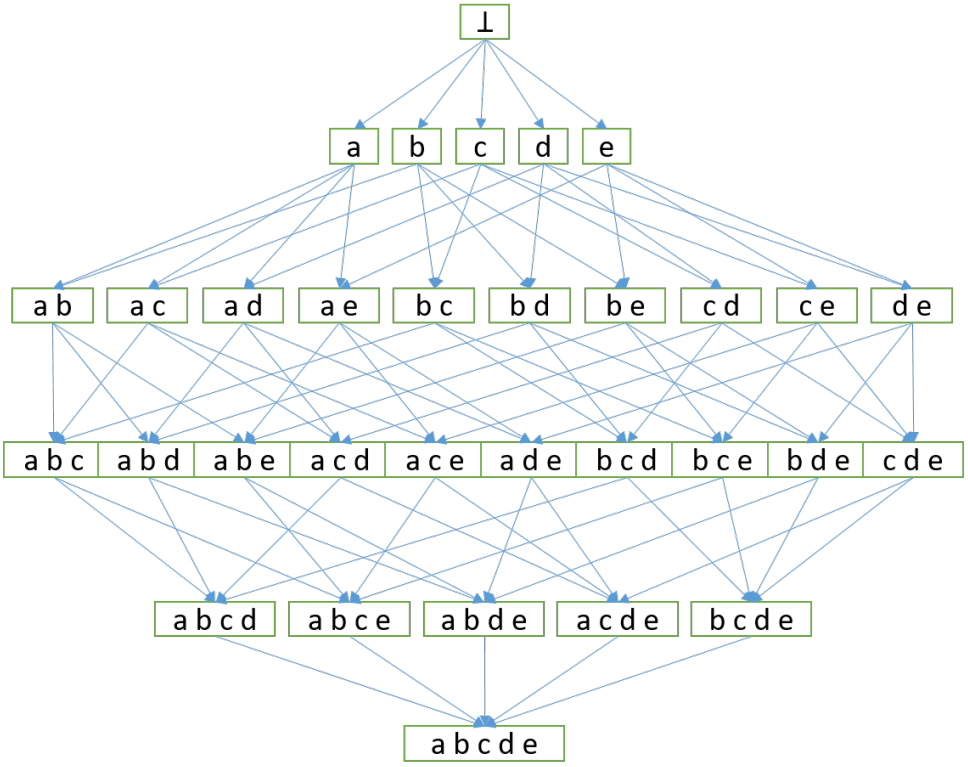

FIGURE 2.2: Lattice representing the search space of $\mathcal{D}$

2.2 Big Data and Distributed Frameworks

Today's shift towards horizontal scaling in hardware has highlighted the need of distributed algorithms. Being able to analyze big data is a huge value from both an economic and social point of view. Unfortunately, traditional tools have demonstrated to be not reliable for dealing with such large amount of data.

Starting from data storage, new solutions had to be developed to replace traditional relational database managements systems. We have firstly witnessed the development of distributed file systems such as Google File System [24] and its derivative Hadoop Distributed File System [25]. For the computational issues, already well-known parallel frameworks have shown their limitations due to fault tolerance and resiliency lacks. In the meanwhile, new processing models spread out. MapReduce [26] is the most popular example of a generic batch-oriented distributed paradigm. With its reliable and fault-tolerant architecture, it allows to exploit the resources of more commodity machines (nodes). The ratio behind the spread of the paradigm is that 'shifts the computation to the data'. In fact, taking advantage of the data locality, allowing the nodes to process just the shard of the data they store.

A MapReduce application consists of two main phases. In the first phase, called "map", each shard of the dataset is processed locally by each node of the commodity clusters, which output one or more key-values couples. Map results are exchanged among the cluster nodes and aggregate the tuples per key: this is the "shuffle" phase. This operation, which is very optimized, is one of the killer feature which a MapReduce-like algorithm should strongly exploit (it is also the unique communication among the nodes of the commodity cluster). Finally, the reduce phase is run for each unique keys and iterates through all the associated values.

Designed to cope with very large datasets, the Java-based framework Hadoop [25] is the most widely adopted MapReduce implementation. It allows programmers not to concern to low level details and to focus just on the algorithm design.

However, Hadoop and MapReduce paradigm does not fit at all iterative processes. In this case, each iteration would require a complete read and transmission (shuffle phase) of the input dataset, which is critical when dealing with huge datasets. This issue motivated the development of a new in-memory distributed platform called Apache Spark [27]. This framework, when possible, allows machines to cache data and intermediate results in memory, instead of reloading it from the disk at each iteration. Spark has also introduced a new type of data collection called RDD (Resilient Distributed Dataset). Every RDD modification is done just by the generation of another RDD, keeping trace of all the transformations in order to be able to regenerate data in case of failures. Furthermore, RDDs avoid on-disk materialization until not strictly mandatory, i.e. when an action requires a result to be returned to the driver program, saving resources in terms of communication and I/O costs. Spark supports both Graph-based and Streaming processes, demonstrating to be more flexible than Hadoop, still keeping full compatibility with the latter.

Because of the winning features of Hadoop and Spark, testified by their spread in the academic environment, in this dissertation we will focus into these distributed frameworks, analyzing the best-in-class Hadoop and Spark-based works and utilizing their paradigm for further advancements of the state of the art.

However, Hadoop and Spark are not the only frameworks supporting the parallelization of Data mining algorithms. GraphLab [28], Google Pregel [29] and its open-source counterpart Giraph [30] are fault-tolerant, graph-based framework while SimSQL [31], for instance, exploits an SQL-based approach. Distributed systems are popular also because they became very easy to use: as already stated, Message Passing Interface (MPI) [32], one of the most adopted framework in academic environment, works efficiently only on very low level programming such as C.

2.2.1 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 [33] 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 [34], instead, provides a SQL toolkit of algorithms that run over Hadoop. Finally, MLLib [35] 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.

2.3 Big Data and FIM (maybe should put this before 2.2)

In this Section we will motivate the need of scalable frequent itemeset mining algorithms and their relations with input dataset size and minimum support threshold. After that, the issues related to the problem statement will be discussed.

Input size and minimum support value. As already largely discussed, we are witnessing an explosive increasing of data availability. Process and support this extremely large amount of data is very challenging. Data mining algorithms, for their nature, as better detailed in the last part of this Section, are among the hardest processes to parallelize. Specifically, the parameters which most affect the mining are 2. The first, as predictable, is the input data size, while the second is the minimum support threshold.

The first issue is already intuitive since a bigger data collection is harder to analyze. The problem is caused by the inner data structure (e.g., FP-tree [36], Enumeration Tree [23], Prefix Tree [37], ...) leveraged by the algorithm itself to explore the search space. Larger datasets lead to larger data structures built by the algorithm itself. Larger data structures require large amount of memory and computational resources to be maintained and explored.

The second issue is related to minimum support threshold, which, in our problem, is directly mirrored to the targeted depth of the analysis. Even for datasets not belonging to big data environment, a very low support extraction could require huge amount of resources. The lower it is, the more challenging in terms of resource the mining will be. Therefore, it is likely to occur that frequent itemset miner is easily able to complete the itemsets extraction with a certain minimum support threshold, and running out of memory with the same input and a lower support. Even in this case, the motivations are

related to the inner structure used by the algorithms to explore the search space [22]. A low minimum support threshold leads to a deeper exploration of the search space. For some algorithms like Apriori [38], it could lead to the generation and testing of all the possible combinations of the dataset items. The mining considers any possible items co-occurrency and it can happen that the output of the process exceeds the input data size (as clearly shown in Tables 2.1a and 2.1c). In addition, please note that the size and the complexity of these structure do not scale linear with the minimum support threshold [21], [22]. For these reasons, this parameter is very important analyzing the performance of a frequent itemset mining algorithm.

These two aspects are the most critical issues for itemset extraction. As we will see in Chapters 3 and 4, also input data distribution has an impact. For the sake of clarity, for the moment, we ignore this feature, which is very dependent on the algorithms nature.

Motivations. Now, let us focus on the reasons behind the needs to mine huge datasets and/or extract itemsets with low minimum support thresholds. Focusing on the first aspect, the need of scalability in terms of input dataset size is an obvious consequence of the requirement to analyze huge data collections. Indeed, in big data analysis domain, frequent itemset mining is very useful tool. For instance, thinking about the summarization properties of frequent itemsets, we can assume that the larger is the dataset, the more essential would be its summarization.

As already mentioned, even low supports could represent very challenging mining processes. The reader could wonder about the need of such amount of frequent itemsets, given that, one of the most intuitive usage examples are related to data summarization. However, in many applications, frequent pattern mining can be considered as a preprocessing than a final step. One of the most intuitive contexts is the "infrequent" itemset extraction and outlier detection algorithms [15],[18]. The need of a deep extraction is also motivated by the hitches to include any interestingness measures (apart from the support) into the extraction process. Hence, a very common behavior is to extract as many itemsets as possible and then apply any sort of interestingness filter. For instance, already in this dissertation, will be introduced a work, in which a special type of itemsets is mined from a set of frequent itemsets (see Chapter 5 for further details).

Challenges. The parallelization of the mentioned data structures represents the main contribution behind the development of distributed and parallel frequent itemset mining algorithms. Like almost all the data mining tools, this technique, as already discussed, does not fit parallel and/or distributed implementations. In distributed an parallel domains, an ideal approach assumes to divide the problem into independent non-overlapping sub-problems, which can be assigned to commodity cluster nodes. In

this way, (i) the resource are completely exploited and (ii) the communication costs, a concrete bottleneck in distributed environment, are reduced as much as possible.

In FIM environment, the task that should be parallelized is the search space exploration, which is done through ad-hoc data structures. As detailed in Chapters 3 and 4, all the distributed FIM algorithms smartly split and distribute the processing of these data structures, most of the them in "divide and conquer" fashion. This technique overcomes the main memory issues. However, this split is often sub-optimal:

- First, in order to be independent for the mining task related to the respective algorithm, the set of resulting partitions could overlap each other (please note that the overlapping degree is dependent from the data distribution). Overlapping partitions entails that the global memory of the cluster should be larger than the already huge input data, storing redundant data [33],[5] which can lead to redundant and useless output [39],[5]. Furthermore, it increases the communication costs because more data have to be transmitted across the network.
- In addition, in centralized algorithms, some pruning techniques are often used to limit the search space exploration, saving time and resources. Some of the challenges related to the parallelization of exploration task parallelization are the difficulties related to these pruning techniques, which assumes a state centralized memory. In [5], we have addressed this issue, introducing a trade-off among the benefits related to a centralized memory ("state") and the ones related to the degree of parallelization (i.e. number of independent parallel tasks) (further details in Chapter 4).

These are the main challenges and the possible drawbacks related to a parallel implementation of frequent itemset mining algorithms. In the next Chapter we will evaluate how the best-in-class state of the art algorithms have addressed these questions. In Chapter 4, instead, we will introduce our solution concerning parallel *high-dimensional* frequent pattern mining.

Chapter 3

Frequent Itemset Mining for Big Data: an experimental review

3.1 Introduction

As already mentioned, existing data 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 and parallelization of data analytics techniques.

Several traditional centralized mining algorithms have been proposed. They are very efficient when the datasets can be completely loaded in main memory. However, they cannot cope with Big Data, because they are not designed for a parallel and distributed environment. The recent shift towards horizontal scalability has highlighted the need of distributed/parallelized data mining algorithms able to exploit the available hardware resources and distributed computing frameworks (e.g., Apache Hadoop [25], Apache Spark [27]). In this Section, we focus on distributed/parallel itemset mining algorithms in the Big Data context because they represent exploratory approaches widely used to discover frequent co-occurrences from the data. These algorithms have been widely exploited in different application domains (e.g., network traffic data [40], healthcare [41], biological data [42], energy data [43], images [44], open linked data [45], document and data summarization [46, 47, 48]).

The parallelization of the frequent itemset mining problem in a distributed environment by means of the MapReduce programming paradigm and a Big Data framework is not

an easy task. The main challenge is devising a smart partitioning of the problem in independent subproblems, each one based on a subset of the data, to exploit the computation power of a cluster of servers in parallel. In the following, we will describe how this problem has been addressed so far and which are pros and cons of the current parallel algorithms by taking into consideration load balancing and communication costs, which are two very important issues in the distributed domain. They are strictly related to the adopted parallelization strategy and usually represent the main bottlenecks of parallel algorithms.

The contributions of this review are the followings.

- A theoretical analysis of the algorithmic choices that have been proposed to address the itemset mining problem in the Big Data context, with the analysis of their expected impact on main memory usage, load balancing, and communication costs.
- An extensive evaluation campaign to assess the reliability of our expectations. Precisely, we ran more than 300 experiments on 14 synthetic datasets and 2 real datasets to evaluate the execution time, load balancing, and communication costs of five state-of-the-art parallel itemset mining implementations.
- The identification of strengths and weaknesses of the algorithms with respect to the input dataset features (e.g., data distribution, average transaction length, number of records), and specific parameter settings.
- The discussion of promising open research directions for the parallelization of the itemset mining problem.

The results described in this Chapter have been presented in [3] and [4] and are organized as follows. Section 3.2 provides a brief description of the state-of-the-art centralized itemset mining algorithms. Section 3.3 describes the algorithmic strategies adopted so far to partition and parallelize the frequent itemset mining problem by means of the MapReduce paradigm, while Section 3.4 describes the state-of-the-art distributed algorithms and their implementations. In Section 3.5 we benchmark the selected algorithms with a large set of experiments on both real and synthetic datasets. Section 3.6 summarizes the concrete and practical lessons learned from our evaluation analysis, while Section 3.7 discusses the open issues raised by the experimental validation of the theoretical analysis, highlighting some possible research directions to support a more effective and efficient data mining process on Big Data collections.

3.2 Centralized algorithms

The search space exploration strategies of the distributed approaches are often inspired by the solutions adopted by the centralized approaches. Hence, this section shortly introduces the main strategies of the centralized itemset mining algorithms. This introduction is useful to better understand the algorithmic choices behind the distributed algorithms.

The frequent itemset mining task is challenging in terms of execution time and memory consumption because the size of the search space is exponential with the number of items of the input dataset [22]. Two main search space exploration strategies have been proposed: (i) level-wise or breadth-first exploration of the candidate itemsets in the lattice and (ii) depth-first exploration of the lattice.

The most popular representative of the breadth-first strategy is Apriori [38]. Starting from single items, it iteratively generates and counts the support of the candidate itemsets of size $k+1$ from the frequent itemsets of size k. At each iteration k, the supports of the candidate itemsets of length k are counted by performing a new scan of the input dataset. The search space is pruned by exploiting the downward-closure property, which guarantees that all the supersets of an infrequent itemset are infrequent too. Specifically, the downward-closure property allows pruning the set of candidate itemsets of length $k+1$ by considering the frequent itemsets of length k. The Apriori algorithm is significantly affected by the density of the dataset. The higher the density of the dataset, the higher the number of frequent itemsets and hence the amount of candidate itemset stored in main memory. The problem becomes unfeasible when the number of candidate itemsets exceeds the size of the main memory.

More efficient and scalable solutions exploit the depth-first visit of the search space. FP-Growth [36], which uses a prefix-tree-based main memory compressed representation of the input dataset, is the most popular depth-first based approach. The algorithm is based on a recursive visit of the tree-based representation of the dataset with a "divide and conquer" approach. In the first phase the support of each single item is counted and only the frequent items are stored in the "header table". This information allows pruning the search space by avoiding the analysis of the itemsets extending infrequent items. Then, the FP-tree, that is a compact representation of the dataset, is built exploiting the header table and the input dataset. Specifically, each transaction is included in the FP-tree by adding or extending a path on the tree, exploiting common prefixes. Once the FP-tree associated with the input dataset is built, FP-growth recursively splits the itemset mining problem by generating conditional FP-trees and visiting them. Given an arbitrary prefix p, where p is a set of items, the conditional FP-tree with respect to p, also

called projected dataset with respect to p, is substantially the compact representation of the transactions containing p. Each conditional FP-tree contains all the knowledge needed to extract all the frequent itemsets extending its prefix p. FP-growth decomposes the initial problem by generating one conditional FP-tree for each itemt i and invoking the itemset mining procedure on each of them, in a recursive depth-first fashion.

FP-growth suits well dense datasets, because they can be effectively and compactly represented by means of the FP-tree data structure. Differently, with sparse datasets, the compressions benefits of the FP-tree are reduced because there will be a higher number of branches [13] (i.e., a large number of subproblems to generate and results to merge).

Another very popular depth-first approach is the Eclat algorithm [37]. It performs the mining from a vertical transposition of the dataset. In the vertical 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 candidate itemsets sharing a common prefix), which allows smartly merging tidlists to select frequent itemsets. Prefix-based equivalence classes are mined independently, in a "divide and conquer" strategy, still taking advantage of the downward closure property. Eclat is relatively robust to dense datasets. It is less effective with sparse distributions, because the depth-first search strategy may require generating and testing more (infrequent) candidate itemsets with respect to Apriori-like algorithms [49].

3.3 Itemset mining parallelization strategies

Two main algorithmic approaches are proposed to address the parallel execution of the itemset mining algorithms by means of the MapReduce paradigm. They are significantly different because (i) they use different solutions to split the original problem in subproblems and (ii) make different assumptions about the data that can be stored in the main memory of each independent task.

Data split approach. It splits the problem in "similar" subproblems, executing the same function on different data chunks. Specifically, each subproblem computes the local supports of all candidate itemsets on one chunk on the input dataset (i.e., each subproblem works on the complete search space but on a subset of the input data). Finally, the local results (i.e., the local supports of the candidate itemsets) emitted by each subproblem/task are merged to compute the global final result (global support of each itemset). The main assumptions of this approach are that

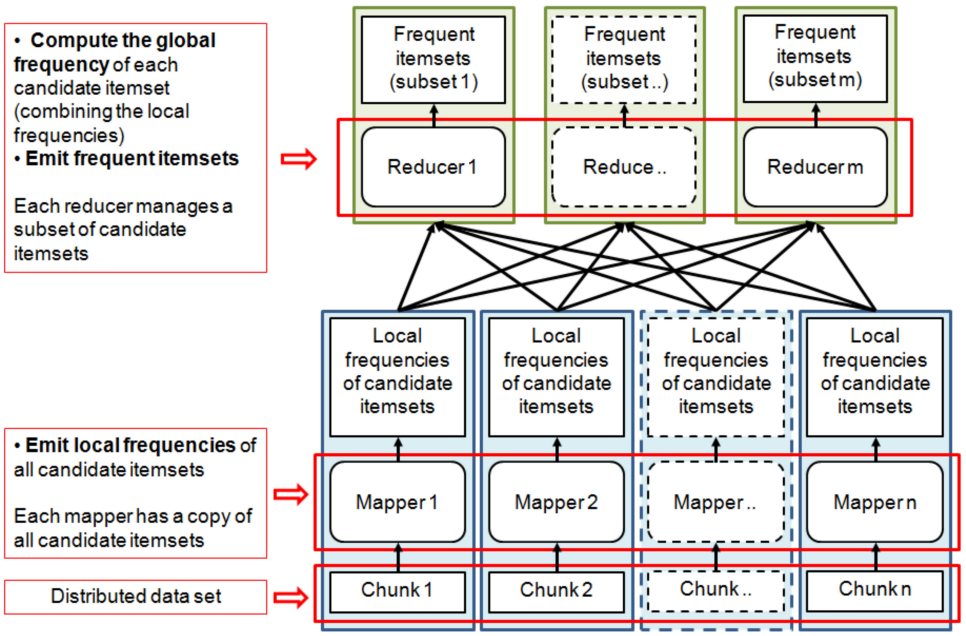

FIGURE 3.1: Itemset mining parallelization: Data split approach

(i) the problem can be split in "similar' subproblems working on different chunks of the input data and (ii) the set of candidate itemsets is small enough that it can be stored in the main memory of each task.

Search space split approach. It splits the problem by assigning to each subproblem the visit of a subset of the search space (i.e., each subproblem visits a part of the lattice). Specifically, this approach generates, from the input distributed dataset, a set of projected datasets, each one small enough to be stored in the main memory of a single task. Each projected dataset contains all the information that is needed to extract a subset of itemsets (i.e., each dataset contains all the information that is needed to explore a part of the lattice) without needing the contribution of the results of the other tasks. The final result is the union of the itemset subsets mined from each projected dataset.

Figures 3.1 and 3.3 depict the first and the second parallelization strategies, respectively. In the data split approach (Figure 3.1), the map phase computes the local supports of the candidate itemsets in its data chunk (i.e., each mapper runs a "local itemset mining extraction" on its data chunk). Then, the reduce phase merges the local supports of each candidate itemset to compute its global support. This solution requires each mapper to store a copy of the complete set of candidate itemsets (i.e., a copy of the lattice). This set must fit in the main memory of each mapper. Since the complete set of candidate itemsets is usually too large to be stored in the main memory of a single mapper, an iterative solution, inspired by the level-wise centralized itemset mining algorithms, is used. Figure 3.2 reports the iterative solution. At each iteration k only the subset

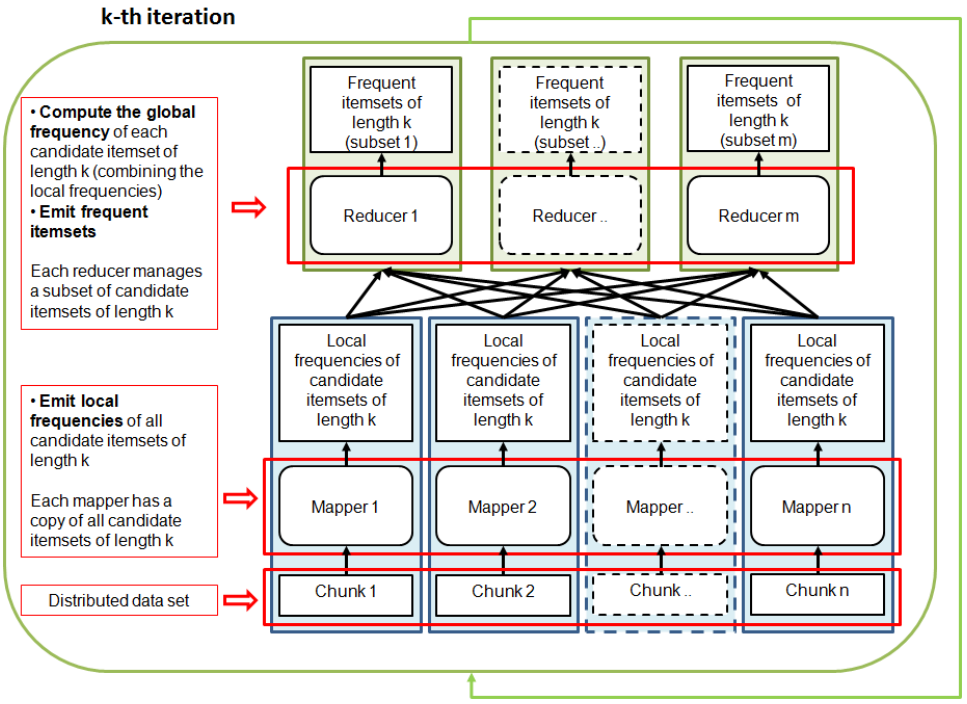

FIGURE 3.2: Itemset mining parallelization: Iterative Data split approach

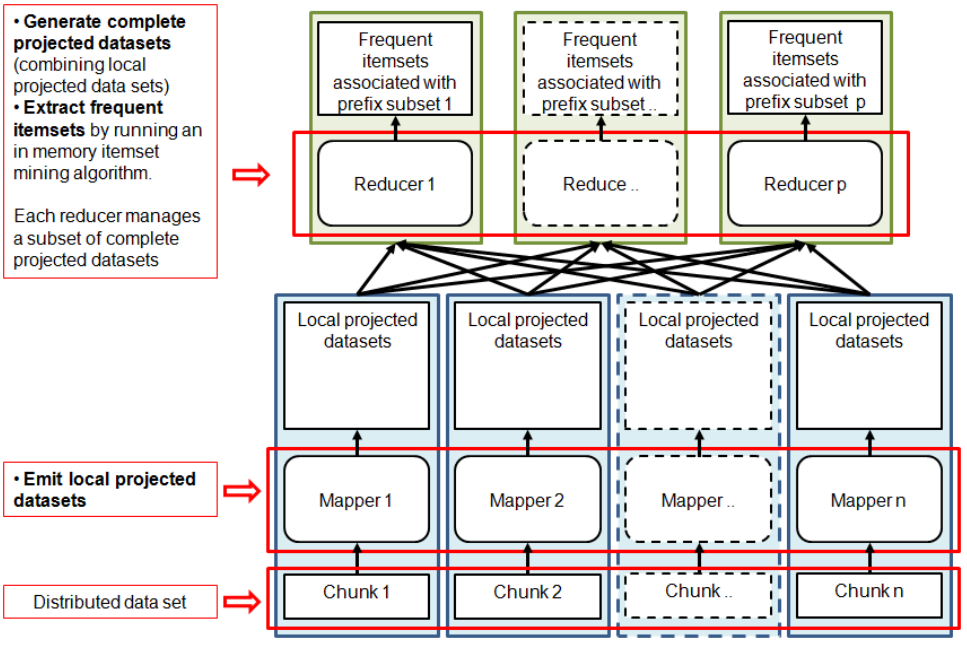

FIGURE 3.3: Itemset mining parallelization: Search space split approach

of candidates of length k are considered and hence stored in the main memory of each mapper. This approach, thanks also to the exploitation of the apriori-principle to reduce the size of the candidate sets, allows obtaining subsets of candidate itemsets that can be loaded in the main memory of every mapper.

In the search space split approach (Figure 3.3), the map phase generates a set of local

TABLE 3.1: Comparison of the parallelization approaches.

Criterion	**Iterative data split approach (Figure 3.2)**	**Search space split approach (Figure 3.3)**
Type of split/Split of the search space	Each subproblem analyzes a different subset of the input data and computes the local supports of all the candidate itemsets of length k on its chunks of data. The final result is given by the merge of the local results.	Each subproblem analyzes a different subset of itemsets/a different part of the search space. The final result is the union of the local results.
Usage of main memory	The candidate set of length k is stored in the main memory of a single task.	The complete projected dataset is stored in the main memory of a single task.
Communication cost	Number of candidate itemsets $\times$ number of mappers $\times$ number of iterations.	Sum of the sizes of the local projected datasets.
Load balancing	Load balancing is achieved by associating the same number of itemsets to each reducer.	The tasks could be significantly unbalanced depending on the characteristics of the projected datasets assigned to each node.
Maximum number of mappers	Number of chunks	Number of chunks
Maximum number of reducers	Number of candidate itemsets	Number of items

projected datasets. Specifically each mapper generates a set of local projected datasets based on its data chunk. Each local projected dataset is the projection of the input chunk with respect to a prefix p.[1] Then, the reduce phase merges the local projected datasets to generate the complete projected datasets. Each complete projected dataset is provided as input to a standard centralized itemset mining algorithm running in the main memory of the reducer and the set of frequent itemsets associated to it are mined. Each reducer is in charge of analyzing a subset of complete projected datasets by running the itemset mining phase on one complete projected dataset at a time. Hence, the main assumption, in this approach, is that each complete projected dataset must fit in the main memory of a single reducer.

Table 3.1 summarizes the main characteristics of the two parallelization approaches with respect to the following criteria: type of split of the problem, usage of main memory, communication costs, load balancing, and maximum parallelization (i.e. maximum number of mappers and reducers).

Type of split/Split of the search space. The main difference between the two parallelization approaches is the strategy adopted to split the problem in subproblems. This choice has a significant impact on the other criteria.

Usage of main memory. The different usage of the main memory of the tasks impact on the reliability of the two approaches. The data split approach assumes that the candidate itemsets of length k can be stored in the main memory of each mapper. Hence, it is not able to scale on dense datasets characterized by large candidate sets.

[1]Note that the projected datasets can overlap because the transactions associated with two distinct prefixes p_1 and p_2 can be overlapped.

Differently, the search space split approach assumes that each complete projected dataset can be stored in the main memory of a single task. Hence, this approach runs out of memory when large complete projected datasets are generated.

Communication costs. In a parallel MapReduce algorithm, communication costs are important, because the network can easily become the bottleneck if large amounts of data are sent on it. The communication costs are mainly related to the outputs of the mappers which are sent to the reducers on the network. For the data split approach the data that is sent on the network is linear with respect to the number of candidate itemsets, the number of mappers, and the number of iterations. Differently, for the search space approach, the amount of data emitted by the mappers is equal to the size of the projected datasets.

Load balancing. The different split of the problem in subproblems significantly impacts on load balancing. For the data split approach, the execution time of each mapper is linear with respect to the number of input transactions and the execution time of each reducer is linear with respect to the number of assigned itemsets. Hence, the data split approach can easily achieve a good load balancing by assigning the same number of data chunks to each mapper and the same number of candidate itemsets to each reducer. Differently, the search space split approach is potentially unbalanced. In fact, each subproblem is associated with a different subset of the lattice, related to a specific projected dataset and prefix, and, depending on the data distribution, the complexity of the subproblems can significantly vary. A smart assignment of a set of subproblems to each node would mitigate the unbalance. However, the complexity of the subproblems is hardly inferable during the initial assignment phase.

Maximum number of mappers and reducers. The two approaches are significantly different in terms of "maximum parallelization degree", at least in terms of number of maximum exploitable reducers. The maximum parallelization of the map phase is equal to the number of data chunks for both approaches. Differently, the maximum parallelization of the reduce phase is equal to the number of candidate itemsets for the data split approach, because potentially each reducer could compute the global frequency of a single itemset, whereas it is equal to the number of global projected datasets for the second approach, which can be at most equal to the number of items. Since the number of candidate itemsets is greater than the number of items, the data split approach can potentially reach a higher degree of parallelization with respect to the search space split approach.

The two parallelization approaches are used to design efficient parallel implementations of well-known centralized itemset mining algorithms. Specifically, the data split

approach is used to implement the parallel versions of level-wise algorithms (like Apriori [38]), whereas the search space split approach is used to implement parallel versions of depth-first recursive approaches (like FP-growth [36] and Eclat [37]).

3.4 Distributed itemset mining algorithms

This section describes the algorithms, and available implementations, representing the state-of-the-art solutions in the parallel frequent itemset mining context. We considered the following algorithms: YAFIM [50], PFP [51], BigFIM [39], and DistEclat [39]. The only algorithm which is lacking a publicly available implementation is YAFIM. Among the considered algorithms, YAFIM belongs to the ones based on the data split approach, while PFP and DistEclat are based on the search space split approach. Finally, BigFIM mixes the two strategies, aiming at exploiting the pros of them. For PFP we selected two popular implementations: Mahout PFP and MLlib PFP, which are based on Hadoop and Spark, respectively. The description of the four selected algorithms and their implementations are reported in the following subsections.

3.4.1 YAFIM

YAFIM [50] is an Apriori distributed implementation developed in Spark. 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 disk at each iteration. YAFIM exploits Spark RDDs to cope with these issues. Precisely, it assumes that all the dataset can be loaded into an RDD to speed up the counting operations. Hence, after the first phase in which all the transactions are loaded in an RDD, the algorithm starts the iterative Apriori algorithm organizing the candidates in a hash tree to speed up the search. Being strongly Apriori-based, it inherits the breadth-first strategy to explore and partition 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 shared 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.

3.4.2 Parallel FP-growth (PFP)

Parallel FP-growth [51], called PFP, is a distributed implementation of FP-growth that 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. PFP is based on the search space split parallelization strategy reported in Section 3.3. Specifically, the distributed algorithm is based on building independent FP-trees (i.e., projected datasets) that can be processed separately over different nodes.

The algorithm consists of 3 MapReduce jobs.
First job. It builds the Header Table, that is used to select frequent items, in a MapReduce "Word Count" manner.
Second job. In the second job, the mappers project with respect to group of items (prefixes) all the transactions of the input dataset to generate the local projected contributions to the projected datasets. Then, the reducers aggregate the projections associated with the items of the same group and build independent complete FP-trees from them. Each complete FP-tree is managed by one reducer, which runs a local main memory FP-growth algorithm on it and extracts the frequent itemsets associated with it.
Third job. Finally, the last MapReduce job selects the top k frequent closed itemsets.

The independent complete FP-trees can have different characteristics and this factor has a significant impact on the execution time of the mining tasks. As discussed in Section 3.3, this factor significantly impacts on load balancing. Specifically, when the independent complete FP-trees have different sizes and characteristics, the tasks are unbalanced because they addresses subproblems with different complexities. This problem could be potentially solved by splitting complex trees in sub-trees, each one associated with an independent subproblem of the initial one. However, defining a metric to split a tree in such a way to obtain sub-mining problems that are equivalent in terms of execution time is not easy. In fact, the execution time of the itemset mining process on an FP-Tree is not only related to its size (number of nodes) but also to other characteristics (e.g., number of branches and frequency of each node). Depending on the dataset characteristics, the communication costs can be very high, especially when the projected the datasets overlap significantly because in that case the overlapping part of the data is sent multiple times on the network.

Spark PFP [35] represents a pure transposition of PFP to Spark. It is included in MLlib, the Spark machine learning library. The algorithm implementation in Spark is very close to the Hadoop sibling. The main difference, in terms of addressed problem, is that MLlib PFP mines all the frequent itemsets, whereas Mahout PFP mines only the top k closed itemsets.

Both implementations, being strongly inspired by FP-growth, keep from the underlying centralized algorithm the features related to the search space exploration (depth-first) and the ability to efficiently mine itemsets from dense datasets.

3.4.3 DistEclat and BigFIM

DistEclat [39] is a Hadoop-based frequent itemset mining algorithms inspired by the Eclat algorithm, whereas BigFIM [39] is a mixed two-phase algorithm that combines an Apriori-based approach with an Eclat-based one.

DistEclat is a frequent itemset miner developed on Apache Hadoop. It exploits a parallel version of the Eclat algorithm to extract a superset of closed itemsets

The algorithm mainly consists of two steps. The first step extracts k-sized prefixes (i.e., frequent itemsets of length k) with respect to which, in the second step, the algorithm builds independent projected subtrees, each one associated with an independent sub-problem. Even in this case, the main idea is to mine these independent trees in different nodes, exploiting the search split parallelization approach discussed in Section 3.3.

The algorithm is organized in 3 MapReduce jobs.
First job. In the initial job, a MapReduce job transposes the dataset into a vertical representation.
Second job. In this MapReduce job, each mapper extracts a subset of the k-sized prefixes (k-sized itemsets) by running Eclat on the frequent items, and the related tidlists, assigned to it. The k-sized prefixes and the associated tidlists are then split in groups and assigned to the mappers of the last job.
Third job. Each mapper of the last mapReduce job runs the in main memory version of Eclat on its set of independent prefixes. The final set of frequent itemsets is obtained by merging the outputs of the last job.

The mining of the frequent itemsets in two different steps (i.e., mining of the itemsets of length k in the second job and mining of the other frequent itemsets in the last job) aims at improving the load balancing of the algorithm. Specifically, the split in two steps allows obtaining simpler sub-problems, which are potentially characterized by similar execution times. Hence, the application is overall well-balanced.

DistEclat is designed to be very fast but it assumes that all the tidlists of the frequent items should be stored in main memory. In the worst case, each mapper needs the complete dataset, in vertical format, to build all the 2-prefixes [39]. This impacts negatively on the scalability of DistEclat with respect to the dataset size. The algorithm inherits

from the centralized version the depth-first strategy to explore the search space and the preference for dense datasets.

BigFIM is an Hadoop-based solution very similar to DistEclat. Analogously to DistEclat, BigFIM is organized in two steps: (i) extraction of the frequent itemsets of length less than or equal to the input parameter k and (ii) execution of Eclat on the sub-problems obtained splitting the search space with respect to the k-itemsets. The difference lies in the first step, where BigFIM exploits an Apriori-based algorithm to extract frequent k-itemsets, i.e., it adopts the data split parallelization approach (Section 3.3). Even if BigFIM is slower than DistEclat, BigFIM is designed to run on larger datasets. The reason is related to the first step in which, exploiting an Apriori-based approach, the k-prefixes are extracted in a breadth-first fashion. Consequently, the nodes do not have to keep large tidlists in main memory but only the set of candidate itemsets to be counted. However, this is also the most critical issue in the application of the data split parallelization approach, because, depending on the dataset density, the set of candidate itemsets may not be stored in main memory.

Because of the two different techniques used by BigFIM in its two main steps (data split and then search space split), in the first step BigFIM achieves the best performance with sparse datasets, while in the second phase it better fits dense data distributions.

DistEclat and BigFIM are the only algorithms specifically designed for addressing load balancing and communication cost by means of the prefix length parameter k. In particular, the choice of the length of the prefixes generated during the first step affects both load balancing and communication cost.

3.5 Experimental evaluation

In this section, the results of the experimental comparison are presented. The behaviors of the algorithm reference implementations are compared by considering different data distributions and use cases. The experimental evaluation aims at understanding the relations between the algorithm performance and its parallelization strategies. Algorithm performance are evaluated in terms of (i) efficiency (i.e., execution time and scalability) under different conditions (Sections 3.5.2-3.5.6), (ii) load balancing (Section 3.5.7), and (iii) communication costs (Section 3.5.8).

3.5.1 Experimental setup

The experimental evaluation includes the following four algorithms, which are described in Section 3.4:

- the Parallel FP-Growth implementation provided in Mahout 0.9 (named Mahout PFP in the following) [33],
- the Parallel FP-Growth implementation provided in MLlib for Spark 1.3.0 (named MLlib PFP in the following) [35],
- the June 2015 implementation of BigFIM [52],
- the version of DistEclat downloaded from [52] on September 2015.

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[2], the execution times of Mahout PFP, BigFIM and DistEclat may increase with respect to MLlib PFP. However, in our experiments, the numbers of frequent itemsets and closed itemsets are in the same order of magnitude. 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. Specific experiments with different settings are explicitly indicated. The default setting of each algorithm was chosen by taking into account the physical characteristics of the Hadoop cluster, to allow each approach to exploit the hardware and software configuration at its best.

- For Mahout PFP, the default value of k is set to the lowest value forcing Mahout PFP to mine all frequent closed itemsets.
- 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 of the the number of partitions MLlib PFP cannot scale to very long transactions or very low *minsup*. Higher values, instead, do not lead to better scalability, while affecting performance.

[2]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.

- The default value of the prefix length parameter of both BigFIM and DistEclat is set to 2, which achieves a good tradeoff among efficiency and scalability of the two approaches.
- 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 parameter value is specifically 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 [53], 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 3.2, where the name of each dataset consists of pairs <parameter,value>. Finally, two real datasets have been used to simulate real-life use cases. They are described in Section 3.5.6.

All the experiments, except the speedup analysis, were performed on a cluster of 5 nodes running the Cloudera Distribution of Apache Hadoop (CDH5.3.1) [54]. Each cluster node is a 2.67 GHz six-core Intel(R) Xeon(R) X5650 machine with 32 Gigabytes of main memory and SATA 7200-rpm hard disks. 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.

For the speedup experiments we used a larger cluster of 30 nodes[3] with 2.5 TB of total RAM and 324 processing cores provided by Intel CPUs E5-2620 at 2.6GHz, running the same Cloudera Distribution of Apache Hadoop (CDH5.3.1) [54].

From a practical point of view, all the implementations revealed to be quite easy to deploy and use. Actually, the only requirement for all the implementations to be run was the Hadoop/Spark installation (from a single machine scenario to a large cluster). Only the MLlib PFP implementation requires few additional steps and some coding skills, since it is delivered as a library: users must develop their own class and compile it.

[3] http://bigdata.polito.it

TABLE 3.2: 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

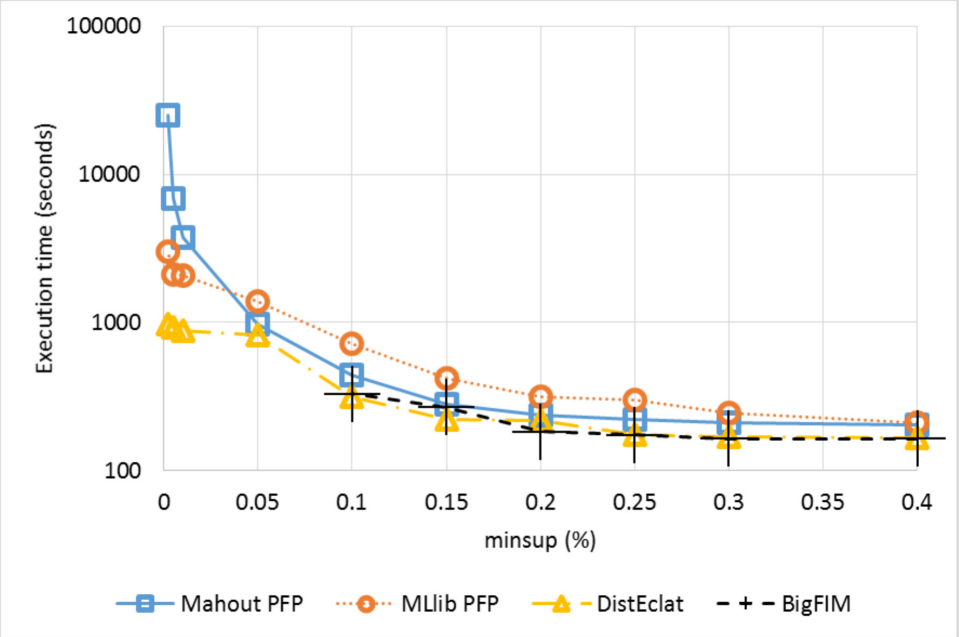

FIGURE 3.4: Execution time for different *minsup* values (Dataset #1), average transaction length 10.

3.5.2 Impact of the minsup support threshold

The minimum support threshold (*minsup*) has a high impact on the complexity of the itemset mining task.

To avoid the bias due to a specific single data distribution, two different datasets have been considered: Dataset #1 and Dataset #3 (Table 3.2). They share the same average maximal pattern length (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

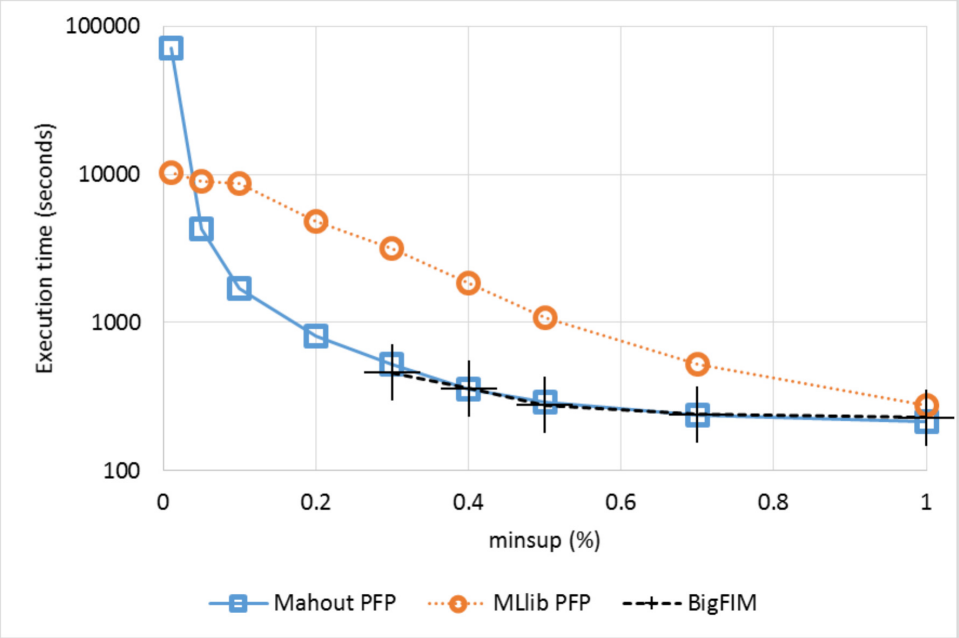

FIGURE 3.5: Execution time for different *minsup* values (Dataset #3), average transaction length 30.

for Dataset #3. Being the other characteristics constant, longer transactions lead to a higher dataset density, which results into a larger number of frequent itemsets.

Figure 3.4 reports the execution time of the algorithms when varying the *minsup* threshold from 0.002% to 0.4% and considering 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 performances. The performance gap largely increases with *minsup* values lower than 0.05%. BigFIM is as fast as DistEclat when *minsup* is higher than 0.1%, but below this threshold BigFIM runs out of memory during the extraction of 2-itemsets.

In the second set of experiments, we analyzed the execution time of the algorithms for different minimum support values on Dataset #3, which is characterized 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 3.5. 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-itemset generation phase. Finally, DistEclat was not able to

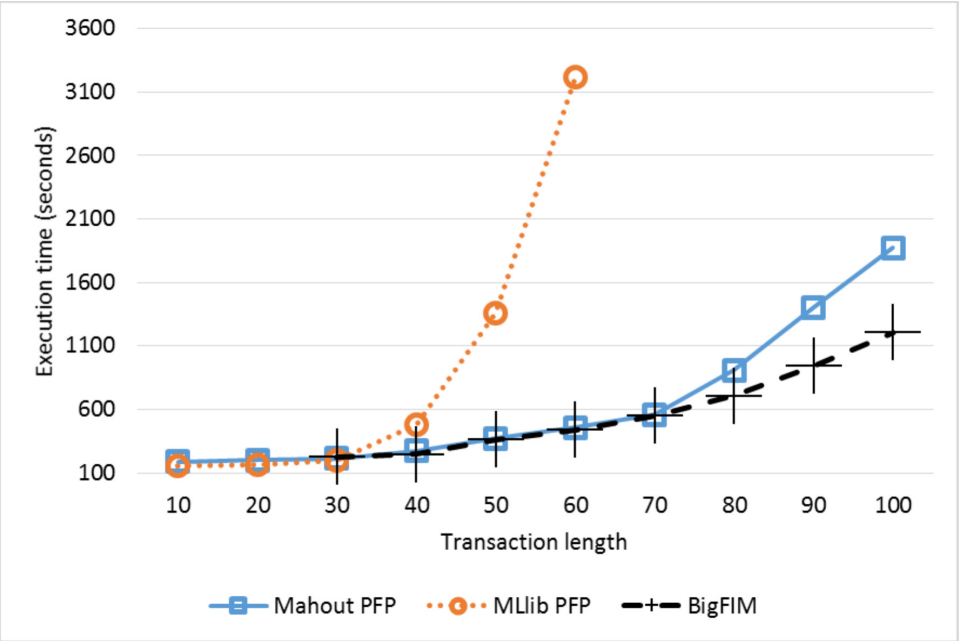

FIGURE 3.6: Execution time with different average transaction lengths (Datasets #1–10, *minsup* 1%).

run because the size of the initial tidlists was already too big.
Overall, as expected, 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.

3.5.3 Impact of the average transaction length

We analyzed the effect of different average transaction lengths, from 10 to 100 items per transaction. We fixed the number of transactions to 10 millions. To this aim, Datasets #1–10 were used (see Table 3.2). Longer transactions often lead to more dense datasets and a larger number of long frequent itemsets. This generally corresponds to more computationally intensive tasks. The execution times obtained are reported in Figure 3.6 and Figure 3.7, with a respective *minsup* value of 1% and 0.1%. In the experiment of Figure 3.6, BigFIM and DistEclat execution times for transaction length of 10 and 20 are not reported because, for these configurations, no 3-itemsets are extracted and hence the two algorithms crashed[4]. For higher transaction lengths, DistEclat is not included

[4]Due to the absence of a specific test, BigFIM and DistEclat crash if no itemsets longer than the value of the prefix length parameter are mined.

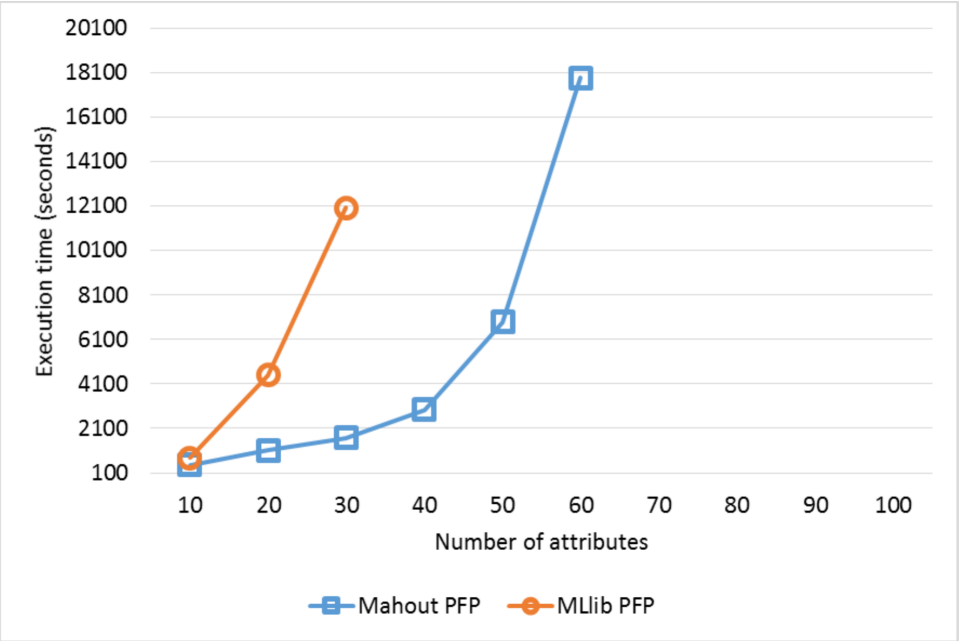

FIGURE 3.7: Execution time with different average transaction lengths (Datasets #1–10, *minsup* 0.1%).

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; (ii) Mahout PFP and BigFIM have a similar trend until 70 items per transactions, when Mahout PFP becomes slower than BigFIM.
The experiments of Figure 3.7, shows a very similar trend, with exception that also BigFIM is not able to run.
Overall, despite the Apriori-based initial phase, BigFIM proved to be the best scaling approach for very long transactions and a relatively high minsup. When the minsup is decreased only Mahout PFP is able to cope with the complexity of the task.

3.5.4 Impact of the number of transactions

We evaluated the effect of varying the number of transactions, i.e., the dataset size, without changing intrinsic data characteristics (e.g., transaction length or data distribution). The experiments have been performed on Datasets #1, #11–14 have been used (see Table 3.2), which have a number of transactions ranging from 10 millions to 1 billion. The *minsup* is set to 0.4%, which is the highest value for which the mining leverages both phases of BigFIM, and it corresponds to the highest value used in the experiments of Section 3.5.2. Since in the experiment the relative minsup threshold is

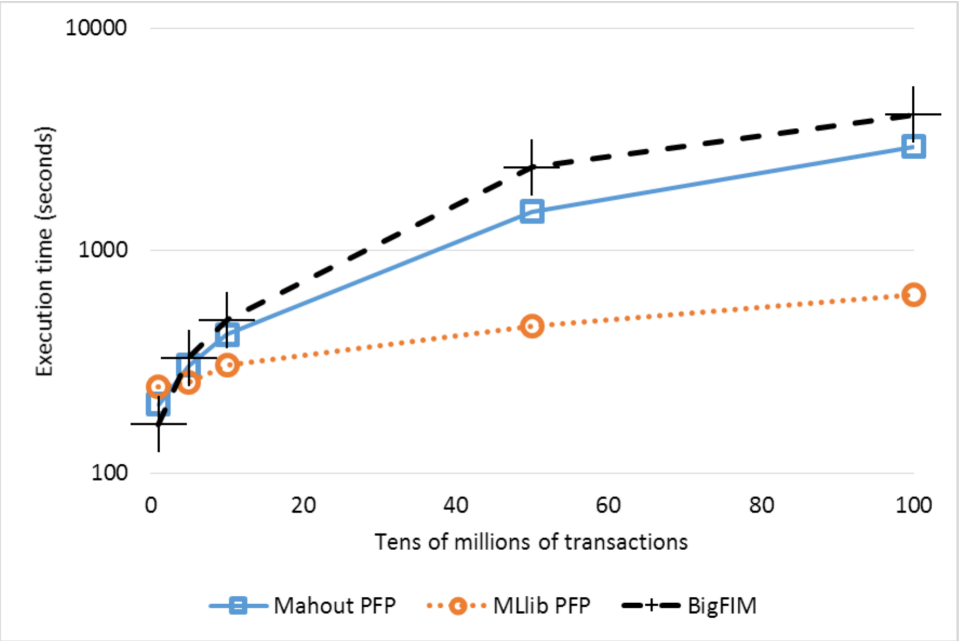

FIGURE 3.8: Execution time with different numbers of transactions (Datasets #1, #11–14, *minsup* 0.4%).

fixed, from the mining point of view, the search space exploration is similar and not particularly challenging, as shown in Section 3.5.2. What really affects this experiment is the algorithms reliability dealing with such amounts of data.

As shown in Figure 3.8, all the considered algorithms scale almost linearly with respect to the dataset cardinality, with BigFIM being the slowest, closely followed by Mahout PFP, and with MLlib PFP being by far the fastest approach, with execution times reduced by almost an order of magnitude. PFP implementations are faster than BigFIM because they read from the disk the input dataset only twice. BigFIM pays the iterative disk reading activities during its initial Apriori phase when the number of records of the input dataset increases. Finally, DistEclat fails under its assumption that the tidlists of the entire dataset should be stored in each node, and it is not able to complete the extraction beyond 10 million transactions.

3.5.5 Scalability in terms of parallelization degree

We analyzed the speedup by running the same mining problem with increasing numbers of parallel tasks. The dataset selection and the *minsup* parameter choice are difficult since we need to identify a mining problem satisfying two conditions: (i) allowing all the executions to complete with any number of parallel tasks, and, at the same time, (ii) being very demanding so that the distributed framework is actually exploited. We

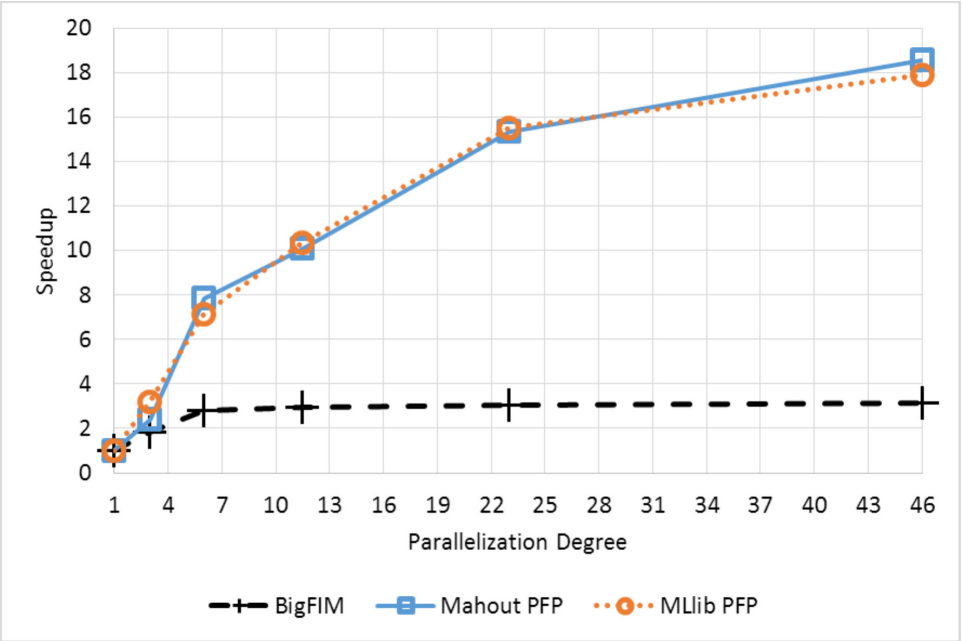

FIGURE 3.9: Speedup with different parallelization degrees (Dataset #14, *minsup* 0.4%)

selected *minsup* 0.4% and Dataset #14 (see Table 3.2) to be light enough for condition (i) and demanding enough for condition (ii).

Figure 3.9 shows the speedup results. A parallelization degree equal to 1 corresponds to the minimal computational resource setting, i.e., the configuration with only two parallel independent tasks. Its execution time is the reference with respect to which the speedup is computed. Specifically, the speedup of a configuration with a parallelization degree equal to p is computed as

$$speedup(paral\_degree = p) = \frac{Exec\_Time(paral\_degree = 1)}{Exec\_Time(paral\_degree = p)}$$

Ideally, the speedup should be equal to the parallelization degree p itself, i.e., increasing the number of resources (parallel tasks) of a factor p, should lead to a speedup equal to p.

In this experiment, it is clear that the FP-Growth-based implementations provide a better speedup. BigFIM, on the contrary, is not able to leverage a number of parallel tasks higher than 6. Because of the size of the dataset, DistEclat is not able to perform the mining.

3.5.6 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 3.3.

TABLE 3.3: 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

3.5.6.1 URL tagging

We evaluated the selected algorithms on the Delicious dataset [55], 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. It is very sparse because of the huge number of different URLs and tags. Additional characteristics of the dataset are reported in Table 3.4.

This experiment simulates the environment of a service provider that periodically analyzes 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. Because of the high sparsity of the dataset, we identified the setting *minsup*=0.01% as the best tradeoff.

Table 3.4 reports the cumulative number of transactions for the different periods of time (i.e., the cardinality of the input dataset) and the number of frequent itemsets extracted

TABLE 3.4: 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

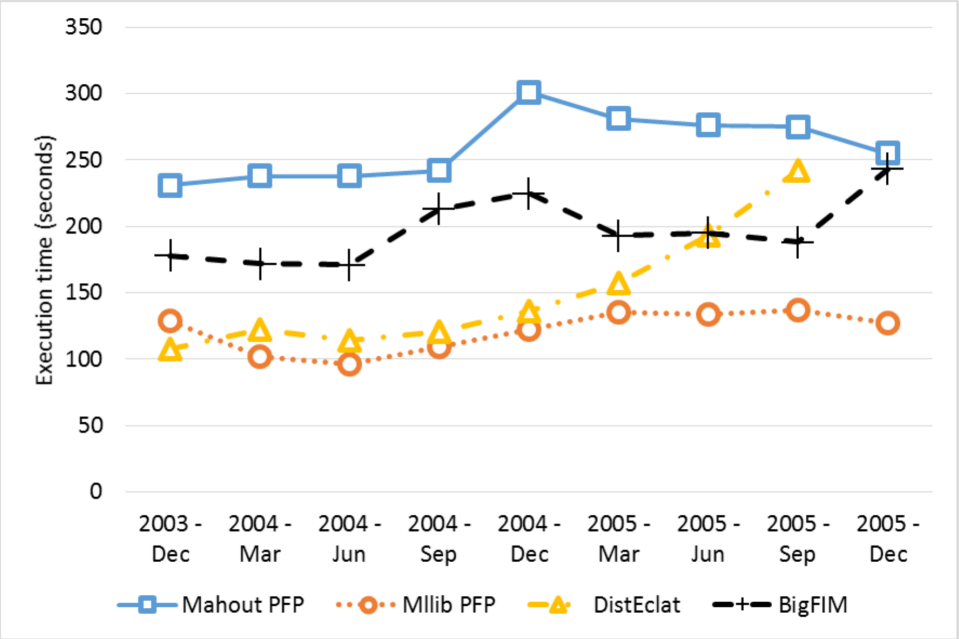

FIGURE 3.10: Execution time for different periods of time on the Delicious dataset (*minsup*=0.01%)

with a fixed *minsup* of 0.01%, while the execution times of the different algorithms are shown in Figure 3.10.

MLlib PFP consistently proves to be the fastest approach, with DistEclat following. However, while DistEclat is slightly faster than MLlib PFP only with the first, smallest dataset (up to Dec 2003, with 150 thousands transactions), when the dataset size increases, DistEclat execution time does not scale. DistEclat 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 higher execution times. Apart from DistEclat, all algorithms complete the task with similar performance despite increasing the dataset cardinality from 150 thousand transactions to 41 millions, thanks to the constant relative *minsup* threshold which reduces the number of frequent itemsets for

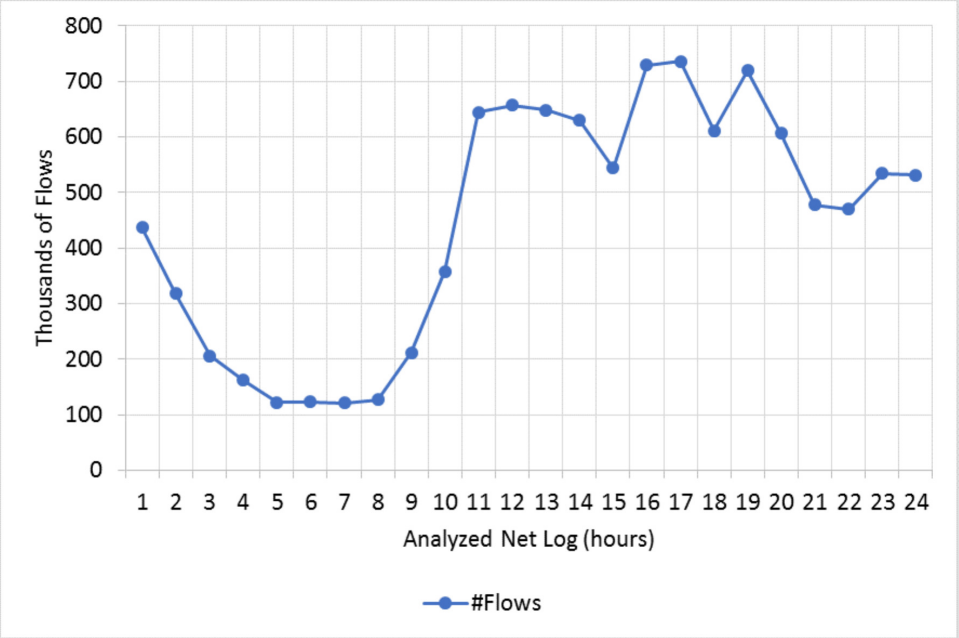

FIGURE 3.11: Number of flows for each hour of the day.

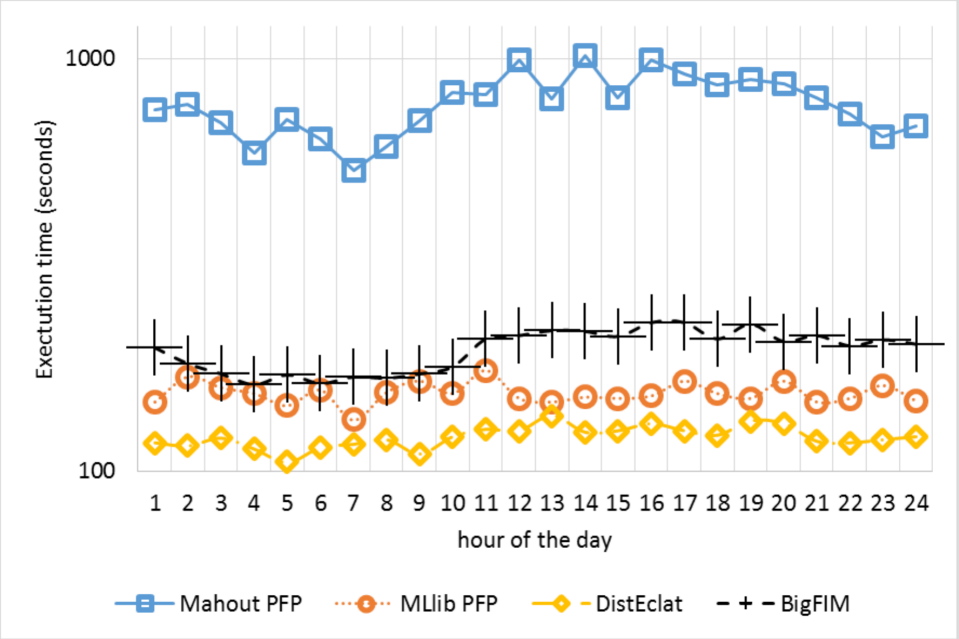

FIGURE 3.12: Execution time of different hours of the day. (dataset 31, *minsup*=1%)

decreasing density of the dataset. Hence, MLlib PFP is the best choice for this dataset characterized by short transactions (the transaction length is 4).

3.5.6.2 Network traffic flows

This use case entails the analysis of a network environment by using a network traffic log dataset, where each transaction represents a TCP flow. A network flow is a bidirectional

TABLE 3.5: 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

communication between a client and a server. The dataset has been gathered through Tstat [56, 57], a popular internet traffic sniffer broadly used in literature [58, 59], by performing a one 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). Numerical attributes have been discretized by using the same approach adopted in [59]. 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 3.11.

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 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.

The results are reported in Figure 3.12, 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

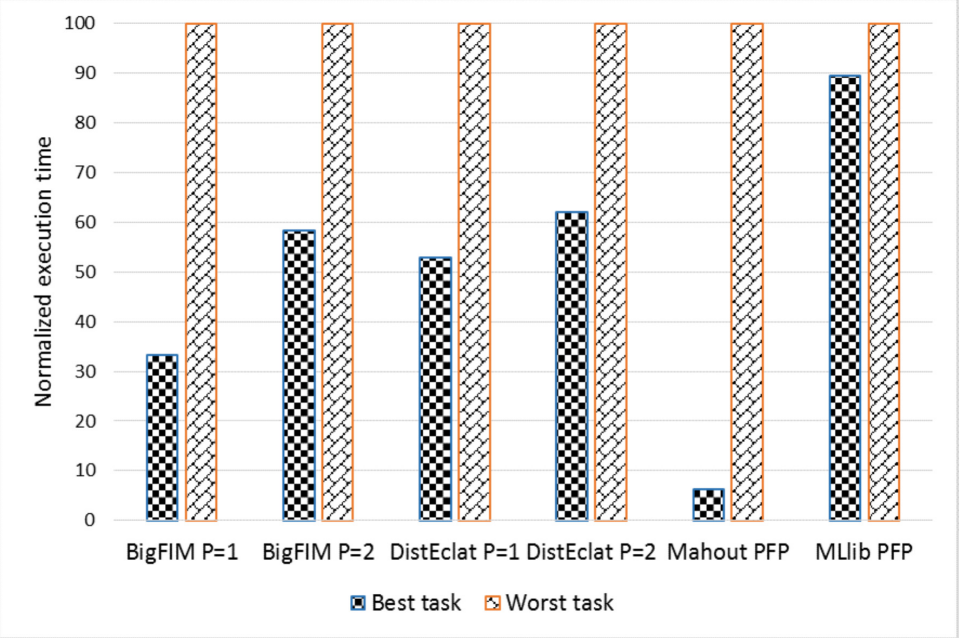

FIGURE 3.13: Normalized execution time of the most unbalanced tasks.

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.

3.5.7 Load balancing

We analyzed load balancing on a 1-hour-long subset of the network log dataset (Table 3.3) 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 3.13.

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%. The difference between MLlib PFP and Mahout PFP can be correlated to the granularity of the subproblems. The smaller the subproblems, the better the load balancing because their execution times are more similar. MLlib PFP allows specifying the number of partitions, i.e., of subproblems, which obviously impacts on the granularity of each subproblem. Hence, setting opportunely this parameter, a good load balancing result is achieved. Differently, Mahout PFP automatically sets the number of subproblems and

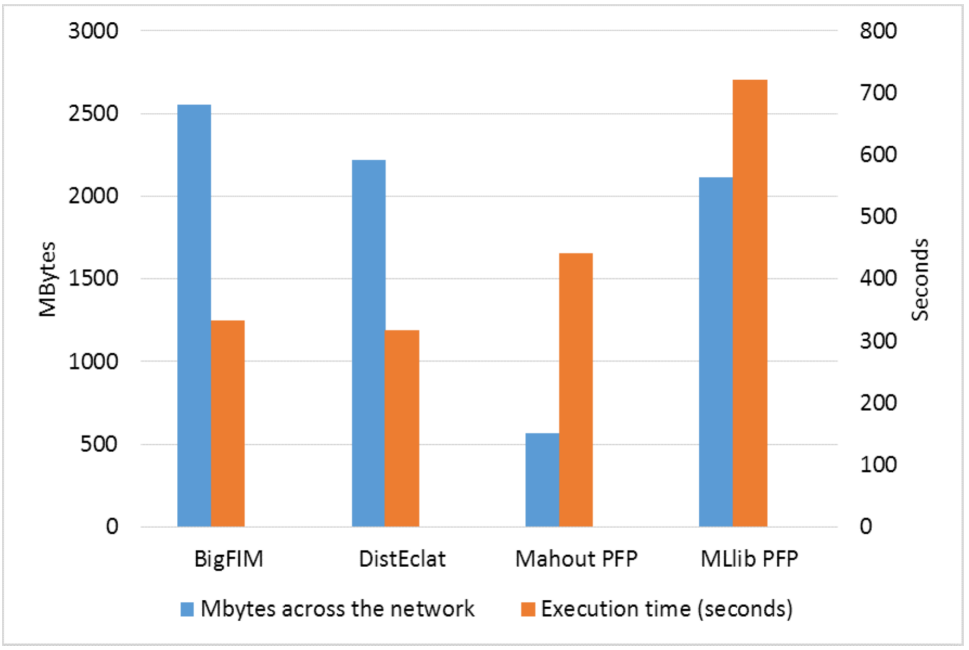

FIGURE 3.14: Communication costs and performance for each algorithm, Datasets #1, *minsup* 0.1%. The graph reports an average between transmitted and reveiced data.

the current heuristic used to set it does not seem to work well on the considered datasets (unbalanced subproblems are generated).

We included BigFIM and DistEclat with 2 different first-phase prefix sizes. For these algorithms, 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 3.4.3.

3.5.8 Communication costs

To evaluate the communication cost, 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.

The 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 3.14 reports, for each algorithm, the average value among transmitted and received traffic, compared to the total execution time. Firstly, the two measures do not seem to be correlated: higher communication costs are associated 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. The difference between DistEclat and BigFIM is not large because with only 2-length prefixes just an extra iteration is done by BigFIM. 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.

3.5.9 Discussion

The experiments confirm that the performance of the data-split-based algorithms (i.e., BigFIM in its first phase) is highly affected by the number of candidate itemsets, which must be stored in the temporary main memory of each task. Specifically, BigFIM crashes during its Apriori-based phase when low *minsup* values or dense datasets are considered, due to the large number of generated candidate itemsets. This issue does not affect the approaches based on the search split strategy (Mahout PFP and MLlib PFP), since they do not need to store candidate itemsets as an intermediate result. Hence, Mahout PFP and MLlib PFP proved to be more suitable than BigFIM to process large dataset sizes, high-density datasets, and low *minsup* thresholds. DistEclat deserves a separate consideration: even if it is based on the search space approach, it often runs out of memory, because in its initial job it needs to store the *tidlists* of all frequent items in main memory and this operation becomes easily unfeasible when large or dense datasets are considered.

Experiments also highlight the predominant importance of load balancing in the itemset mining problem, in particular when comparing BigFIM to Mahout PFP. Since the initial mining phase of BigFIM is based on the data split parallelization approach, it reads many times the input dataset (differently than Mahout PFP). Moreover, BigFIM is also characterized by greater communication costs than Mahout PFP. These two factors should impact significantly on the execution time of BigFIM. Instead, not only the execution time of BigFIM is comparable with that of Mahout PFP with 1000-million record datasets (Figure 3.8), but BigFIM is also even faster than Mahout PFP in specific cases, e.g., with datasets with an average number of items per transaction greater than 70 (Figure 3.6). The rationale of such results is the better load balancing of BigFIM with respect to Mahout PFP. Results highlight that load balancing seems to be predominant on the number of dataset reads (I/O costs) and communication costs in the parallelization of the itemset mining problem.

3.6 Lessons Learned

The reported experiments provide a wide view of the different behaviours of the algorithms in various experimental settings. 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 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 characterized by different values of minimum support (*minsup*), transaction length (dimensionality), number of transactions (cardinality), and dataset density, besides two real-life use cases. Performance in terms of execution time, load balancing, and communication cost have been evaluated: a one-table summary of the results is reported in Table 3.6. As a result of the described experience, the following general suggestions emerge:

- **High reliability.** 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.
- **High cardinality and low-dimensional data.** 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.
- **High-dimensional data.** For high-dimensional datasets, **BigFIM** resulted the fastest approach, but it cannot cope with *minsup* values as low as the others. In those cases, **Mahout PFP** represents the only option.
- **Limited dataset size.** 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.

3.7 Open research issues

The comparative study presented in this review highlighted interesting research directions to enhance distributed itemset mining algorithms for Big Data.

TABLE 3.6: 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 3.5.2	Section 3.5.2	Section 3.5.3	Section 3.5.4
	minsup	*minsup*	transaction length	millions of transactions
Mahout PFP	0.002%	0.01%	**100 (0.1%)**	100
MLlib PFP	0.002%	**0.01%**	60	**100**
BigFIM	0.1%	0.3%	100 (1%)	100
DistEclat	**0.002%**	-	-	1

Smarter load balancing techniques. The experimental evaluation allowed us to show that load balancing issues significantly affect distributed itemset mining performance, more than communication and I/O costs (e.g., reading the dataset many times). Specifically, the different complexity among the task-level sub-problems leads to load unbalance in the cluster (i.e., some sub-problems are more computationally expensive and time consuming than others causing inefficient resource usage). Load balancing improvements should be addressed in the design of new distributed frequent itemset mining algorithms. In that context, we believe that a new research direction to investigate is the definition of variable-length prefixes, with respect to which the mining sub-problems are defined, hence leading to a more balanced exploration of the search space.

Self-tuning itemset mining frameworks. As discussed in this analysis, different algorithms have been proposed in literature to discover frequent itemsets. However, the efficient exploitation of each algorithm strongly depends on specific skills and expertise. The analyst is required to select the best method to efficiently deal with the underlying data characteristics, and manually configure it (e.g., from input parameters settings, such as the *minsup* threshold, the k parameter of BigFIM, etc., to distributed frameworks tuning). Thus, state-of-the-art algorithms may become ineffective because of the inefficient hand-picked choices of the inappropriate specific implementations, and cumbersome parameter-configuration sessions. The improvements in algorithm usability should be addressed by designing innovative self-tuning itemset mining frameworks, capable of intelligently selecting the most appropriate itemset extraction algorithm and automatically configuring it. **Missing support for really high-dimensional datasets** The performance analysis have included a mining experiments on datasets with up to 100 dimensions. Even if Mahout PFP has outperformed the competitors, its performances are still very weak for low minimum support values. On the other hand, 100-features dataset certainly do not represent a state-of-the-art high dimensional problem, which can be instead characterized by thousands of million of dimensions. Thus, arises from the review is a concrete lack of a real scalable implementation which focus on the number of items per transaction.

Chapter 4

Frequent Itemset Mining for High-Dimensional data

4.1 Introduction

In the last years, the increasing capabilities of recent applications to produce and store huge amounts of information, the so called "Big Data" [60], have changed dramatically the importance of the intelligent analysis of data. Data mining, together with machine learning [61], is considered one of the fondamental tools on which Big Data analytics are based. In both academic and industrial domains, the interest towards data mining, which focuses on extracting effective and usable knowledge from large collections of data, has risen. The need for efficient and highly scalable data mining tools increases with the size of the datasets, as well as their value for businesses and researchers aiming at extracting meaningful insights increases.

Frequent (closed) itemset mining is among the most complex exploratory techniques in data mining. It is used to discover frequently co-occurring items according to a user-provided frequency threshold, called minimum support. Existing mining algorithms revealed to be very efficient on simple datasets but very resource intensive in Big Data contexts. In general, the application of data mining techniques to Big Data collections is characterized by the need of huge amount of resources. For this reason, we are witnessing the explosion of parallel and distributed approaches, typically based on distributed frameworks, such as Apache Hadoop [25] and Spark [27]. As clearly shown in Chapter 3, unfortunately, most of the scalable distributed techniques for frequent itemset mining have been designed to cope with datasets characterized by few items per transaction (low dimensionality, short transactions). Their design, on the contrary, focuses on very

large datasets in terms of number of transactions. Currently, only single-machine implementations exist to address very long transactions, such as Carpenter [23], and no distributed implementations at all.
Nevertheless, many scientific applications, such as bioinformatics or networking, generate a large number of events characterized by a variety of features. Thus, high-dimensional datasets have been continuosly generated. For instance, most gene expression datasets are characterized by a huge number of items (related to tens of thousands of genes) and a few records (one transaction per patient or tissue). Many applications in computer vision deal with high-dimensional data, such as face recognition. An increasing portion of big data is actually related to geospatial data [62] and smart-cities. Some studies have built this type of large datasets measuring the occupancy of different car lanes: each transaction describes the occupancy rate in a captor location and in a given timestamp [63]. In the networking domain, instead, the heterogeneous environment provides many different datasets characterized by high-dimensional data, such as URL reputation, advertising, and social network datasets [64]. To effectively deal with those high-dimensional datasets, novel and distributed approaches are needed.

This work introduces PaMPa-HD [5], [6], a parallel MapReduce-based frequent closed itemset mining algorithm for high-dimensional datasets. PaMPa-HD relies on the Carpenter algorithm [23]. The PaMPa-HD design[1],through an ad-hoc synchronization technique, takes into account crucial design aspects, such as load balancing and robustness to memory-issues. Furthermore, different strategies have been proposed to easily tune up the parameter configuration. The algorithm has been thoroughly evaluated on real high dimensional datasets. PaMPa-HD outperforms the state-of-the-art distributed approaches in execution time and by supporting lower minimum support threshold.

The paper is organized as follows: Section 4.2 briefly reintroducess the frequent (closed) itemset mining problem, Section 4.3 briefly describes the centralized version of Carpenter, and Section 4.4 presents the proposed PaMPa-HD algorithm. Section 4.5 describes the experimental evaluations proving the effectiveness of the proposed technique, Section 4.6 discusses possible applications of PaMPa-HD and, finally, Section 4.7 introduces future works and conclusions.

4.2 Frequent itemset mining background

Since Frequent Itemset Mining preliminaries were introduced far before in the dissertation, let us just recall and deepen the key concepts fundamental to better understand

[1]The source code of PaMPa-HD can be downloaded from https://github.com/fabiopulvi/PaMPa-HD

$\mathcal{D}$	
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,f,v
5	a,b,d,f,g,l,q,s,t

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

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

(B) Transposed representation of $\mathcal{D}$

$TT\|_{\{2,3\}}$	
item	tidlist
a	4,5
e	-
h	-
v	4

(C) $TT|_{\{2,3\}}$: example of conditional transposed table

FIGURE 4.1: Running example dataset $\mathcal{D}$

PaMPa-HD and its enumeration tree-based exploration strategy. The running example has been slightly modified from the one presentes in Chapter 2.

As already mentioned, a transactional dataset can also be represented in a vertical format, which is usually a more effective representation of the dataset when the average number of items per transactions is orders of magnitudes larger than the number of transactions. This representation, called *transposed table* TT, assumes that each row consists of an item i and its list of transactions, i.e., $tidlist(\{i\})$. Let r be an arbitrary row of TT, $r.tidlist$ denotes the tidlist of row r. Figure 4.1b reports the transposed representation of the running example reported in Figure 4.1a.

Given a transposed table TT and a tidlist X, the conditional transposed table of TT on the tidlist X, denoted by $TT|_X$, is defined as a transposed table such that: (1) for each row $r_i \in TT$ such that $X \subseteq r_i.tidlist$ there exists one tuple $r'_i \in TT|_X$ and (2) r'_i contains all tids in $r_i.tidlist$ whose tid is higher than any tid in X. For example, consider the transposed table TT reported in Figure 4.1b. The projection of TT on the tidlist *{2,3}* is the transposed table reported in Figure 4.1c. Each transposed table $TT|_X$ is associated with an itemset composed by the items in $TT|_X$. For instance, the itemset associated with $TT|_{\{2,3\}}$ is *{aehv}* (see Figure 4.1c).

4.3 The Carpenter algorithm

The most popular techniques to perform itemset mining (e.g., Apriori [38] and FP-growth [36]) adopt the itemset enumeration approach (see Section 3.2 for further discussion). However, itemset enumeration revealed to be ineffective with datasets with a high average number of items per transactions [23]. To tackle this problem, the Carpenter algorithm [23] was proposed. Specifically, Carpenter is a frequent itemset extraction algorithm devised to handle datasets characterized by a relatively small number of transactions but a huge number of items per transaction. To efficiently solve the itemset mining problem, Carpenter adopts an effective depth-first transaction enumeration approach based on the transposed representation of the input dataset. To illustrate the centralized version of Carpenter, we will use the running example dataset $\mathcal{D}$ reported in Figure 4.1a, and more specifically, its transposed version (see Figure 4.1b). Recall that in the transposed representation each row of the table consists of an item i with its tidlist. For instance, the last row of Figure 4.1b shows that item v appears in transactions 1, 2, 3, 4.

Basically, Carpenter builds a transaction enumeration tree by exploiting a set of pruning rules which avoid the expansion of useless branch of the tree. In the tree, each node corresponds to a conditional transposed table $TT|_X$ and its related information (i.e., the tidlist X with respect to which the conditional transposed table is built and its associated itemset). The transaction enumeration tree, when pruning techniques are not applied, contains all the tid combinations (i.e., all the possible tidlists X). Figure 4.2 reports the transaction enumeration tree obtained by processing the running example dataset. To avoid the generation of duplicate tidlists, the transaction enumeration tree is built by exploring the tids in lexicographical order (e.g., $TT|_{\{1,2\}}$ is generated instead of $TT|_{\{2,1\}}$). Each node of the tree is associated with a conditional transposed table on a tidlist. For instance, the conditional transposed table $TT|_{\{2,3\}}$ in Figure 4.1c, matches the node $\{2, 3\}$ in Figure 4.2.

Carpenter performs a depth first search (DFS) of the enumeration tree to mine the set of frequent closed itemsets. Referring to the tree in Figure 4.2, the depth first search would lead to the visit of the nodes in the following order: {1}, {1,2}, {1,2,3}, {1,2,3,4}, {1,2,3,4,5}, {1,2,3,5}, {...}. For each node, Carpenter applies a procedure that decides if the itemset associated with that node is a frequent closed itemset or not. Specifically, for each node, Carpenter decides if the itemset associated with the current node is a frequent closed itemset by considering:

1. The tidlist X associated with the node, useful to enforce the depth-first exploration and to check the actual support of the itemset

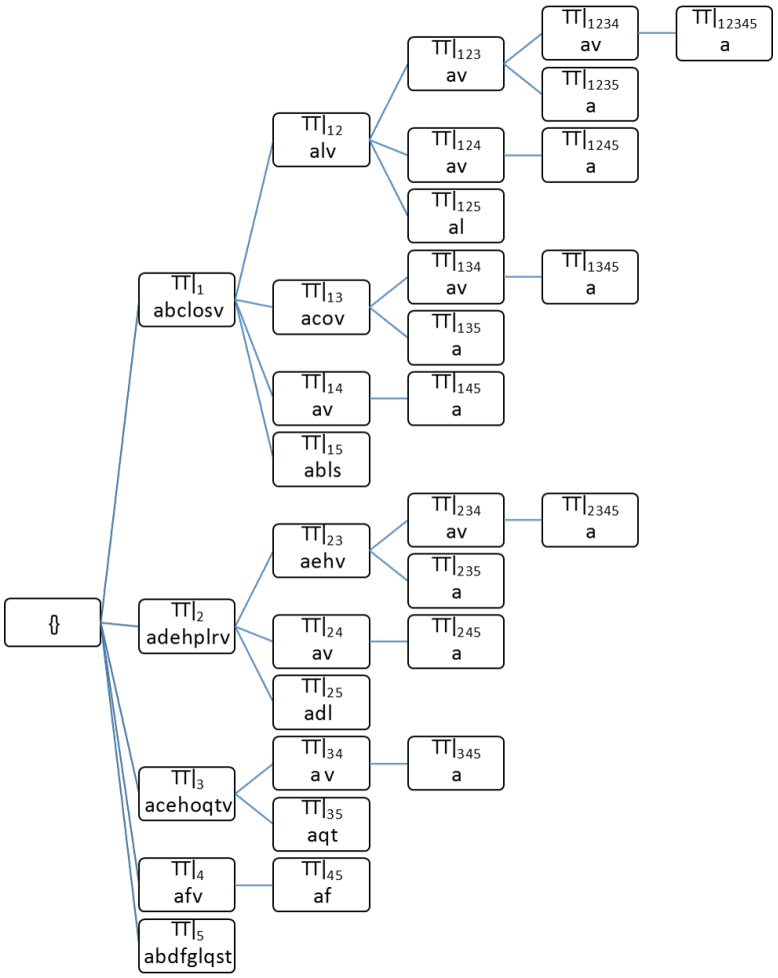

FIGURE 4.2: The transaction enumeration tree of the running example dataset in Figure 4.1a. For the sake of clarity, no pruning rules are applied to the tree.

2. The conditional transposed table $TT|_X$, used to obtain the itemset associated to the node and, through the remaing tids, determine how and if the node should be expanded
3. The set of itemsets found up to the current step of the tree search, used to avoid to process the same itemset twice (due to the enumeration tree architecture, the real support of the itemset is the one obtained the first time the itemset is processed in a depth-first exploration manner)
4. The enforced minimum support threshold (*minsup*), used to decide if the itemset is a frequent closed itemset

Based on the theorems reported in [23], if the itemset I associated with the current node is a frequent closed itemset then I is included in the frequent closed itemset set. Moreover, by exploiting the analysis performed on the current node, part of the remaining search space (i.e., part of the enumeration tree) can be pruned, to avoid the analysis of nodes that will never generate new closed itemsets. To this purpose, three pruning rules are applied on the enumeration tree, based on the evaluation performed on the current node and the associated transposed table $TT|_X$:

- **Pruning rule 1.** If the size of X, plus the number of distinct tids in the rows of $TT|_X$ does not reach the minimum support threshold, the subtree rooted in the current node is pruned.
- **Pruning rule 2.** If there is any tid tid_i that is present in all the tidlists of the rows of $TT|_X$, tid_i is deleted from $TT|_X$. The number of discarded tids is updated to compute the correct support of the itemset associated with the pruned version of $TT|_X$.
- **Pruning rule 3.** If the itemset associated with the current node has been already encountered during the depth first search, the subtree rooted in the current node is pruned because it can never generate new closed itemsets.

The tree search continues in a depth first fashion moving on the next node of the enumeration tree. More specifically, let tid_l be the lowest tid in the tidlists of the current $TT|_X$, the next node to explore is the one associated with $X' = X \cup \{tid_l\}$.

Among the three rules mentioned above, pruning rule 3 assumes a global knowledge of the enumeration tree explored in a depth first manner. This, as detailed in section 4.4, is very challenging in a distributed environment that adopts a shared-nothing architecture, like the one we address in this work.

4.4 The PaMPa-HD algorithm

In this section we describe the new algorithm, called PaMPa-HD, proposed in this paper. Specifically, we describe how PaMPa-HD parallelizes the itemset mining process and applies the pruning rules discussed in Section 4.3 in a parallel environment. Furthermore, we discuss how, through an ad-hoc synchronization phase, PaMPa-HD achieves a good load balancing and robustness to memory issues.

As discussed in the previous section, given the complete enumeration tree (see Figure 4.2), the centralized Carpenter algorithm extracts the whole set of closed itemsets

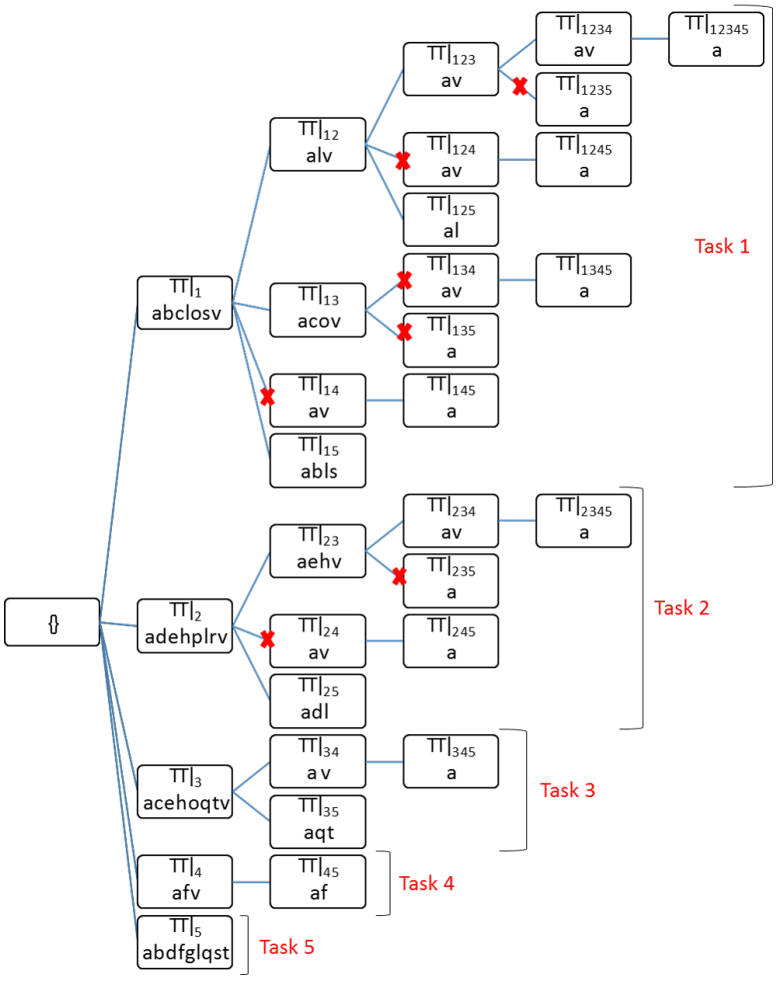

FIGURE 4.3: Running toy example: each node expands a branch of the tree independently. For the sake of clarity, pruning rule 1 and 2 are not applied. The pruning rule 3 is applied only within the same task: the small crosses on the edges represent pruned nodes due to local pruning rule 3, e.g. the one on node {2 4} represents the pruning of node {2 4}.

by performing a depth first search (DFS) of the tree. Differently, in order to parallelize the mining process, the PaMPa-HD algorithm splits the depth first search process in a set of (partially) independent sub-processes, that autonomously evaluate sub-trees of the search space. Specifically, the whole problem can be split by assigning each subtree rooted in $TT|_X$, where X is a single transaction id in the initial dataset, to an independent sub-process. Each sub-process applies the centralized version of Carpenter on its conditional transposed table $TT|_X$ and extracts a subset of the final closed itemsets. The subsets of closed itemsets mined by each sub-process are merged to compute the whole closed itemset result. Since the sub-processes are independent, they can be executed in

parallel by means of a distributed computing platform, e.g., Hadoop. Figure 4.3 shows the application of the proposed approach on the running example. Specifically, five independent sub-processes are executed in the case of the running example, one for each row (transaction) of the original dataset. The crosses on the nodes represent the local pruning within each parallel task. Partitioning the enumeration tree in sub-trees allows processing bigger enumeration trees with respect to the centralized version. However, this approach does not allow fully exploiting pruning rule 3 because each sub-process works independently and is not aware of the partial results (i.e., closed itemsets) already extracted by the other sub-processes. Hence, each sub-process can only prune part of its own search space by exploiting its "local" closed itemset list, while it cannot exploit the closed itemsets already mined by the other sub-processes. For instance, Task T2 in Figure 4.3 extracts the closed itemset av associated with node $TT|_{2,3,4}$. However, the same closed itemset is also mined by T1 while evaluating node $TT|_{1,2,3}$. In the centralized version of Carpenter, the duplicate version of av associated with node $TT|_{1,2,4}$ is not generated because $TT|_{1,2,4}$ follows $TT|_{1,2,3}$ in the depth first search, i.e., the tasks are serialized and not parallel.

Since pruning rule 3 has a high impact on the reduction of the search space, its inapplicability leads to a negative impact on the execution time of the distributed algorithm (see Section 4.5 for further details). To address this issue, we share partial results among the sub-processes. Each independent sub-process analyzes only a part of the search subspace. Then, when a maximum number of visited nodes is reached, the partial results are synchronized through a synchronization phase. Of course, the exploration of the tree finishes also when the subspace has been completely explored.

Specifically, the sync phase filters the partial results (i.e., nodes of the tree still to be analyzed and found closed itemsets) globally applying pruning rule 3. The pruning strategy consists of two phases. In the first one, all the transposed tables and the already found closed itemsets are analyzed. The transposed tables and the closed itemsets related to the same itemset are grouped together in a bucket. For instance, in our running example, each element of the bucket B_{av} can be:

- a frequent closed itemset av extracted during the subtree exploration of the node $TT_{3,4}$,
- a transposed table associated to the itemset av among the ones that still have to be expanded (nodes $TT_{1,2,3}$ and $TT_{2,3,4}$).

We remind the readers that, because of the independent nature of the Carpenter subprocesses, the elements related to the same itemset can be numerous, because obtained in

different subprocesses. Please note that all the extracted closed itemsets come together with the tidlist of the node in which they have been extracted.

In the second phase, in order to respect the depth-first pruning strategy of the rule 3, for each bucket it is kept only the oldest element (transposed table or closed itemset) based on a depth-first order. The depth-first sorting of the elements can be easily obtained comparing the tidlists of the elements of the bucket. Therefore, in our running example from the bucket B_{av}, it is kept the node $TT_{1,2,3}$ (See Figure 4.5) . The transposed tables which are not pruned in this phase are then expanded to continue the enumeration tree exploration.

Afterwards, a new set of sub-processes is defined from the filtered results, starting a new iteration of the algorithm. In the new iteration, the Carpenter tasks process also the frequent closed itemsets obtained in the previous iteration, which are used to enrich the local memory of the task and enhance the effectiveness of the local pruning. The Carpenter tasks process the remaining transposed tables, that are expanded, as before, until the maximum number of processed tables is reached. In order to enhance the effectiveness of the pruning rules related to the local Carpenter task, the tables are processed in a depth-first order. After that, as before, in the synchronization phase, pruning rule 3 is applied. The overall process is applied iteratively by instantiating new sub-processes and synchronizing their results, until there are no nodes left. The application of this approach to our running example is represented in Figure 4.4, in which the small crosses represent the pruning related to the local state memory; and in Figure 4.5, in which the bigger crosses represent the pruning related to the synchronization phase. The table related to the itemset *av* associated with the tidlist/node {2, 3, 4} is pruned because the synchronization job discovers a previous table with the same itemset, i.e. the node associated with the transaction ids combination {1, 2, 3}. The use of this approach allows the parallel execution of the mining process, providing at the same time a very high reliability dealing with heavy enumeration trees, which can be split and pruned according to pruning rule 3. Of course, this architecture cannot deliver the same pruning efficiency characterizing the centralized implementation of Carpenter in which the complete tree depth-first exploration is known.

The introduction of the sync phase leads also to a better load balancing of the tasks. At each synchronization, the tables to process are redistributed among the tasks. Therefore, the task related to the first branches of the tree, which are the ones with more nodes than others, are splitted into several subtasks. In this way, as shown Section 4.5, we achieve a better exploitation of the resources.

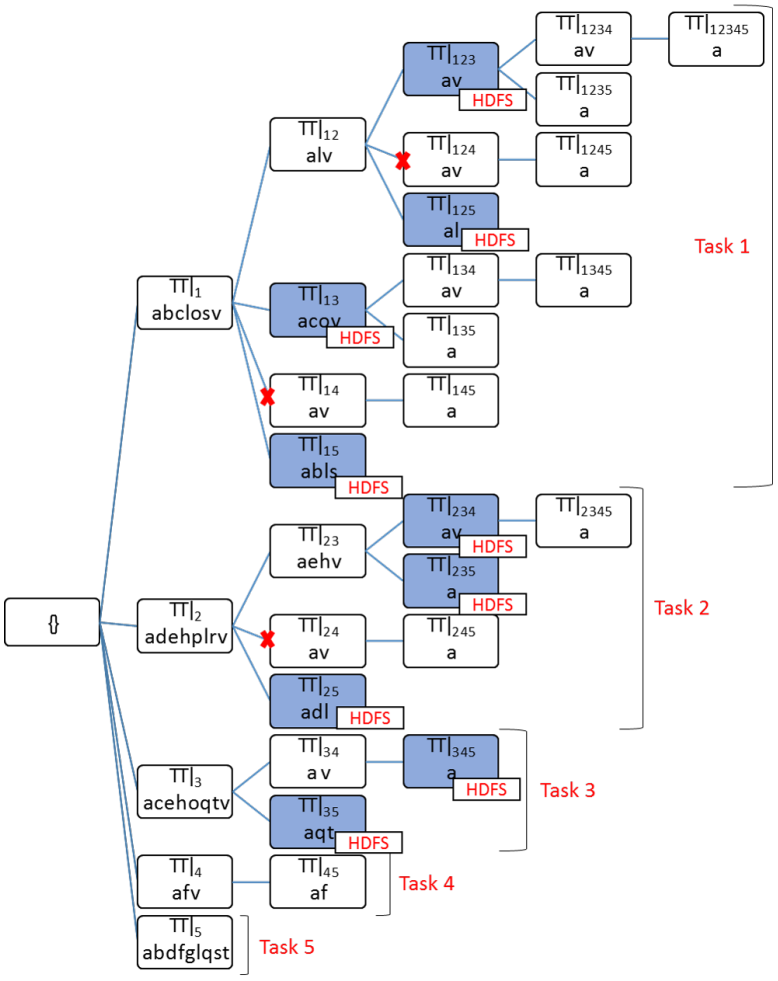

FIGURE 4.4: Execution of PaMPa-HD on the running example dataset. For the sake of clarity, pruning rules 1 and 2 are not applied. The dark nodes represent the nodes that have been written to HDFS in order to apply the synchronization job.

4.4.1 Implementation details

PaMPa-HD implementation uses the Hadoop MapReduce framework. The algorithm consists of three MapReduce jobs as shown in PaMPa-HD pseudocode (Algorithm 1).

The Job 1, whose pseudocode is reported in Algorithm 2, is developed to distribute the input dataset to the independent tasks, which will run a local and partial version of the Carpenter algorithm. The second job performs the synchronization of the partial results and exploits the pruning rules. At the end, the last job interleaves the Carpenter execution with the synchronization phase.

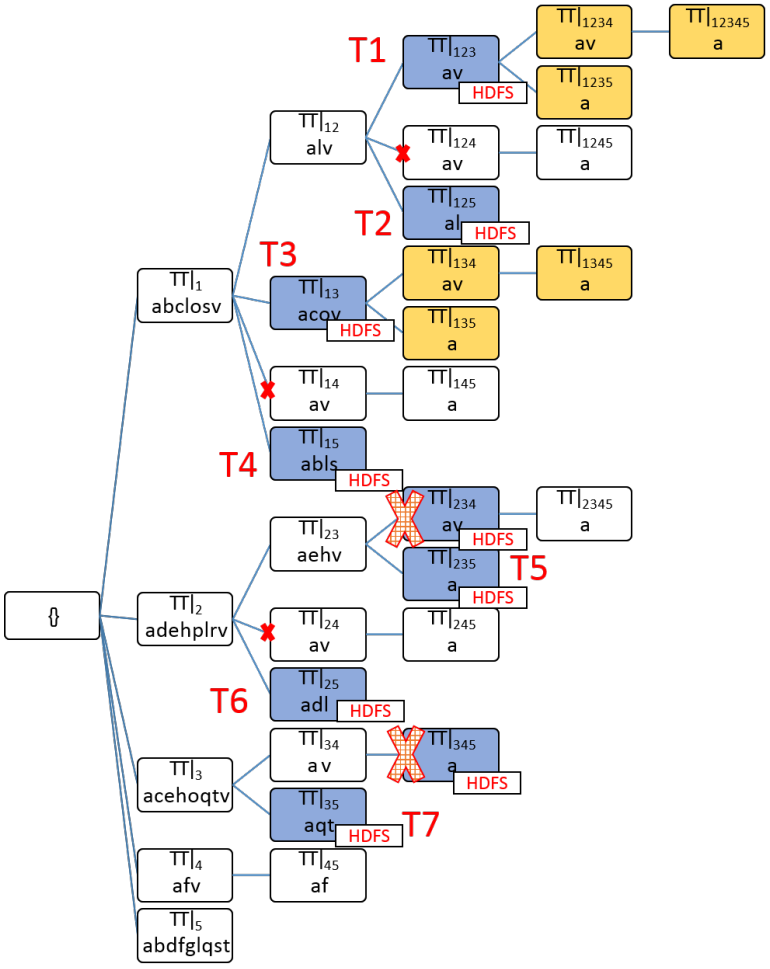

FIGURE 4.5: Execution of PaMPa-HD on the running example dataset. For the sake of clarity, pruning rules 1 and 2 are not applied. The big checked crosses on nodes represent the nodes which have been removed by the synchronization job, e.g., the one on node {2 3 4} represents the pruning of node {2 3 4}.

Job 1 (Algorithm 2). Each mapper is fed with a transaction of the input dataset, which is supposed to be in a vertical representation, together with the minsup parameter. As detailed in Algorithm 2, each transaction is in the form $item, tidlist$. For each transaction, the mapper performs the following steps. For each tid t_i of the input tidlist, given $TL_{greater}$ the set of tids $(t_{i+1}, t_{i+2}, ..., t_n)$ greater than the considered tid t_i (lines 2-7 in Algorithm 2).

- If $|TL_{greater}| >= minsup$, output a key-value pair <key= t_i; value= $TL_{greater}$, item>, then analyze t_{i+1} of the tidlist.
- Else discard the tidlist.

Algorithm 1 PaMPa-HD at a glance

```
1: procedure PaMPa-HD(minsup; initial TT)
2:     Job 1 Mapper: process each row of TT
         and send it to reducers, using as key values
         the tids of the tidlists
3:     Job 1 Reducer: aggregate TT|_x and run
         local Carpenter until expansion threshold is
         reached or memory is not enough
4:     Job 2 Mapper: process all the closed itemset
         or transposed tables from the previous job
         and send them to reducers
5:     Job 2 Reducer: for each itemset belonging
         to a table or a frequent closed, keep
         the eldest in a Depth First fashion
6:     Job 3 Mapper: process each closed itemset
         and TT|_x from the previous job.
         For the transposed tables run local Carpenter
         until expansion threshold is reached
7:     Job 3 Reducer: for each itemset belonging
         to a table or a frequent closed, keep
         the eldest in a Depth First fashion
8:     Repeat Job 3 until there are no more
         conditional tables
9: end procedure
```

For instance, if the input transaction is the tidlist of item b (b, 1 2 3) and minsup is 1, the mapper will output three pairs: <key=1; value=2 3, b>, <key=2; value=3, b>, <key=3; value=b>.

After the map phase, the MapReduce shuffle and sort phase aggregates the <key,value>pairs and delivers to reducers the nodes of the first level of the tree, which represent the transposed tables projected on a single tid (lines 10-13 in Algorithm 2). The tables in Figure 4.6 illustrate the processing of a row of the initial Transposed representation of D. Given that each key matches a single transposed table TT_X, each reducer builds the transposed tables with the tidlists contained in the "value" fields.

From this table, a local Carpenter routine is run (line 14 in Algorithm 2). Carpenter recursively processes a transposed table expanding it in a depth-first manner (see Section 4.3 for further details). However, the local Carpenter routine stops when the number of processed transposed tables is over the given maximum expansion threshold. This allows periodically performing the synchronization among the parallel tasks and hence enforcing pruning rule 3. All the intermediate results of the local invocation of the Carpenter routine are written to HDFS (lines 15-17 in Algorithm 2).

During the local Carpenter process, the found closed itemsets and the explored branches are stored in memory in order to apply a local pruning. The closed itemsets are emitted as output at the end of the task, together with the tidlist of the node of the tree in which they have been found (lines 18-20 in Algorithm 2). This information is required by the synchronization phase in order to establish which element is the eldest in a depth first exploration, i.e., which element is visited first in a depth first exploration (e.g. the node associated with tidlist {1, 2, 3, 5} is eldest than the node associated with tidlist {2, 3,

item	tidlist
a	1,2,3,4,5

(A) Transposed representation of $\mathcal{D}$: tidlist of item a

key	value
1	2,3,4,5 |a
2	3,4,5 |a
3	4,5 |a
4	5 |a
5	- |a

(B) Emitted key-value entries from the example row in Table 4.1b

key	value
3	4,5 |a
3	- |c
3	- |e
3	- |h
3	- |o
3	5 |q
3	5 |t
3	4 |v

(C) key-value entries for key 3

$TT\|_{\{3\}}$	
item	tidlist
a	4,5
c	-
e	-
h	-
o	-
q	5
t	5
v	4

(D) $TT|_{\{3\}}$: composed with the received values

FIGURE 4.6: Job 1 applied to the running example dataset ($minsup = 1$): local Carpenter algorithm is run from the Transposed Table 4.6d.

4} in a depth-first exploration order).

Algorithm 2 Dataset distribution and local and partial Carpenter execution (Job 1)

1: **procedure** MAPPER($minsup; item_i; tidlist\ TL$)
2: **for** $j = 0$ to $|(TL)| - 1$ **do**
tidlist $TL_{greater}$: set of tids greater than the considered tid t_j.
3: **if** $|TL_{greater}| \geq minsup$ **then**
4: output <key= t_j; value= $TL_{greater}$, item>
5: **else** Break
6: **end if**
7: **end for**
8: **end procedure**
9: **procedure** REDUCER($key = tid\ X, value = tidlists\ TL[\,]$)
10: Create new transposed table $TT|_X$
11: **for each** tidlist TL_i of $TL[\,]$ **do**
12: add TL_i to $TT|_X$ (populate the transposed table)
13: **end for**
14: Run $Carpenter(minsup; TT|_X; max\_exp)$
15: **for each** transposed table I found but not processed **do**
16: $Output <$ $itemset$; $tidlist + TransposedTable\ I\ rows >$
17: **end for**
18: **for each** frequent closed itemset found **do**
19: $Output(<$ $itemset; tidlist + support >)$
20: **end for**
21: **end procedure**

Job 2 (Algorithm 3). The synchronization phase is a straightforward MapReduce job in which mappers input is the output of the previous job: it is composed of the closed

frequent itemsets found in the previous Carpenter tasks and intermediate transposed tables that still have to be expanded. The itemsets are associated to their minsup and the tidlist related to the node of the tree in which they have been found; the transposed tables are associated to the table content, the corresponding itemset and the table tidlist.

- For each table, the mappers output a pair of the form:
  <key=itemset; value=tidlist,table_rows>(lines 2 - 5 of Algorithm 3);
- for each itemset, the mappers output a pair in the form:
  <key=itemset; value=tidlist,minsup>(lines 6 - 11 of Algorithm 3).

The shuffle and sort phase delivers to the reducers the pairs aggregated by keys. The reducers, which match the buckets introduced in Section 4.4, compare the entries and emit, for the same key or itemset, only the oldest version in a depth first exploration (lines 15 - 21 of Algorithm 3). For instance, referring to our running example in Figure 4.5, in the reducer related to the itemset *av* are collected the entries related to the nodes T_{123} and T_{234}. Since the tidlist 123 is previous than 234 in a depth-first exploration order, the reducer keeps and emits only the entry related to the node T_{123}. With this design, the redundant tables that can be obtained due to the independent nature of the Carpenter tasks, which can explore nodes related to the same itemsets, are discarded. This pruning is very similar to the one performed in centralized memory at the cost of a very MapReduce-like job (similar to a *WordCount* application).

Job 3 (Algorithm 4). This is a mixture of the two previous jobs. In the Map phase all the remaining tables are expandend by a local Carpenter routine. The Reduce phase, instead, applies the same kind of synchronization that is run in the synchronization job. The job has two types of input: transposed tables and frequent closed itemsets. The former are processed respecting a depth-first sorting and expanded until it is reached the maximum expansion threshold (line 5 of Algorithm 4). From that moment, the tables are not expanded but sent to the reducers (lines 6 - 8 of Algorithm 4). Please note that the tree exploration processing the initial transposed tables in a depth-first order is the same to a centralized architecture, enhancing the impact of pruning rule 3 (which strongly relies on this exploration manner). The latter (i.e. the frequent closed itemsets of the previous PaMPa-HD job) are processed in the following way. If in memory there is already an oldest depth-first entry of the same itemset, the closed itemset is discarded. If there is not, it is saved into memory and used to improve the local pruning effectiveness (lines 2 - 3). At the end of the task, all the frequent closed itemsets found are sent to the reducers, where the redundant elements are pruned. This job is iterated until all the transposed tables have been processed.

Thanks to the introduction of a global synchronization phase (Job 2 and Job 3 in Algorithms 3 and 4), the proposed PaMPa-HD approach is able to apply pruning rule 3 and handle high-dimensional datasets, otherwise not manageable due to memory issues.

Algorithm 3 Synchronization Phase and exploitation of the pruning rule 3 (Job 2)

```
1: procedure MAPPER(Frequent Closed itemset;
       Transposed table)
2:     if Input I is a table then
3:         itemset ← ExtractItemset(I)
4:         tidlist ← ExtractTidlist(I)
5:         Output(< itemset; tidlist + table I rows >)
6:     else (i.e. input I is a frequent closed Itemset)
7:         itemset ← ExtractItemset(I)
8:         tidlist ← ExtractTidlist(I)
9:         support ← ExtractSupport(I)
10:        Output(< itemset; tidlist + support >)
11:    end if
12: end procedure
13: procedure REDUCER(key = itemset;
       value = itemsets & tables T[ ])
14:    oldest ← null
15:    for each itemset or table T of T[ ] do
16:        tidlist ← ExtractTidlist(T)
17:        if tidlist previous of oldest in a Depth-First Search then
18:            oldest ← T
19:        end if
20:    end for
21:    Output(< itemset + oldest >)
22: end procedure
```

Algorithm 4 Interleaving of the Carpenter execution and synchronization phase (Job 3)

```
1: procedure MAPPER(Frequent Closed itemset; Transposed table)
2:     if Input I is a frequent closed itemset then
3:         save I to local memory
4:     else (i.e. input I is a Transposed Table)
5:         Run Carpenter(minsup; TT|_X; max_exp)
6:         for each transposed table I found but not processed do
7:             Output < itemset; tidlist + TransposedTable I rows >
8:         end for
9:     end if
10:    for each frequent closed itemset found do
11:        Output(< itemset; tidlist + support >)
12:    end for
13: end procedure
14: procedure REDUCER(key = itemset;
        value = itemsets & tables T[ ])
15:    oldest ← null
16:    for each itemset or table T of T[ ] do
17:        tidlist ← ExtractTidlist(T)
18:        if tidlist previous of oldest in a Depth-First Search then
19:            oldest ← T
20:        end if
21:    end for
22:    Output(< itemset + oldest >)
23: end procedure
```

4.5 Experiments

In this section, we present a set of experiments to evaluate the performance of the proposed algorithm. Firstly, we asses the impact on performance of the maximum expansion threshold (max_exp) parameter (Section 4.5.1). This phase is mandatory in order to tune-up the parameter configuration to compare the proposed approach with the state-of-the-art algorithms. Because the tuning of the parameter is not trivial, we discuss and experimentally evaluate some self-tuning strategies to automatically set the max_exp parameter and improve the performance (Section 4.5.2).

Next, we evaluate the speed of the proposed algorithm, comparing it with the state-of-the-art distributed approaches (Section 4.5.3). Finally, we experimentally analyze the impact of (i) the number of transactions of the input dataset (Section 4.5.4), (ii) the number of parallel tasks (Section 5.6.3), and (iii) the communication costs and load balancing behavior (Section 4.5.6).

Experiments have been performed on two real-world datasets. The first is the PEMS-SF dataset [65], which describes the occupancy rate of different car lanes of San Francisco bay area freeways (15 months worth of daily data from the California Department of

Transportation [66]). Each transaction represents the daily traffic rates of 963 lanes, sampled every 10 minutes. It is characterized by 440 rows and 138,672 attributes (6 x 24 x 963), and it has been discretized in equi-width bins, each representing 0.1% occupancy rate.

As mentioned, PaMPa-HD design is focused on scaling up in terms of number of attributes, being able to cope with high-dimensional datasets. For this reason, we have used a 100-rows version of the PEMS-SF dataset for all the experiments. However, we have used the full dataset and several down-sampled versions (in terms of number of rows) to measure the impact of the number of transactions on the performance of the algorithm (Section 4.5.4).

The second dataset is the Kent Ridge Breast Cancer [67], which contains gene expression data. It is characterized by 97 rows that represent patient samples, and 24,482 attributes related to genes. The attributes are numeric (integers and floating point). Data have been discretized with an equal-depth partitioning using 20 buckets (similarly to [23]). The discretized versions of the real datasets are publicly available at http://dbdmg.polito.it/PaMPa-HD/.

TABLE 4.1: Datasets

Dataset	Number of transactions	Number of different items	Number of items per transaction
PEMS-SF Dataset	440	8,685,087	138,672
Kent Ridge Breast Cancer Dataset	97	489,640	24,492

PaMPa-HD is implemented in Java 1.7.0_60 using the Hadoop MR API. The experiments were performed on a cluster of 5 nodes running Cloudera Distribution of Apache Hadoop (CDH5.3.1). Each cluster node is a 2.67 GHz six-core Intel(R) Xeon(R) X5650 machine with 32 Gbyte of main memory running Ubuntu 12.04 server with the 3.5.0-23-generic kernel.

4.5.1 Impact of the maximum expansion threshold

In this section we analyze the impact of the maximum expansion threshold (*max_exp*) parameter, which indicates the maximum number of nodes to be explored before a preemptive stop of each distributed sub-process is forced. This parameter, as already discussed in Section 4.4, strongly affects the enumeration tree exploration, forcing each

parallel task to stop before completing the visit of its sub-tree and send the partial results to the synchronization phase. This approach allows the algorithm to globally apply pruning rule 3 and reduce the search space. Low values of *max_exp* threshold increase the load balancing, because the global problem is split into simpler and less memory-demanding sub-problems, and, above all, facilitate the global application of pruning rule 3, hence a smaller subspace is searched. However, higher values allow a more efficient execution, by limiting the start and stop of distributed tasks (similarly to the context switch penalty) and the synchronization overheads. Above all, higher values enhance the pruning effect of the state centralized memory. In order to assess the impact of the expansion threshold parameter, we have performed two sets of experiments. In the first one we perform the mining on the PEMS-SF (100 transactions) dataset with minsup = 10, by varying *max_exp* from 100 to 100,000,000. The minsup value has been empirically selected to highlight the different performance related to different values (trivial mining would be overwhelmed by overhead costs of the MapReduce framework). In Figure 4.7 are shown the results in terms of execution time and number of iterations (i.e., the number of jobs)[2]. It is clear how the *max_exp* parameter can influence the performance, with wall-clock times that can be doubled with different configurations. The best performance in terms of execution time is achieved with a maximum expansion threshold equal to 10,000 nodes. With lower values, the execution times are slightly longer, while there is an evident performance degradation with higher *max_exp* values. This result highlights the importance of the synchronization phase. Increasing the *max_exp* parameter makes the number of iterations decreasing, but more useless tree branches are explored, because pruning rule 3 is globally applied less frequently. Lower values of *max_exp*, instead, raising the number of iterations, introduce a slight performance degradation caused by iterations overheads.

The same experiment is repeated with the Breast Cancer dataset and a minsup value of 5. As shown in Figure 4.8, even in this case, the best performances are achieved with *max_exp* equal to 10,000. In this case, differences are more significant with lower *max_exp* values, although with a non-negligible performance degradation with higher values.

The value of *max_exp* impacts also the load balancing of the distributed computation among different nodes. With low values of *max_exp*, each task explores a smaller enumeration sub-tree, decreasing the size difference among the sub-trees analyzed by different tasks, thus improving the load balancing. Table 4.2 reports the minimum and the maximum execution time of the mining tasks executed in parallel for both the

[2]Please note that in all the experiments, for the sake of clarity, the confidence intervals (obtained after a sufficient number of executions and with complementary level of significance of 95%) are omitted from the graphs.

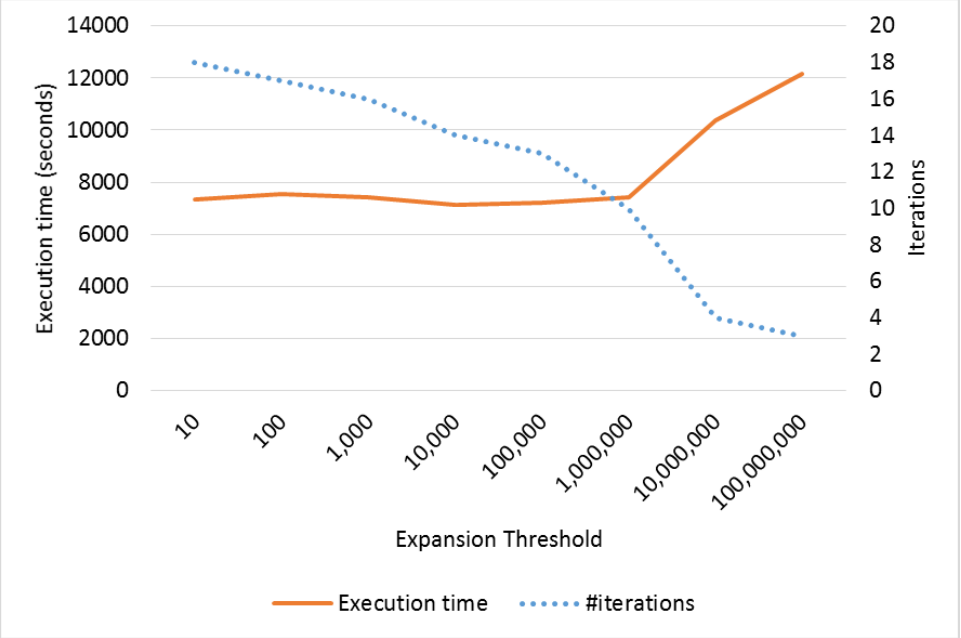

FIGURE 4.7: Execution time and number of iterations for different *max_exp* values on PEMS-SF dataset with *minsup*=10.

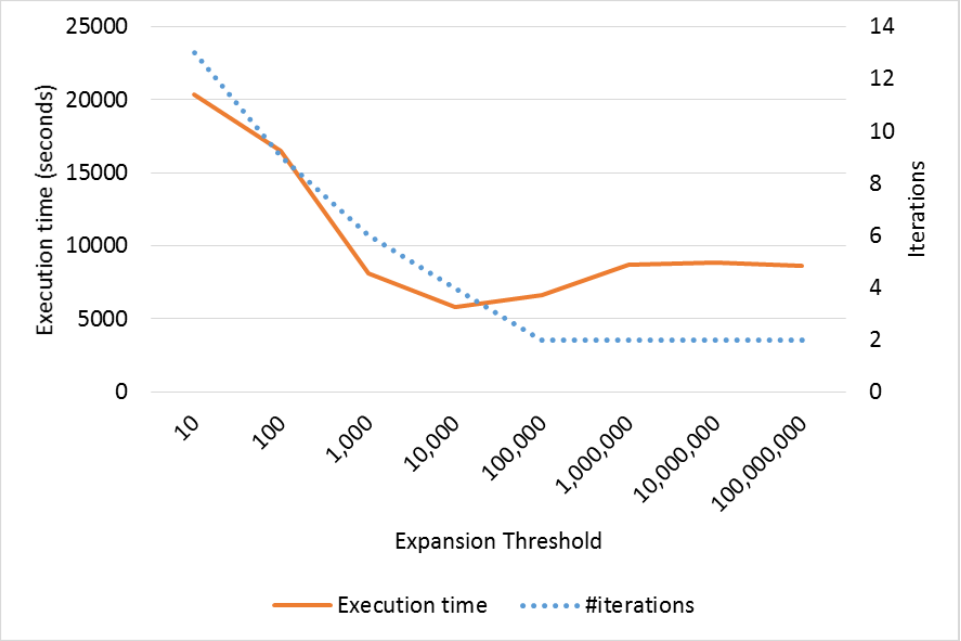

FIGURE 4.8: Execution time and number of iterations for different *max_exp* values on Breast Cancer dataset with *minsup*=5.

datasets and for two extreme values of *max_exp*. The load balance is better for the lowest value of *max_exp*.

The *max_exp* choice has a non-negligible impact on the performances of the algorithm. However, as demonstrated by the curves in Figures 4.7 and 4.8, it is very dependent

TABLE 4.2: Load Balancing

	Task execution time Breast Cancer		Task execution time PEMS-SF	
Maximum expansion threshold	Min	Max	Min	Max
100,000,000	7 m	2h 16m 17s	44s	2h 20m 28s
10	6m 21s	45m 16s	6s	2m 24s

on the use case and distribution of the data. In the next subsection we introduce and motivate some tuning strategies related to *max_exp*.

4.5.2 Self-tuning strategies

This section introduces some heuristic strategies related to the *max_exp* parameter. The aim of this experiment is to identify a heuristic technique able to improve the performances of the algorithm and easily configure the algorithm parameter. The heuristic consists in the automatic modification, inside the mining process, of the *max_exp* parameter, without requiring the user to manually tune it. To introduce the techniques, we provide motivations behind their design in the following. Because of the enumeration tree structure, the first tables of the tree are the most populated. Each node, in fact, is generated from its parent node as a projection of the parent transposed table on a tid. In addition, the first nodes are, in the average, the ones generating more sub-branches. By construction, their transposed table tidlists are, by definition, longer than the ones of their children nodes. This increases the probability that the table could be expanded. For these reasons, the tables of the initial mining phase are the ones requiring more resources and time to be processed. On the other hand, the number of nodes to be processed by each local Carpenter iteration tends to increase with the number of iterations. Still, this factor is mitigated by (i) the decreasing size of the tables and (ii) the eventual end of some branches expansion (i.e. when there are not more tids in the node transposed table). These reasons motivated us to introduce four strategies (Table 4.3) that assume a maximum expansion threshold which is increased with the number of iterations. These strategies start with very low values in the initial iterations (i.e. when the nodes require a longer processing time) and increase *max_exp* during the mining phases.

Strategy #1 is the simplest: *max_exp* is increased with a factor of X at each iteration. For instance, if *max_exp* is set to 10, and X is set to 100 at the second iteration it is raised to 1000 and so on. In addition to this straightforward approach, we leverage information about (i) the execution time of each iteration and the (ii) pruning effect

(i.e. the percentage of transposed tables / nodes that are pruned in the synchronization job).

The aim of the *strategy #2* is balancing the execution times among the iterations, trying to avoid a set of very short final jobs. Specifically, *strategy #2* increases, at each iteration, the *max_exp* parameter with a factor of $X^{T_{old}/T_{new}}$, where T_{new} and T_{old} are, respectively, the execution times of the previous two jobs.

For *strategy #3*, we analyzed the pruning impact of the synchronization phase (i.e. the percentage of pruned table due to redundancy). An increasing percentage of pruned tables means that there are a lot of useless tables that are generated. Hence, this could suggest to limit the growth of the *max_exp* parameter. However, the pruning effect is an information which cannot be easily interpreted. In fact, an increasing trend of the pruning percentage is also normal, since the number of nodes that are processed increases exponentially. Given that our intuition is to rise the *max_exp* among the iterations, in *strategy #3*, we increase the *max_exp* parameter with a factor $X^{Pr_{old}/Pr_{new}}$, given Pr_{new} and Pr_{old} the relative number of pruned tables in the previous two jobs. In this way, when the pruning impact increases ($Pr_{new} \geq Pr_{old}$), the growth of *max_exp* is slowed.

Finally, *strategy #4* is inspired by the congestion control of TCP/IP (a data transmission protocol used by many Internet applications [68]). This strategy, called "Slow Start", assumes two ways for growing the window size (i.e. the number of packets that are sent without congestion issues): an exponential one and a linear one. In the first phase, the window size is increased exponentially until it reaches a threshold ("ssthresh", which is calculated from some empirical parameters such as Round Trip Time value). From that moment, the growth of the window becomes linear, until a data loss occurs. In *strategy #4*, the *max_exp* is handled like the congestion window size.

In our case, we just inherit the two growth factor approach. Therefore, our "slow start" strategy consists in increasing the *max_exp* of a factor of X ($X \geq 10$) until the last iteration reaches an execution time greater than a given threshold. After that, the growth is more stable, increasing the parameter of a factor of 10. Please note that we have fixed the threshold to the execution time of the first two jobs (Job 1 and Job 2). These jobs, for the architecture of our algorithm, consists of the very first Carpenter iteration. They are quite different than the others since the first Mapper phase builds the initial projected transposed tables (first level of the tree) from the input file. This choice is consistent with our initial aim, that is to normalize the execution times of the last iterations which are often shorter than the first ones.

TABLE 4.3: Strategies

Strategy #1(X)	Constant growth of the parameter	Increasing at each iteration with a factor of X
Strategy #2(X)	Job balancing via execution time analysis	Increasing at each iteration with a factor of $X^{T_{old}/T_{new}}$
Strategy #3(X)	Job balancing via pruning impact analysis	Increasing at each iteration with a factor of $X^{Pr_{old}/Pr_{new}}$
Strategy #4	Slow start	Fast increase with a factor of X, slow increase with a factor of 10

TABLE 4.4: Strategies performance

Strategies	PEMS-SF	Breast Cancer
Strategy #1	-6.48% (X = 10)	-19.03% (X = 100,000)
Strategy #2	–3.73% (X = 1,000)	-0.02% (X = 10,000)
Strategy #3	-4.42% (X = 100)	+1.59% (X = 100)
Strategy #4	+9.39% (X = 100)	-16.17% (X = 1,000)

Strategy #1 is the one achieving the best performances for both the datasets. Table 4.4 reports the best performance for each strategy, in terms of relative performance difference with the best results obtained with a fixed *max_exp* parameter. For PEMS-SF dataset, even *strategies #2 and #3* are able to achieve positive gains. For Breast Cancer dataset *strategy #1* is the best, followed by *strategy #4*: these are the only ones achieving significant positive gain over the fixed *max_exp* approach. All the strategies are evaluated with X from 10 to 100,000.

As shown in Table 4.4, the results among the datasets are quite different. It is clear that Breast Cancer data distribution better fits the fast growth of the parameter, as shown by the better results with respect to the PEMS-SF dataset. The benefits of the growth of the *max_exp* parameter with PEMS-SF dataset are, indeed, limited. The reason behind this behavior is related to the data distribution. With PEMS-SF dataset, the mining process generates more intermediate results. In this scenario, a more frequent synchronization phase delivers more benefits with respect to the Breast Cancer dataset. The analysis is confirmed also by the best values of X with the two datasets. Breast Cancer experiments are characterized by a higher increase factor than the ones related to PEMS-SF dataset.

Since the best performance is achieved with values of 10 and 100,000 respectively for

PEMS-SF and Breast Cancer datasets (improvement of almost 6% and 20%), we will use this configuration for the experiments comparing PaMPa-HD with other distributed approaches.

4.5.3 Execution time

Here we analyze the efficiency of PaMPa-HD by comparing it with three distributed state-of-the-art frequent itemset mining algorithms:

1. Parallel FP-growth [51] available in Mahout 0.9 [69], based on the FP-growth algorithm [36] [3]
2. DistEclat [39], based on the Eclat algorithm [37]
3. BigFIM [39], inspired from the Apriori [38] and DistEclat

This set of algorithms represents the most cited implementations of frequent itemset mining distributed algorithms. All of them are Hadoop-based and are designed to extract the frequent closed itemsets (DistEclat and BigFIM actually extract a superset of the frequent closed itemsets). The parallel implementation of these algorithms has been aimed to scale in the number of transactions of the input dataset. Therefore, they are not specifically developed to deal with high-dimensional datasets as PaMPa-HD. The algorithms have been already discussed in detail in Section 3.4.

Even in this case, the frameworks are compared over the two real dataset (PEMS-SF and Breast Cancer datasets) The experiments are aimed to analyze the performance of PaMPa-HD with respect to the best-in-class approaches in high-dimensional use-cases. The first set of experiments has been performed with the 100-rows version PEMS-SF dataset [65] and minsup values 35 to 5.[4]

As shown in Figure 4.9, in which minsup axis is reversed to improve readability, PaMPa-HD is the only algorithm able to complete all the mining task to a minsup value of 5 rows or 5%. All the approaches show similar behaviors with high minsup values (from 30 to 35). With a minsup of 25, PFP shows a strong performance degradation, being not able to complete the mining. In a similar way, BigFIM shows a performance degradation with a minsup of 20, running out of memory with a minsup of 15. DistEclat, instead,

[3] The Spark MLlib [70] implementation has not been included into the evaluation because it extracts all the frequent itemsets and not just the closed ones.

[4] The algorithms parameters, which will be introduced in Section **??**, has been set in the following manner. PFP has been set to obtain all the closed itemsets; the prefix length of the first phase of BigFIM and DistEclat, instead, has been set to 3, as suggested by the original paper [39], when possible (i.e. when there were enough 3-itemsets to execute also the second phase of the mining).

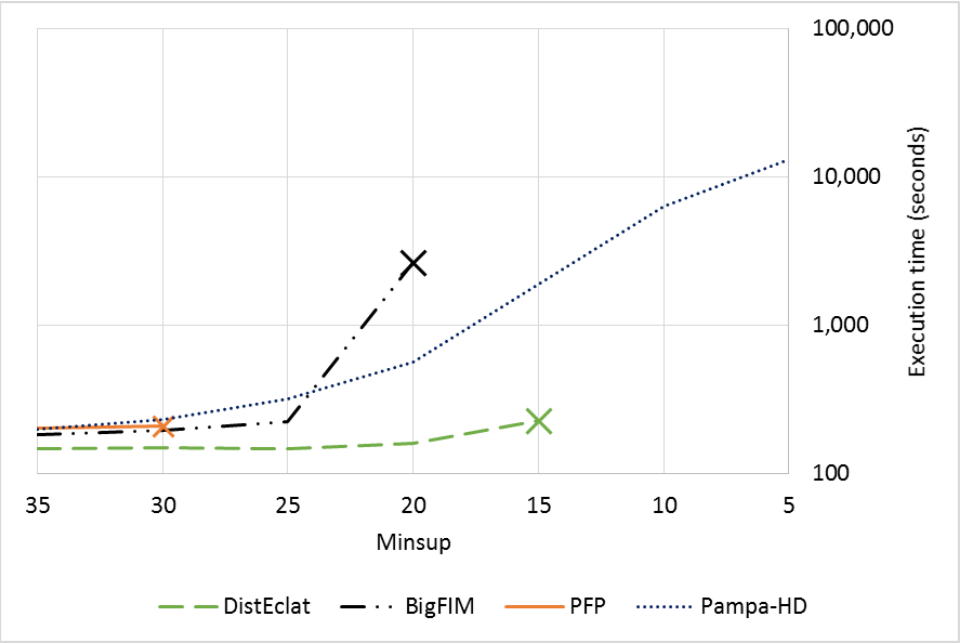

FIGURE 4.9: Execution time for different Minsup values on the PEMS-SF dataset (100-rows).

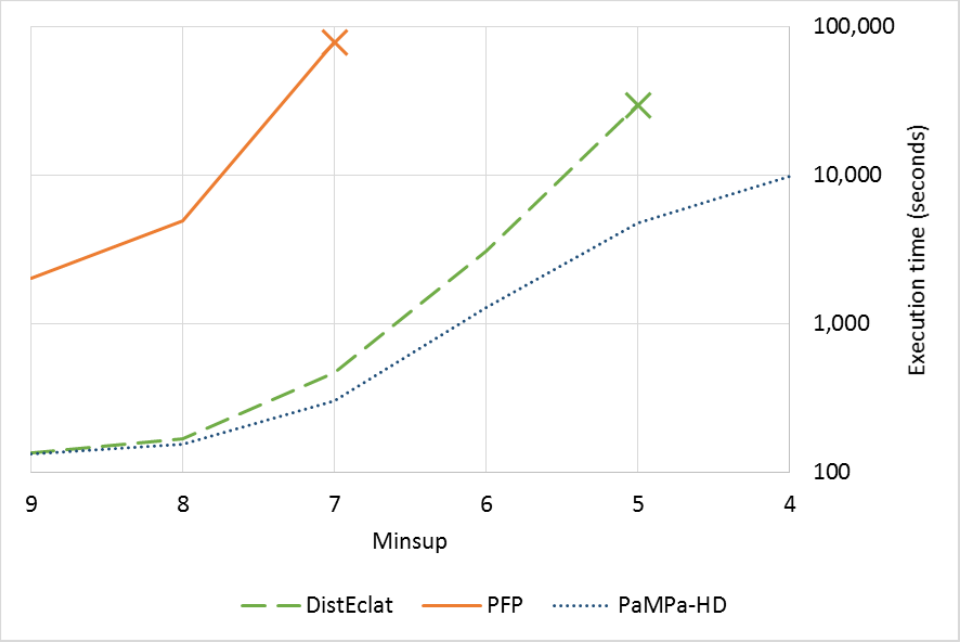

FIGURE 4.10: Execution time for different Minsup values on the Breast Cancer dataset.

shows very interesting execution time until running out of memory with a minsup of 10. PaMPa-HD, even if slower than DistEclat with minsup values from 25 to 15, is able to complete all the tasks.

The second set of experiments are performed with the Breast Cancer dataset [67]. As reported in Figure 4.10 (Even in this case, minsup axis is reversed to improve readability, the minsup is absolute), PaMPa-HD is the most reliable and fast approach. This time,

BigFIM is not able to cope even with the highest minsup values, while PFP shows very slow execution times and runs out of memory with a minsup value of 6. DistEclat is able to achieve good performances but is always slower than PaMPA-HD (with a minsup value equal to 4, it is not able to complete the mining within several days of computation). From these results, we have seen how traditional best-in-class approaches such as BigFIM, DistEclat and PFP are not suitable for high-dimensional datasets. They are slow and/or not reliable when coping with the curse of dimensionality. PaMPa-HD, instead, demonstrated to be most suitable approach with datasets characterized by a high number of items and a small number of rows. After the comparison with the state of the art distributed frequent itemset mining algorithms, the next subsections will experimentally analyze the behavior of PaMPa-HD with respect to the number of transactions, number of independent tasks, communication costs and load balancing.

4.5.4 Impact of the number of transactions

This set of experiments measures the impact of the number of transactions on PaMPa-HD performances. To this aim, the PEMS-SF datasets will be used in three versions (100-rows, 200-rows and full). The algorithm is very sensitive to this factor: the reasons are related to its inner structure. In fact, the enumeration tree, for construction, is strongly affected by the number of rows. A higher number of rows leads to:

1. A higher number of branches. As shown in the example in Figure 4.2, from the root of the tree, it is generated a new branch for each tid (transaction-id) of the dataset.
2. Longer and wider branches. Since each branch explores its research subspace in a depth-first order, exploring any combination of tids, each branch would result with a greater number of sub-levels (longer) and a greater number of sub-branches (wider)

Therefore, the mining processes related to the 100-rows version and to the 200-rows or the full version of PEMS-SF dataset are strongly different. With a number of rows incremented by, respectively, 200% and more than 400%, the mining of the augmented versions of PEMS-SF dataset is very challenging for the enumeration-tree based PaMPa-HD. The performance degradation is resumed in Figure 4.11, where, for instance, with a minsup of 35%, the execution times related to the 100-rows and the full version of the PEMS-SF dataset differ of almost two orders of magnitude.

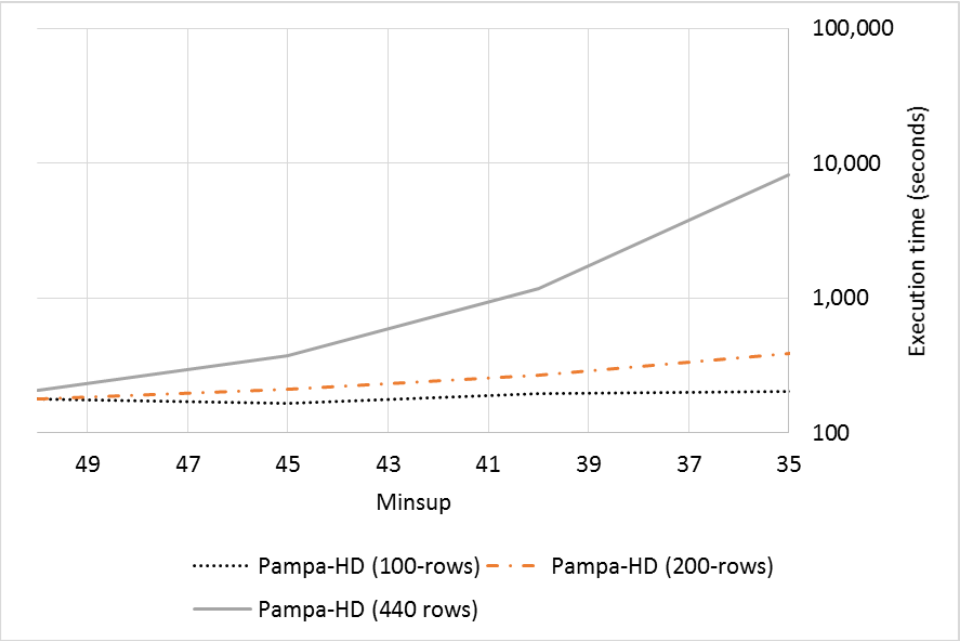

FIGURE 4.11: Execution times for different versions of PEMS-SF for PaMPa-HD.

The behavior and the difficulties of PaMPa-HD with datasets with an incremental number of rows, is, unfortunately, predictable. This algorithmic problem represents a challenging and interesting open issues for further developments.

4.5.5 Impact of the number of nodes

The impact of the number of independent tasks involved in the algorithm execution is a non-trivial issue. Adding a task to the computation would not only deliver more resources such as memory or CPU, but it also leads to split the chunk of the enumeration tree that is explored by each task. On the one hand, this means to reduce the search space to explore, lightening the task load. On the other hand, this reduces the state centralized memory and the impact of the related pruning. It can be interpreted as a trade-off between the benefits of the parallelism against the state. In Figure 4.12 and Figure 4.13, it is reported the behavior of PaMPa-HD with a mining process on the datasets PEMS-SF and Breast Cancer. The minsup values, respectively of 20 and 6, have been chosen in order to highlight the performance differences among the different degree of parallelism and datasets. Interestingly, the mining on PEMS-SF dataset is less sensitive to the number of reducers, with an execution time that is just halved when the independent tasks included in the computation pass from 1 to 17. The experiment of Breast Cancer instead, Figure 4.13, shows a stronger performance gain. As before, the behavior is related to the dataset data distribution which causes the PEMS-SF mining process generating more intermediate tables. In this case, the advantages related to

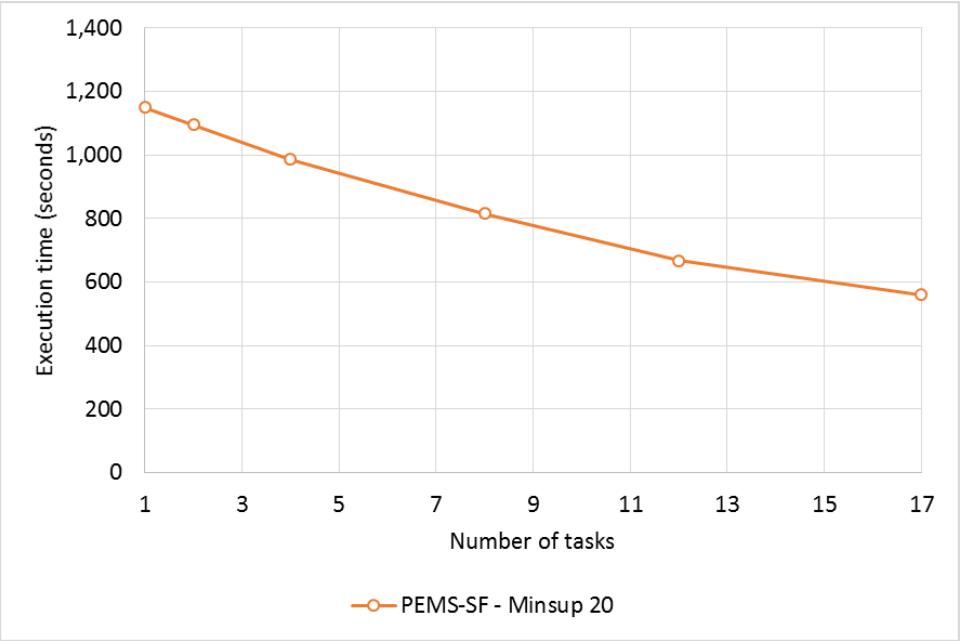

FIGURE 4.12: Execution times for PEMS-SF dataset with different number of parallel tasks.

additional independent nodes into the mining is mitigated by the loss of state in the local pruning phase inside the nodes. With additional nodes, each node is pushed to a smaller exploration of the search space, decreasing the effectiveness of the local pruning. These specific results recall a very popular open issue in distributed environments. In problems characterized by any kind of "state" benefit (in this case, the local pruning inside the tasks), a higher degree of parallelism does not lead to better performance a priori.

4.5.6 Load Balancing and communication costs

The last analyses are related to the load balancing and the communication costs of the algorithm. These issues represent very important factor in such a distributed environment. Communication costs are among the main bottlenecks for the performance of parallel algorithms [71]. A bad-balanced load among the independent tasks leads to few long tasks that block the whole job.

PaMPa-HD, being based on the Carpenter algorithm, mainly consists on the exploration of an enumeration tree. The basic idea behind the parallelization is to explore the main branches of the tree independently within parallel tasks (Figure 4.3). For this reason, each task needs the information (i.e. transposed tables) related to its branch expansion. The ideal behavior of a distributed algorithm would be to distribute the least amount of data, avoiding redundant informations as much as possible. The reason is that network

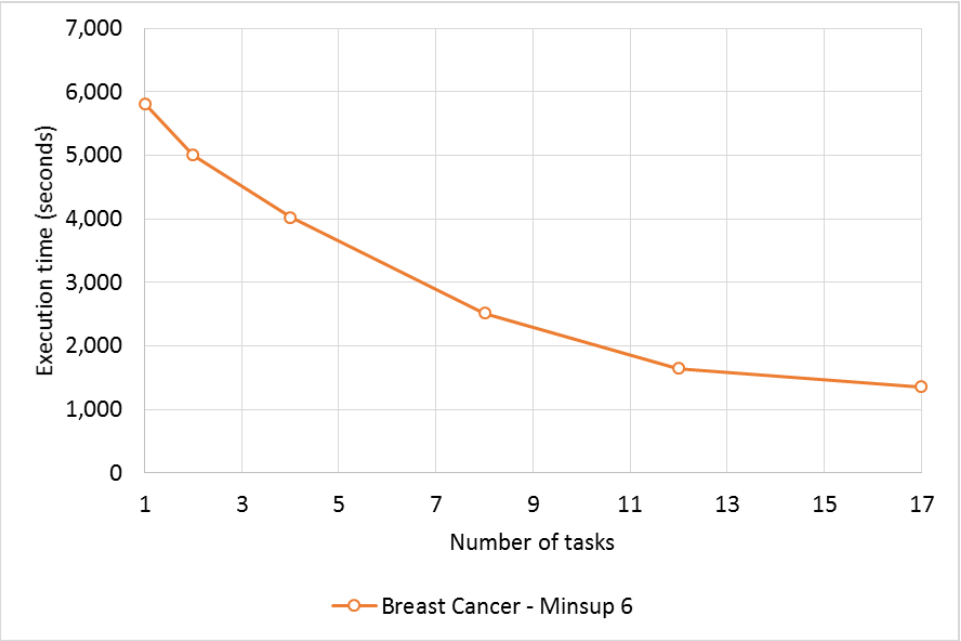

FIGURE 4.13: Execution times for Breast Cancer dataset with different number of parallel tasks.

communications are very costly in a Big Data scenario. Unfortunately, the structure of the enumeration tree of PaMPa-HD assumes that some pieces of data of the initial dataset is sent to more than one task. For instance, some data related to nodes $TT|_2$ and $TT|_3$ are the same, because from node $TT|_2$ will be generated the node $TT|_{2,3}$. This is an issue related to the inner structure of the algorithm and a full independence of the initial data for each branch cannot be reached.

In addition, the architecture of the algorithm, with its synchronization phase, increases the I/O costs. In order to prune some useless tables and improve the performance, the mining process is divided in more phases writing the partial results into HDFS. However, as we have already seen when studying the impact of *max_exp* (Figure 4.7 and Figure 4.8), in some cases additional synchronization phases lead to better execution times, despite their related overhead.

In Figure 4.14 and 4.15, the communication cost during a mining process is reported. The spikes are related to the shuffle phases, in which the redundant tables and closed itemsets are removed. The flat part of the curve between the spikes is longer in the case of the Breast Cancer dataset because of the adopted strategy. Its mining has been executed with a more aggressive increasing of the *max_exp* parameter (steps of 10 for PEMS-SF dataset, 10,000 for Breast Cancer dataset), which leads to a very long period without synchronization phases.

The load balancing is evaluated by comparing the execution time of the fastest and slowest tasks related to the iteration job in which this difference is strongest. The most

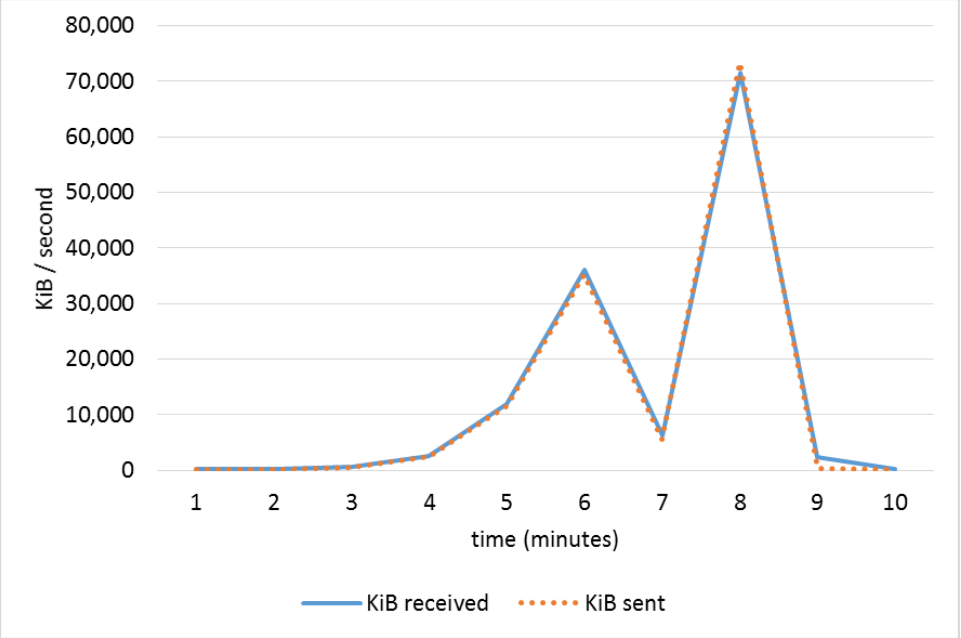

FIGURE 4.14: Received and sent data in the commodity cluster network during PEMS-SF dataset mining, minsup=20.

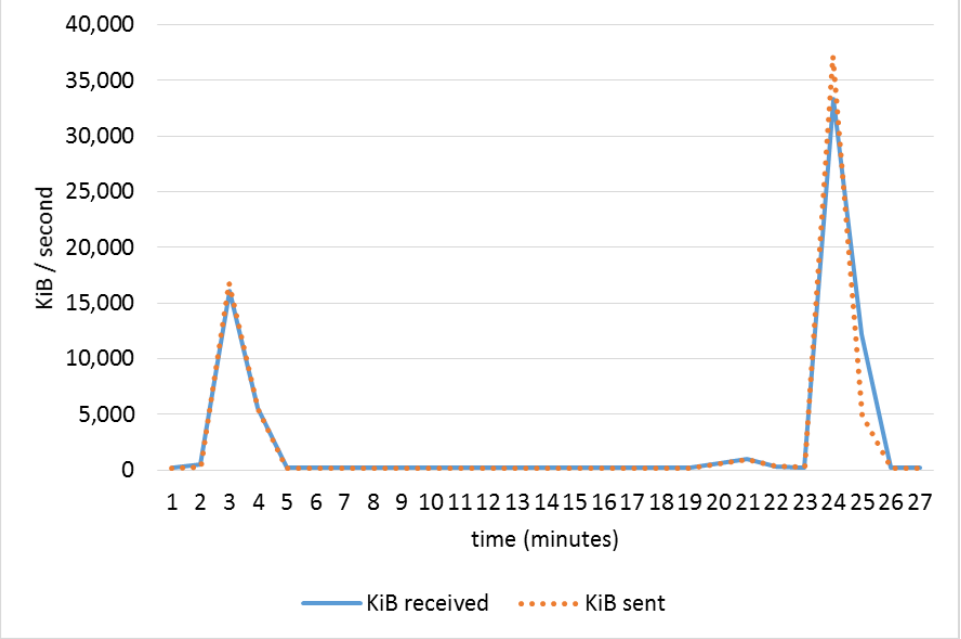

FIGURE 4.15: Received and sent data in the commodity cluster network during Breast Cancer dataset mining, minsup=6.

unbalanced phase of the job is, not surprisingly, the mapper phase of the Job 3. This job is iterated until the mining is complete and it is the one more affected by the increase of the *max_exp* parameter (iterations characterized by high *max_exp* value are likely characterized by long and unbalanced task). The difference among the fastest and the slowest mapper is shown in Table 4.5. It is clear that the mining on PEMS-SF dataset is more balanced among the independent tasks. Even in this case, the reason is the different

TABLE 4.5: Load Balancing

Dataset	Slowest Task Execution time	Fastest Task Execution time
PEMS-SF	3mins 58 sec	3mins 37sec
Breast Cancer	20mins 33sec	8mins 42sec

increment value in the Strategy #1 (10 for PEMS-SF dataset, 10,000 for Breast Cancer dataset). A slower *max_exp* increasing leads to more balanced tasks.

4.6 Applications

Since PaMPa-HD is able to process extremely high-dimensional datasets, it enriches the set of algorithm able to deal with datasets characterized by a very large variety of features (e.g. [72], [73]). Consequentely, many fields of applications which exploits frequent itemset to discover hidden correlations and association rules [74] could benefit of it. The first example is bioinformatics [75] and health environments: researchers in this domain often cope with data structures defined by a large number of attributes, which matches gene expressions, and a relatively small number of transactions, which typically represent medical patients or tissue samples. Furthermore, smart cities and computer vision applications are two important domains which can benefit from our distributed algorithm, thanks to their heterogeneous nature. Another field of application is the networking domain. Some examples of interesting high-dimensional dataset are URL reputation, advertisements, social networks and search engines. One of the most interesting applications, which we plan to investigate in the future, is related to internet traffic measurements. Currently, the market offers an interesting variety of internet packet sniffers like [56], [76]. Collected datasets, that include traffic flows in which the item are flow attributes ([40], [77], [78]), represent already a very promising application domain for data mining techniques, where PaMPa-HD can be efficiently exploited

4.7 Conclusion

This Chapter introduced PaMPa-HD, a novel frequent closed itemset mining algorithm able to efficiently parallelize the itemset extraction from extremely high-dimensional datasets. Experimental results show its good scalability and its efficient performance in dealing with real-world datasets characterized by up to 8 millions different items and, above all, an average number of items per transaction over hundreds of thousands, on

a small commodity cluster of 5 nodes. PaMPa-HD outperforms state-of-the-art algorithms, by showing a better scalability than all popular distributed approaches, such as PFP, DistEclat and BigFIM.

Further developments of the algorithm can be related to the analysis of the trade-off between the benefits of the scalability and the ones related to the local state. In addition, future works could analyze the introduction of better load balancing mechanisms. The increasing *max_exp* parameter introduced by the self-tuning strategies leads to a degradation of the load balancing between the parallel tasks of the job. As shown in Table 4.2, higher *max_exp* values decrease load balancing (i.e. only few tasks running), wasting the resources assigned to the tasks that are already complete. Forcing the synchronization phase after a fixed period of time would limit the amount of time in which the resources are not completely exploited. From the algorithmic point of view, this is not a loss, since the tables are expanded in a depth-first fashion. The last tables, hence, are the ones with highest probability to be pruned. This future development, therefore, would analyze the choice of the *time-out* which forces the synchronization phase.

4.8 Relevant pubblications

D. Apiletti, E. Baralis, T. Cerquitelli, P. Garza, P. Michiardi, and F. Pulvirenti, "Pampahd: A parallel mapreduce-based frequent pattern miner for high-dimensional data," in *IEEE ICDM Workshop on High Dimensional Data Mining (HDM)*, Atlantic City, NJ, USA, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7395755

D. Apiletti, T. Baralis, Elena Cerquitelli, P. Garza, P. Michiardi, and F. Pulvirenti, "A parallel map-reduce algorithm to efficiently support itemset mining on high dimensional data," *Submitted to Big Data Research - Under Review.*

Chapter 5

Frequent Itemset Mining in Distributed Scalable Frameworks

This Chapter of the dissertation will introduce (i) a big data mining framework and (ii) its utilization to extract a new type of itemsets, called misleading generalized itemsets. The comprehensive framework was initially designed to analyze network traffic logs [7] and provide users with a variety of network analytics services. After that, it has been extended to support a new real life scenario (Smart Cities environment) and to focus on a new type of itemset called misleading generalized itemsets [10],[8],[9]. The first use case taken into account to evaluate the effectiveness of the framework is related to network traffic analysis. In this field, important issues are communication profiling, anomaly or security threat detection, and recurrent pattern discovery. Traffic analyses are commonly performed on: (i) packet payloads, (ii) traffic metrics, or (iii) some statistical features computed on traffic flows. In the second use case, the proposed framework has been applied to analyze the traffic law infractions committed by the citizens of Turin, an important business and cultural center in northern Italy. Real infraction data is provided as open data by the Turin administration. The target of the analysis is to improve the efficiency of public services, the transparency of public administrations, and the awareness of the degree of civilization of urban people.

Especially for network traffic analysis, a significant research effort has been devoted to the application of data mining techniques. The proposed approaches address the discovery of significant correlations among data [79, 80], the extraction of knowledge useful for prediction [81], and the clustering of network data with similar properties [82]. However, due to the continuous growth in network speed, petabytes of data may be transferred through a network every day. These ”big data” collections stress the limits

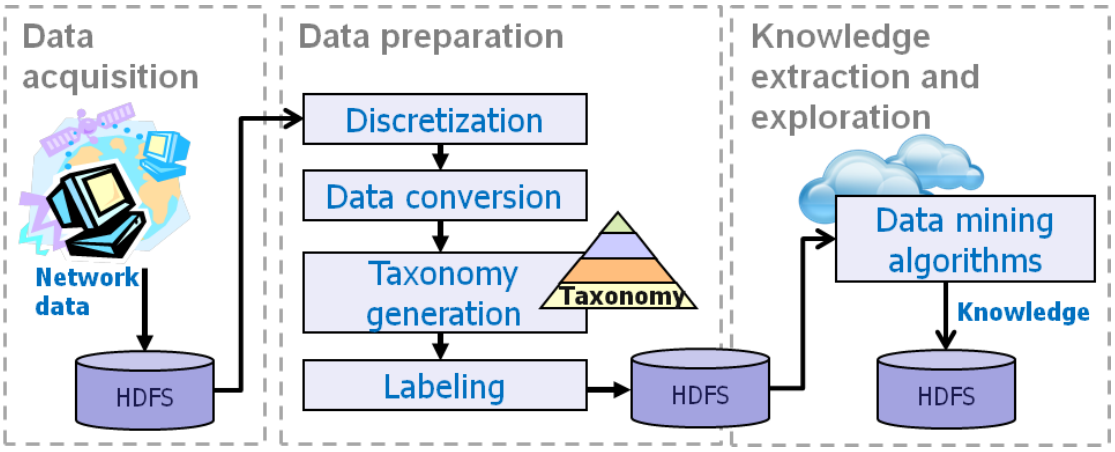

FIGURE 5.1: Architecture of NEMICO

of existing data mining techniques and thus they set new horizons for the design of innovative data mining approaches.

The Chapter is organized as follows. Section 5.1 presents NEMICO (NEtwork MIning in the CloUd), a data mining system focused on efficiently discovering interesting knowledge from Big network datasets by means of distributed approaches. After that, in Section 5.2, we will introduce an instance of the framework, named MGI-CLOUD (Misleading Generalized Itemset miner in the CLOUD), developed to mine misleading generalized itemsets. Section 5.3 overviews most relevant previous works while Section 5.4 states the problem addressed in this work. Section 5.5 presents the MGI-CLOUD architecture while an experimental evaluation of our approach is reported in Section 5.6. Finally, Section 5.7 draws conclusions and discusses future research directions.

5.1 The NEMICO architecture

NEMICO consists of a series of distributed MapReduce jobs related to different step of the knowledge discovery process. It ranges from network data acquisition to knowledge exploitation, as we will detail in the next chapters. In Figure 5.1 are shown the building blocks of the NEMICO architecture. To effectively support analysts in discovering different and interesting kinds of knowledge, a broad variety of data mining algorithms can be integrated in the system such as exploratory techniques (e.g., association rules, clustering) and prediction ones (e.g., classification and regression algorithms).

In this job pipeline, each job takes as input the result of one or more preceding jobs, performing a specific step of the data mining process. (each job is performed by one or more MapReduce tasks running on a Hadoop cluster).

5.1.1 Data acquisition and preprocessing

NeMiCo exploits passive traffic sniffing to acquire massive amounts of network traffic measurements and stores them in HDFS distributed file system. More details about the specific use case data preparation will be provided in Section **to add**.

To suit the raw data to each of the subsequent data mining step, are applied some data preprocessing steps to the input data. A brief description of the main data preparation steps is given below.

Discretization. Discretization concerns the transformation of continuous values into discrete ones. Since some data mining algorithms are unable to cope with continuously valued data, measurement values are discretized prior to running the algorithms. The discretization step can be performed either automatically by using established techniques [83] or semi-automatically by partitioning continuous value ranges into appropriate bins based on the prior knowledge about the measurement domains.

Data conversion. Data conversion entails the transformation of the raw data into the data format expected by the data mining algorithms to apply. It happens that algorithms are designed to handle only a subset of specific format. For example, most association rule mining algorithms are designed to cope with transactional data [83]. Hence, applying association rule mining algorithms requires the acquired data to be tailored to the transactional data format.

Taxonomy generation. The data mining process can be driven by semantics-based models (e.g., taxonomies or ontologies). These models, when available, are used to enrich the source data with multiple-level or multi-faceted information that would result in additional knowledge as output. For instance, a taxonomy, as shown in this Chapter for the use-cases taken into account, consists of set of 'is-a' hierarchies built over the data attributes. These structures are exploited to aggregate specific data values (e.g., the TCP ports) into meaningful higher-level categories. NeMiCo supports both the automatic taxonomy inference over a subset of specific network data attributes (e.g., port number, packet number) and the semi-automatic taxonomy construction.

Labeling. Supervised data mining techniques (e.g., classification) require the labeling of one data attribute as class label. Hence, if the data mining process comprises supervised analyses domain-experts have to specify the class attribute.

The current implementation of NeMiCo includes a first implementation of all the described activities as parallel map jobs.

5.1.2 Knowledge extraction and exploration

Knowledge extraction entails the application of data mining algorithms to find implicit, previously unknown, and potentially useful information from large volumes of network data. NeMiCo comprises novel data mining algorithms that contribute to a paradigm-shift in distributed data mining. The analytics algorithms entail (i) discovering underlying correlations among traffic data (e.g., multiple-level associations among data equipped with taxonomies), (ii) grouping traffic flows with similar properties (e.g., clustering), and (iii) extracting models useful for prediction (e.g., classification, regression).

As already mentioned, the current implementation of NeMiCo comprises Hadoop-based data mining algorithms focused on the extraction of interesting and multiple-level correlations among network data [8]. The next Section will describe the application of such comprehensive framework to the Network traffic and Smart Cities environments.

5.2 Misleading Generalized Itemsets

In this second part of the Chapter, as already mentioned, we will extend the discussed distributed network data mining framework to focus on an established pattern mining technique called generalized itemset extraction [84]. The obtained architecture framework has been named MGI-Cloud (Misleading Generalized Itemset miner in the Cloud) This technique has already been applied to data coming from different application domains (e.g., market basket analysis [84], network traffic data analysis [85], genetic data mining [86]). Generalized itemset mining entails discovering correlations among data at different abstraction levels. By exploiting a taxonomy (i.e., a set of is-a hierarchies) built over the analyzed data, frequent generalized itemsets, which represent recurrent co-occurrences among data items at different granularity levels, are extracted. These patterns are worth considering by domain experts to transform huge amounts of raw data into useful and actionable knowledge. However, a subset of peculiar high-level patterns should be analyzed separately during manual result inspection. More specifically, each generalized itemset has a correlation type which indicates the strength of the correlation between the corresponding items. Misleading Generalized Itemsets (MGIs) [87] are generalized itemsets whose correlations type is in contrast to those of most of their low-level descendant itemsets. These high-level patterns are worth considering for in-depth analysis because they are likely to represent misleading and thus potentially interesting situations. In [87] MGI extraction is performed in main memory on top of frequent level-sharing itemsets. Unfortunately, when coping with Big Datasets, a large number of itemsets is often generated at step (i) thus MGI extraction becomes a

challenging task. However, to the best of our knowledge, no attempt to mine MGIs on a distributed architecture has been made yet.

The remainder part of this Chapter presents MGI-Cloud (Misleading Generalized Itemset miner in the Cloud), an instance of the more general framework NeMiCo designed to efficiently mine MGIs on a distributed computing model. The experimental results show the effectiveness and efficiency of the MGI-Cloud architecture as well as they demonstrate its applicability to the analyzed use-cases.

5.3 Related work

In the last years, a relevant research effort has been devoted to large-scale itemset mining based on the MapReduce paradigm [88]. The goal is to propose itemset extraction algorithms that distribute data and computation across a distributed architecture to scale the mining process towards Big Data [39, 40, 51]. Unlike aforementioned papers, this works investigates the applicability of a generalized pattern mining technique on the MapReduce platform.

TO DO: still to be rephrased + add figure about etl workflow + make the figures and graphs consistent with the formers

The frequent generalized frequent itemset and association rule mining problems [84] have largely been studied by the data mining community. The firstly proposed approach [84] generates itemsets by considering for each item all its ancestors in the taxonomy. To avoid generating all the possible itemsets, the authors in [89, 90] proposed to push (analyst-provided) constraints into the mining process. In parallel, many algorithm optimizations and variations have been proposed [91, 92, 93]. For example, the approach presented in [91] proposes an optimization strategy based on a top-down hierarchy traversal, while in [92] the authors propose to mine closed and maximal generalized itemsets. More recently, a new type of generalized pattern, called Misleading Generalized Itemset (MGI), has been proposed [87]. MGIs are high-level (generalized) itemsets for which a relevant subset of frequent descendants have a correlation type in contrast to their common ancestor. MGIs are worth considering separately from traditional itemsets if their low-level contrasting correlations cover almost the same portion of data as the high-level itemset, because the information provided by traditional high-level patterns becomes misleading. Unlike [87], this paper investigates how to perform MGI mining on the MapReduce platform. Furthermore, it evaluates the MGIs extracted big data acquired in smart city and networking environments.

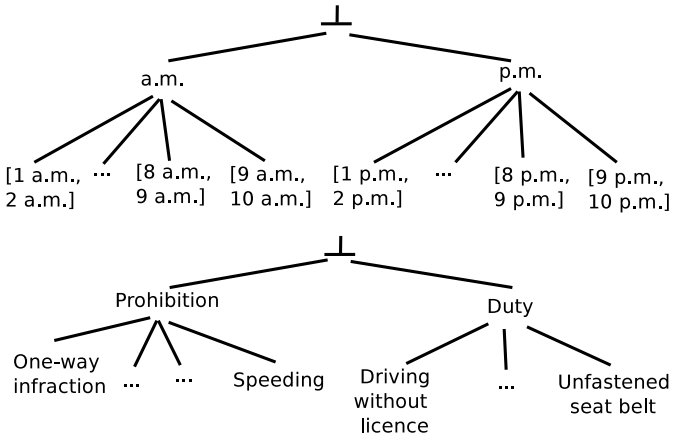

FIGURE 5.2: Example taxonomy built over items in $\mathcal{D}$

5.4 Preliminary concepts and problem statement

A relational dataset $\mathcal{D}$ consists of a set of records, where each record is a set of items [21]. Each item is a pair (*attribute*, *value*). A taxonomy Γ built over the source dataset $\mathcal{D}$ aggregates the data items into higher-level concepts (i.e., the generalized items). Table **??** and Table **??** report two representative examples of relational dataset and taxonomy, respectively, which hereafter will be used as running examples.

TABLE 5.1: Misleading Generalized Itemsets mined from $\mathcal{D}$. min_sup = 10%, max_neg_cor= 0.70, min_pos_cor= 0.80, and max_NOD = 80%.

Rid	**Infraction name**	**time stamp**
1	One-way infraction	[8 a.m.,9 a.m.]
2	One-way infraction	[8 a.m.,9 a.m.]
3	Speeding	[8 a.m.,9 a.m.]
4	Driving without license	[9 a.m.,10 a.m.]
5	Driving without license	[9 a.m.,10 a.m.]
6	Unfastened seat belt	[4 p.m.,5 p.m.]
7	One-way infraction	[8 a.m.,9 a.m.]

TABLE 5.2: Misleading Generalized Itemsets mined from $\mathcal{D}$. min_sup = 10%, max_neg_cor= 0.70, min_pos_cor= 0.80, and max_NOD = 80%.

Frequent itemset (level≥2) [correlation type (Kulc value)]	**Frequent descendants [correlation type (Kulc value)]**	**N overl degr**
{ (Time, a.m.), (Infraction name, Prohibition)} *[positive (5/6=0.83)]*	{(Time, [8 a.m.,9 a.m.]), (Infraction name, One-way infraction)} *[positive (7/8=0.88)]* {(Time, [8 a.m.,9 a.m.]), (Infraction name, Speeding)} *[[negative (5/8=0.63)]*	
{(Time, a.m.), (Infraction name, Duty)} *[negative (1/2=0.50)]*	{(Time, [9 a.m., 10 a.m.]), (Infraction name, Driving without license)} *[positive (1)]*	
{(Time, p.m.), (Infraction name, Duty)} *[negative (2/3=0.66)]*	{(Time, [4 p.m.,5 p.m.]), (Infraction name, Unfastened seat belt)} *[positive (1)]*	

A k-itemset is a set of k (generalized) items. For example, {(*Time*,*a.m.*), (*Infraction name*,*One-way infraction*)} is a 2-itemset, which indicates that the two items co-occur

(possibly at different abstraction levels) in the source data. Items/itemsets are characterized by many notable properties [84], such as support, coverage, descent and level of abstraction according to an input taxonomy Γ. For their definitions please refer to [84, 87, 91]. Similar to [91, 94], we target the correlations among items at same abstraction level, i.e. the itemsets that exclusively contain items with the same level. Such patterns are denoted by *level-sharing itemsets* [91].

The itemset correlation measures the strength of the correlation between its items. Similar to [94], in this paper we evaluate the correlation of a k-itemset I by means of the Kulczynsky (Kulc) correlation measure [95] Kulc values range from 0 to 1. By properly setting maximum negative and minimum positive Kulc thresholds, hereafter denoted by *max_neg_cor* and *min_pos_cor*, the itemsets may be classified as negatively correlated, uncorrelated, or positively correlated itemsets according to their correlation value.

Let $\mathcal{LSI}$ be the set of all frequent level-sharing itemsets in $\mathcal{D}$ according to a minimum support threshold min_sup. Given a frequent level-sharing itemset $X \in \mathcal{LGI}$ of level $l \geq 2$, let Desc*[X,Γ] be the subset of corresponding level-$(l-1)$ X's descendants for which the correlation type is in contrast to those of X. A Misleading Generalized Itemset (MGI) is a pattern in the form $X \triangleright \mathcal{E}$, where $X \in \mathcal{LSGI}$ and $\mathcal{E}$=Desc*[X,Γ] [87].

For example, by enforcing min_sup=10%, max_neg_cor=0.70, and min_pos_cor=0.80, MGI {(Time,a.m.), (Infraction name,Prohibition)} $\triangleright$ {(Time, [8 a.m.,9 a.m.]), (Infraction name,Speeding)} is mined from the dataset in Table **??**, because {(Time,a.m.), (Infraction name,Prohibition)} has a positive correlation (0.83), whereas its descendant itemset {(Time,[8 a.m., 9 a.m.]), (Infraction name,Speeding)} is negatively correlated (0.63).

To measure the degree of interest of a MGI $X \triangleright \mathcal{E}$ with respect to its corresponding traditional itemset version (X), the Not Overlapping Degree (NOD) measure has been defined in [87]. The NOD of an MGI $X \triangleright \mathcal{E}$ is defined as $\frac{\sup(X,\mathcal{D})-\mathrm{cov}(\mathcal{E},\mathcal{D})}{\sup(X,\mathcal{D})}$. It expresses the relative difference between the support of the ancestor itemset X and the coverage of its low-level contrasting correlations in $\mathcal{E}$. The NOD values range from 0 to 1. The lower NOD value we achieve, the more significant the degree of overlapping between the contrasting low-level correlations in $\mathcal{E}$ and their common ancestor X becomes.

The mining task addressed by this paper entails discovering from $\mathcal{D}$ all the MGIs for which the NOD value is less than or equal to a maximum threshold max_NOD. The subset of Misleading Generalized Itemsets mined from Table **??** by setting the maximum NOD threshold to 80% is reported in Table 5.2.

5.5 The MGI-Cloud architecture

The MGI-Cloud architecture provides a cloud-based service for discovering hidden and actionable patterns among potentially Big datasets. We focus our analysis on two specific case studies, i.e., the analysis of the traffic law infractions committed in a urban environment and the Internet traffic generated by an Italian ISP. To efficiently cope with Big Data, the system implementation is distributed and most operations are mapped to the MapReduce programming paradigm [88]. The architecture has been designed as a chain of distributed jobs running on an Hadoop cluster, as described below.

5.5.1 Data retrieval and preparation

Data about traffic law infractions was collected by traffic engineers. Reports about daily infractions are collected in local data repositories, which are then integrated into open Big Datasets. A traffic law infraction dataset $\mathcal{D}$ consists of a set of records r, each one representing a different infraction. Each record is a set of *items*, where items are pairs (*attribute*,*value*). *Attribute* can be a specific characteristic of the traffic law infraction (e.g., infraction name, law article) or a property of the context in which the infraction was committed (location, date, time), while *value* is the value assumed by the corresponding attribute. Hereafter, for the sake of simplicity, we focus our analysis on the following attribute subset: *Infraction name*, *Vehicle type*, *Location*, *Date*, and *Time*. We discretized time stamps (e.g., 8.10 a.m.) into 1-hour time slots (e.g., from 8 a.m. to 9 a.m.) using ad-hoc mapping functions. The dataset reported in Table **??** is already the output of the discretization step.

About the network traffic datasets, it has been obtained collecting network measuremets. To this aim, a passive probe is located on the access link (vantage point) that connects an edge network to the Internet. The passive probe sniffs all incoming and outgoing packets flowing on the link, i.e., packets directed to a node inside the network and generated by a node in the Internet, and vice versa. The probe runs Tstat [56], [57], a passive monitoring tool allowing network and transport layer measurement collection. Tstat rebuilds each TCP connection by matching incoming and outgoing segments. Thus, a flow-level analysis can be performed [57]. A TCP flow is identified by snooping the signaling flags (SYN, FIN, RST). The status of the TCP sender is rebuilt by matching sequence numbers on data segments with the corresponding acknowledgement (ACK) numbers. To evaluate the MGI-Cloud tool in real-world application, we focus on a subset of measurements describing the traffic flow among the many provided by Tstat. The most meaningful features, selected with the support of domain experts, are detailed in the following:

- the Round-Trip-Time (RTT) observed on a TCP flow, i.e., the minimum time lag between the observation of a TCP segment and the observation of the corresponding ACK. RTT is strongly related to the distance between the two nodes.
- the number of hops (Hop) from the remote node to the vantage point observed on packets belonging to the TCP flow, as computed by reconstructing the IP Time-To-Live
- the flow reordering probability (Preord), which can be useful to distinguish different paths
- the flow duplicate probability (Pdup), that can highlight a destination served by multiple paths
- the total number of packets (NumPkt), the total number of data packets (DataPkt), and the total number of bytes (DataBytes) sent from both the client and the server, separately (the client is the host starting the TCP flow)
- the minimum (WinMin), maximum (WinMax), and scale (WinScale) values of the TCP congestion window for both the client and the server, separately
- the TCP port of the server (Port)
- the class of service (Class), as defined by Tstat, e.g., HTTP, video, VoIP, SMTP, etc.

Based on measurements listed above, an input data record is defined by the following features: RTT, Hop, Preord, Pdup, NumPkt, DataPkt, DataBytes, WinMax, WinMin, WinScale, Port, Class. To obtain reliable estimates on reordering and duplicate probabilities, only TCP flows which last more than P = 10 packets are considered. This choice allow focusing the analysis on longlived flows, where the network path has a more relevant impact, thus providing more valuable information.
Since frequent itemset mining requires a transactional dataset of categorical values, data has to be discretized before the mining. The discretization step converts continuously valued measurements into categorical bins. Then, data are converted from the tabular to the transactional format. As already mentioned, attribute selection and data discretization are performed as distributed MapReduce jobs (specifcally, as a single map only job). Each record is processed by the map function and, if the number of packets is above the threshold (10 packets), the corresponding discretized transaction is emitted as a result of the mapping. This task entails an inherently parallel elaboration, considering that can be applied independently to each record.

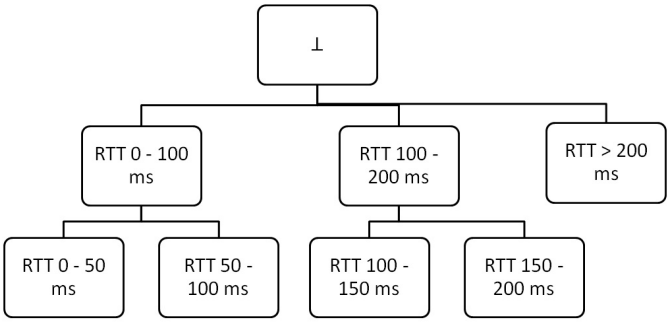

FIGURE 5.3: Example of taxonomie over RTT attribute

5.5.2 Taxonomy generation

To analyze data from a high-level viewpoint, the datasets are equipped with taxonomies. A taxonomy is a set of is-a hierarchies built over data items in $\mathcal{D}$. An example taxonomy built over the dataset in Table **??** is depicted in Table **??**. Items whose value is an high-level aggregation belonging to the taxonomy (e.g., (*Infraction name*,*Duty*)) are called *generalized items*. In Figure 5.3, instead, is reported a sample taxonomy over the attribute RTT of the network traffic dataset.

Analyst-provided taxonomies could be generated either manually or semi-automatically by domain experts.

To perform our analyzes of the traffic law dataset, we built 3-level hierarchies over the contextual attributes (Location, Time, Date). Specifically, geographical addresses are aggregated into the zip code, which in turn are aggregated into the corresponding district; 1-hour time slots are generalized as the corresponding 4- and 12-hour time slots, while dates are generalized as the corresponding month and year. Furthermore, vehicles are generalized as the corresponding category (e.g., *Car*, *Dumper track*, *Pickup track*), and infraction names are aggregated into the corresponding high-level class given by the public administration of Turin. A similar process has been applied to network traffic dataset. There were few cases where it was not possible: for instance, Protocols attributes habe been grouped in classes based on use case domains (similar to [96]).

In this work we consider as input balanced taxonomies (i.e., taxonomies whose hierarchies have all the same height). If experts do not provide balanced taxonomies, a re-balancing procedure similar to those adopted in [94] is applied prior to level-sharing itemset mining.

5.5.3 Level-sharing itemset mining

Given a preprocessed infraction dataset and a minimum support threshold min_sup, this job accomplishes the first MGI mining step, i.e., the extraction of all frequent level-sharing itemsets [91]. This job performs the following tasks.

Dataset extension. This task entails producing a new dataset version which integrates taxonomy information. To enable frequent level-sharing itemset mining from data containing items at different abstraction levels, it generates multiple copies of each record, one for each taxonomy level. While the original record contains only taxonomy leaves (i.e., the dataset items), each copy contains the corresponding combination of item generalizations at a different abstraction level.To avoid unnecessary I/O operations, the extended dataset version is not materialized on disk, but it is directly generated in the map function of the itemset extraction task and then immediately stored into a compact FP-tree structure [21].

Itemset extraction. To efficiently mine frequent level-sharing itemsets [91] from the extended dataset version, this task exploits a variation of the Hadoop-based itemset mining algorithm proposed in [40].

5.5.4 MGI extraction

This job performs MGI mining on top of the frequent level-sharing itemsets. Specifically, it accomplishes the task stated in Section 5.4. This step consists of a MapReduce job, as described in the following. The contribution of this job is new because, to the best of our knowledge, no cloud-based service currently supports MGI mining from Big Data.

To extract MGIs we combine each frequent level-sharing itemset I with its corresponding set of descendant itemsets Desc$[I, \Gamma]$. More specifically, In the map function for each level-sharing itemset I, the following two pairs (*key*, *value*) are emitted: (i) a pair (*key*, *value*), where *key* is the direct ancestor of itemset I and *value* the itemset I with its main properties (i.e., support and Kulc values) and (ii) a pair (*key*, *value*), where *key* is the itemset I is the *value*: itemset I with its main properties (i.e., support and Kulc values). Two pairs are emitted because each itemset can be a descendant of an itemset and a parent of another one at the same time. The first pair allows us to associate I with the ancestor key, whereas the second pair is used to associate I to itself if MGIs in the form $I \triangleright \mathcal{E}$ are extracted. The generated pairs allow us to map each itemset and its corresponding descendants to the same key. Hence, in the reduce function, each key is associated with a specific itemset I and the corresponding set of values contains both the (ancestor) itemset I and its respective descendants. By iterating on the set

of values associated with key I, we generate candidate MGIs $I \triangleright \mathcal{E}$, where $\mathcal{E}$ is the set of I's descendants in contrast to I in terms of correlation type, and we compute the corresponding NOD values. Finally, only the MGIs satisfying the max_NOD threshold are stored into the HDFS file system.

5.6 Experiments

We performed experiments on two real datasets acquired in different domains:

AperTo dataset. This open dataset, available at http://aperto.comune.torino.it, collects information about approximately 2 millions of traffic law infractions committed in the city of Turin over the 3-year period 2011-2013. The dataset is characterized by five attributes (*Infraction name*, *Vehicle type*, *Location*, *Date*, and *Time*). Its size is approximately 198 MB. Hierarchies over the infraction data items were defined according to the guidelines reported in Section 5.5.2.

BigNetData dataset. This relational network traffic dataset has been obtained by performing different capture stages on a backbone link of a nation-wide ISP in Italy that offers us three different vantage points. The dataset has size 192.56 GB and it consists of 413,012,989 records, i.e., one record for each bi-directional TCP flow). A more detailed dataset description is given in [40].

The MapReduce jobs of the MGI-Cloud workflow (see Section 5.5) were developed in Java using the new Hadoop Java APIs. The experiments were performed on a cluster of 5 nodes running Cloudera's Distribution of Apache Hadoop (CDH4.5). Each cluster node is a 2.67 GHz six-core Intel(R) Xeon(R) X5650 machine with 32 Gbyte of main memory running Ubuntu 12.04 server with the 3.5.0-23-generic kernel. All the reported execution times are real times obtained from the Cloudera Manager web control panel.

In the experiments we addressed the following issues: (i) the analysis of the characteristics of the mining results achieved with different parameter settings ((Section 5.6.1)), (ii) the validation of the usefulness of the results achieved for performing in-depth analysis (Section 5.6.2), and (iii) the scalability of the MGI Miner algorithm with the number of nodes (Section 5.6.3). We addressed Tasks (i) and (ii) mainly on AperTo dataset, because data fully complies with the context under analysis (i.e., infraction data analysis), whereas Task (iii) was addressed on BiGNetData, because it is a larger dataset characterized by a fairly complex data distribution.

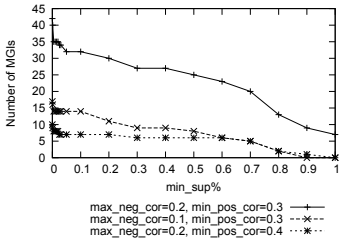

FIGURE 5.4: Effect of the minimum support threshold. max_NOD=60%.

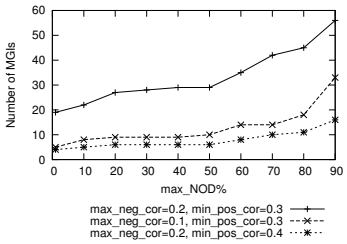

FIGURE 5.5: Effect of the maximum NOD threshold. minsup=0.02%.

5.6.1 Characteristics of the mining results

We analyzed the impact of the main input parameters of the MGI Miner algorithm on the number of MGIs mined. Figure 5.4 summarizes the number of mined MGIs by varying the minimum support threshold (min_sup) for different combinations of minimum positive and maximum negative correlation thresholds (max_neg_cor and min_pos_cor, respectively), while Figure 5.5 shows the number of mined MGIs by varying the max_NOD threshold for the same combinations of correlation threshold values.

The number of mined MGIs non-linearly increases by decreasing the minimum support threshold due to the combinatorial increase in the number of generated frequent itemsets [38]. Since most itemsets have correlation between 0.1 and 0.3, the maximum number of MGIs is extracted if max_neg_cor and min_pos_cor fall in this value range, because the generalization process is most likely to flip the itemset correlation types. As expected, the smaller the gap between max_neg_cor and min_pos_cor, the more MGIs are extracted because correlation type changes occur, on average, more frequently. For all the tested configurations, the set of mined MGIs remains still manageable by domain experts for manual inspection even while setting relatively low support thresholds (e.g., 35 MGIs mined with max_neg_cor=0.2, min_pos_cor=0.3 and min_sup=0.01%).

The number of mined MGIs non-linearly increases while increasing the maximum not overlapping degree threshold max_NOD, because low-level itemsets are more likely to

cover a significant portion of data already covered by the corresponding high-level itemsets. However, in all the performed experiments the set of MGIs, which represent anomalies/contrasting situations, remains easily manageable by domain experts for manual exploration.

5.6.2 Result validation

We examined the MGIs extracted from the AperTo dataset to validate their interestingness and usefulness in a real-life context, i.e., the analysis of the traffic law infractions committed in a urban environment.

As a first example, let us consider the following MGI extracted by enforcing min_sup=0.02%, max_neg_cor=0.1, min_pos_cor=0.4, and max_NOD=60%: {(Location,Zip code 10125), (Infraction name,Prohibition)} ▷ {(Location, Sommeiller Avenue), (Infraction name, One-way infraction)}. The high-level itemset {(Location,Zip Code 10125), (Infraction name,Prohibition)} is negatively correlated, whereas its frequent descendant {(Location, Sommeiller Avenue), (Infraction name, One-way infraction)} is positively correlated and it covers a significant portion of data already covered by the high-level itemset (∼59%). Hence, to a certain extent, analyzing only the traditional high-level itemset instead of the complete MGI could be misleading. This pattern indicates that in a certain area of Turin, identified by zip code 10125, a category of infractions (prohibitions) is not very likely to occur, whereas for a specific avenue within the area wrong way driving prohibition is violated more commonly than expected. Hence, road signs in Sommeiller Avenue could be either not well visible or misplaced. The public administration of Turin should deem such information to be worthy for signage maintenance and monitoring.

Let us consider now the following MGI: {(Location,District 1), (Vehicle type,Private car),(Time, p.m.)} ▷ {(Location, Zip code 10122),(Vehicle type, Private car),(Time, (4 p.m.,8 p.m.]), (Location, Zip code 10121),(Vehicle type, Private car),(Time, (8 p.m.,12 p.m.]), . . . }. The high-level itemset is positively correlated, whereas 11 of its descendant itemsets are negatively correlated and the NOD value of the mined MGI is 58%. District 1 of Turin appears to be an area in which many infractions are committed by private cars during the afternoon, evening, or night. Hence, traffic corps should monitor the area more carefully in these specific daily time periods. However, in 42% of the subareas of district 1 (e.g., the ones identified by zip codes 10121 and 10122, respectively), infractions are less likely to occur than in the others. Therefore, it would be more advisable to monitor the subareas other than district 1.

In summary, MGI extraction from infraction data could help traffic corps optimize road monitoring services and identify anomalous situations be due to either inappropriate citizens' behaviors or to temporary service disruptions.

We have also tried to validate the results obtained from network traffic traces dataset. In this case we focused our analysis on the pattern related to either protocols or RTT values, because we deemed such patterns as interesting to understand application/service server geography.

As an example let use consider the following MGI extracted by enforcing max_neg_cor=0.2, min_pos_cor=0.3, and max_NOD=70%:
{(CLASS=CHAT) (RTT_MIN=100-200)} ▷ {(CLASS=32) (RTT_MIN=165-170), (CLASS=513) (RTT_MIN=145-150)}. The high-level itemset {(CLASS=CHAT) (RTT_MIN=100-200)} is negatively correlated whereas its frequent descendants {(CLASS=32) (RTT_MIN=165-170), (CLASS=513) (RTT_MIN=145-150)} are positively correlated and they cover a significant portion of data already covered by the high-level itemset (especially CLASS=32, with the 32%). This means that the traffic flows associated with any chat protocol and characterized by RTT between 100 and 200 ms are less likely to occur than expected, whereas the flows associated with two specific chat protocols, i.e., MSN (class 32) and Skype (class 513), are likely to have RTTs in the ranges 165-170 ms and 145-150 ms, respectively. Hence, in this case, analyzing only the high-level itemset instead of the complete MGI could be misleading and the pattern may indicate that only some specific chat protocols (MSN, Skype) often rely on servers physically located relatively faraway with each other. In this specific example, MGI analysis proves its effectiveness in the network environment, i.e. helping network administrator to understand and optimize networks and identify anomalous situations. Nevertheless, there are many other possible use cases because of the generality of our approach and its compatibility with huge datasets due to its distributed architecture. **TO DO: mettere esempi in una tabella**

5.6.3 Scalability with the number of cluster nodes

We evaluated the scalability of the proposed architecture by measuring the speedup achieved increasing the number of Hadoop cluster nodes. Specifically, we considered three configurations: 1 node, 3 nodes, and 5 nodes. Figure 5.6 reports the speedup achieved setting min_sup to 1%, max_neg_cor to 0.1, min_pos_cor to 0.3, and max_nod to 60%. The first box in Figure 5.6 (i.e., 1 node) corresponds to a run of MGI-Cloud on a single node. Speedup with increasing nodes is computed against the single-node performance. The achieved results show that our approach scales roughly linearly with the

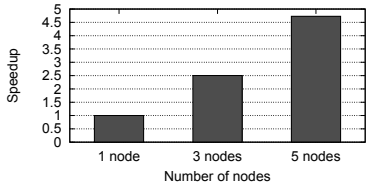

FIGURE 5.6: Speedup on the BigNetData dataset.

number of nodes and the speedup approximately corresponds to the number of cluster nodes.

5.7 Conclusions and future perspectives

This paper presents a cloud-based service for discovering Misleading Generalized Itemsets from Big Data equipped with taxonomies. To cope with Big Data the architecture has been designed to run on a distributed Hadoop architecture [88]. A preliminary analysis of the applicability and usefulness of the proposed architecture was conducted on real Big Data acquired in a smart city environment and related to traffic law infractions. However, the offered service could find application in many other application contexts, such as (i) social network analysis, (ii) network data analysis, and (iii) financial data analysis. As future work, we aim at optimizing and extending the current Hadoop architecture as well as testing its applicability in other real-life contexts.

5.8 Relevant Pubblications

E. Baralis, L. Cagliero, T. Cerquitelli, S. Chiusano, P. Garza, L. Grimaudo, and F. Pulvirenti, "Nemico: Mining network data through cloud-based data mining techniques," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2014, pp. 503–504.

E. Baralis, L. Cagliero, T. Cerquitelli, S. Chiusano, P. Garza, L. Grimaudo, and F. Pulvirenti, "Misleading generalized itemset mining in the cloud," in *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*, Aug 2014, pp. 211–216.

D. Apiletti, E. Baralis, L. Cagliero, T. Cerquitelli, S. Chiusano, P. Garza, L. Grimaudo, and F. Pulvirenti, "Network traffic analysis by means of misleading generalized itemsets," *BIGDAP 2014 - Big Data and Applications*, p. 39, 2014.

E. Baralis, L. Cagliero, T. Cerquitelli, S. Chiusano, L. Grimaudo, F. Pulvirenti, and P. Garza, "Mgi-cloud: discovery of misleading generalized itemsets," in *22nd Italian Symposium on Advanced Database Systems, SEBD 2014, Sorrento Coast, Italy, June 16-18, 2014.*, 2014, pp. 45–52.

Chapter 6

Conclusion

LORE IPSUM...

Works excluded from the dissertation. Please note, for the sake of consistency, the dissertation does not cover entirely the reseach activities covered by my 3-year PhD. Above all, my main research topic of the third year (i.e. streaming algorithms for K-NN graphs and clustering), is not included in this thesis and can be found in the Appendix.

Bibliography

[1] E. Junqué de Fortuny, D. Martens, and F. Provost, "Predictive modeling with big data: is bigger really better?" *Big Data*, vol. 1, no. 4, pp. 215–226, 2013.

[2] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[3] D. Apiletti, P. Garza, and F. Pulvirenti, "A review of scalable approaches for frequent itemset mining," in *East European Conference on Advances in Databases and Information Systems*. Springer International Publishing, 2015, pp. 243–247.

[4] D. Apiletti, T. Baralis, Elena Cerquitelli, P. Garza, and F. Pulvirenti, "Frequent itemsets mining for big data: an experimental analysis," *Submitted to Big Data Research - Under Review*.

[5] D. Apiletti, E. Baralis, T. Cerquitelli, P. Garza, P. Michiardi, and F. Pulvirenti, "Pampa-hd: A parallel mapreduce-based frequent pattern miner for high-dimensional data," in *IEEE ICDM Workshop on High Dimensional Data Mining (HDM)*, Atlantic City, NJ, USA, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7395755

[6] D. Apiletti, T. Baralis, Elena Cerquitelli, P. Garza, P. Michiardi, and F. Pulvirenti, "A parallel map-reduce algorithm to efficiently support itemset mining on high dimensional data," *Submitted to Big Data Research - Under Review*.

[7] E. Baralis, L. Cagliero, T. Cerquitelli, S. Chiusano, P. Garza, L. Grimaudo, and F. Pulvirenti, "Nemico: Mining network data through cloud-based data mining techniques," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*. IEEE Computer Society, 2014, pp. 503–504.

[8] ——, "Misleading generalized itemset mining in the cloud," in *2014 IEEE International Symposium on Parallel and Distributed Processing with Applications*, Aug 2014, pp. 211–216.

[9] E. Baralis, L. Cagliero, T. Cerquitelli, S. Chiusano, L. Grimaudo, F. Pulvirenti, and P. Garza, "Mgi-cloud: discovery of misleading generalized itemsets," in *22nd Italian Symposium on Advanced Database Systems, SEBD 2014, Sorrento Coast, Italy, June 16-18, 2014.*, 2014, pp. 45–52.

[10] D. Apiletti, E. Baralis, L. Cagliero, T. Cerquitelli, S. Chiusano, P. Garza, L. Grimaudo, and F. Pulvirenti, "Network traffic analysis by means of misleading generalized itemsets," *BIGDAP 2014 - Big Data and Applications*, p. 39, 2014.

[11] R. Xu and D. Wunsch, II, "Survey of clustering algorithms," *Trans. Neur. Netw.*, vol. 16, no. 3, pp. 645–678, May 2005.

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

[13] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: Current status and future directions," *Data Min. Knowl. Discov.*, vol. 15, no. 1, pp. 55–86, Aug. 2007.

[14] E. Glatz, S. Mavromatidis, B. Ager, and X. Dimitropoulos, "Visualizing big network traffic data using frequent pattern mining and hypergraphs," *Computing*, vol. 96, no. 1, pp. 27–38, 2014.

[15] L. Cagliero and P. Garza, "Infrequent weighted itemset mining using frequent pattern growth," *IEEE transactions on knowledge and data engineering*, vol. 26, no. 4, pp. 903–915, 2014.

[16] A. Gupta, A. Mittal, and A. Bhattacharya, "Minimally infrequent itemset mining using pattern-growth paradigm and residual trees," in *Proceedings of the 17th International Conference on Management of Data*. Computer Society of India, 2011, p. 13.

[17] I. Paredes-Oliva, I. Castell-Uroz, P. Barlet-Ros, X. Dimitropoulos, and J. Sole-Pareta, "Practical anomaly detection based on classifying frequent traffic patterns," in *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*. IEEE, 2012, pp. 49–54.

[18] D. Brauckhoff, X. Dimitropoulos, A. Wagner, and K. Salamatian, "Anomaly extraction in backbone networks using association rules," *IEEE/ACM Transactions on Networking (TON)*, vol. 20, no. 6, pp. 1788–1799, 2012.

[19] B. Liu, W. Hsu, and Y. Ma, "Integrating classification and association rule mining," in *KDD '98*, 1998, pp. 80–86.

[20] L. Venturini, P. Garza, and D. Apiletti, *BAC: A Bagged Associative Classifier for Big Data Frameworks*. Cham: Springer International Publishing, 2016, pp. 137–146. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-44066-8_15

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

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

[23] F. Pan, G. Cong, A. K. H. Tung, J. Yang, and M. J. Zaki, "Carpenter: Finding closed patterns in long biological datasets," in *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '03. New York, NY, USA: ACM, 2003, pp. 637–642. [Online]. Available: http://doi.acm.org/10.1145/956750.956832

[24] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[25] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project*, vol. 11, p. 21, 2007.

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

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

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

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

[30] "Apache Giraph, last Accessed: 16/10/2015," 2012. [Online]. Available: http://giraph.apache.org/

[31] "simSQL, a system for stochastic analytics, last Accessed: 16/10/2015," 2013. [Online]. Available: http://cmj4.web.rice.edu/SimSQL/SimSQL.html

[32] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[33] "The Apache Mahout machine learning library, last Accessed: 16/10/2015," 2013. [Online]. Available: http://mahout.apache.org/

[34] "MADlib: Big Data Machine Learning in SQL, last Accessed: 16/10/2015." [Online]. Available: http://madlib.net/

[35] "The Apache Spark scalable machine learning library, last Accessed: 16/10/2015," 2015. [Online]. Available: https://spark.apache.org/mllib/

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

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

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

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

[40] D. Apiletti, E. Baralis, T. Cerquitelli, S. Chiusano, and L. Grimaudo, "Searum: A cloud-based service for association rule mining," in *ISPA'13*, 2013, pp. 1283–1290.

[41] D. Antonelli, E. Baralis, G. Bruno, L. Cagliero, T. Cerquitelli, S. Chiusano, P. Garza, and N. A. Mahoto, "MeTA: Characterization of Medical Treatments at Different Abstraction Levels," *ACM TIST*, vol. 6, no. 4, p. 57, 2015.

[42] G. Cong, A. K. H. Tung, X. Xu, F. Pan, and J. Yang, "FARMER: finding interesting rule groups in microarray datasets," in *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, June 13-18, 2004*, G. Weikum, A. C. König, and S. Deßloch, Eds. ACM, 2004, pp. 143–154. [Online]. Available: http://doi.acm.org/10.1145/1007568.1007587

[43] T. Cerquitelli and 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. [Online]. Available: http://ceur-ws.org/Vol-1558/paper15.pdf

[44] M. L. Antonie, O. R. Zaiane, and A. Coman, "Application of data mining techniques for medical image classification," *In MDM/KDD*, 2001.

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

[46] E. Baralis, L. Cagliero, A. Fiori, and P. Garza, "Mwi-sum: A multilingual summarizer based on frequent weighted itemsets," *ACM Trans. Inf. Syst.*, vol. 34, no. 1, p. 5, 2015.

[47] A. de Andrade Lopes, R. Pinho, F. V. Paulovich, and R. Minghim, "Visual text mining using association rules," *Computers & Graphics*, vol. 31, no. 3, pp. 316–326, 2007. [Online]. Available: http://dx.doi.org/10.1016/j.cag.2007.01.023

[48] M. Mampaey, N. Tatti, and 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*, ser. KDD '11. New York, NY, USA: ACM, 2011, pp. 573–581. [Online]. Available: http://doi.acm.org/10.1145/2020408.2020499

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

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

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

[52] S. Moens, E. Aksehirli, , and B. Goethals, "Dist-eclat and bigfim," https://github.com/ua-adrem/bigfim, 2013.

[53] N. Agrawal, T. Imielinski, and A. Swami, "Database mining: A performance perspective," *In IEEE TKDE*, vol. 5, no. 6, 1993.

[54] "Cloudera, last Accessed: 16/10/2015." [Online]. Available: http://www.cloudera.com

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

[56] A. Finamore, M. Mellia, M. Meo, M. Munafò, and D. Rossi, "Experiences of internet traffic monitoring with tstat," *IEEE Network*, vol. 25, no. 3, pp. 8–14, 2011.

[57] M. Mellia, M. Meo, L. Muscariello, and D. Rossi, "Passive analysis of tcp anomalies," *Computer Networks*, vol. 52, no. 14, pp. 2663–2676, 2008.

[58] D. Giordano, S. Traverso, L. Grimaudo, M. Mellia, E. Baralis, A. Tongaonkar, and S. Saha, "Youlighter: An unsupervised methodology to unveil youtube cdn changes," *arXiv preprint arXiv:1503.05426*, 2015.

[59] D. Apiletti, E. Baralis, T. Cerquitelli, S. Chiusano, and 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.

[60] X. Jin, B. W. Wah, X. Cheng, and Y. Wang, "Significance and challenges of big data research," *Big Data Research*, vol. 2, no. 2, pp. 59 – 64, 2015, visions on Big Data. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2214579615000076

[61] O. Y. Al-Jarrah, P. D. Yoo, S. Muhaidat, G. K. Karagiannidis, and K. Taha, "Efficient machine learning for big data: A review," *Big Data Research*, vol. 2, no. 3, pp. 87–93, 2015. [Online]. Available: http://dx.doi.org/10.1016/j.bdr.2015.04.001

[62] J.-G. Lee and M. Kang, "Geospatial big data: Challenges and opportunities," *Big Data Research*, vol. 2, no. 2, pp. 74 – 81, 2015, visions on Big Data. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2214579615000040

[63] M. Cuturi, "UCI machine learning repository. PEMS-SF data set," 2011. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/PEMS-SF

[64] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[65] M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml

[66] "California department of transportation." [Online]. Available: http://http://pems.dot.ca.gov/.Lastaccess:April,21st2016

[67] M. L. data set repository, "Breast cancer dataset (kent ridge." [Online]. Available: http://mldata.org/repository/data/viewslug/breast-cancer-kent-ridge-2Lastaccess:July,15th2015

[68] V. Jacobson, "Congestion avoidance and control," *SIGCOMM Comput. Commun. Rev.*, vol. 18, no. 4, pp. 314–329, Aug. 1988. [Online]. Available: http://doi.acm.org/10.1145/52325.52356

[69] Apache Software Foundation. Apache mahout:: Scalable machine-learning and data-mining library. [Online]. Available: http://mahout.apache.org

[70] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, "MLlib: Machine learning in apache spark," May 2016. [Online]. Available: http://arxiv.org/abs/1505.06807

[71] A. D. Sarma, F. N. Afrati, S. Salihoglu, and J. D. Ullman, "Upper and lower bounds on the cost of a map-reduce computation," *Proc. VLDB Endow.*, vol. 6, no. 4, pp. 277–288, Feb. 2013. [Online]. Available: http://dx.doi.org/10.14778/2535570.2488334

[72] R. Vimieiro and P. Moscato, "A new method for mining disjunctive emerging patterns in high-dimensional datasets using hypergraphs," *Information Systems*, vol. 40, pp. 1 – 10, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0306437913001221

[73] P. Bermejo, L. de la Ossa, J. A. GÃąmez, and J. M. Puerta, "Fast wrapper feature subset selection in high-dimensional datasets by means of filter re-ranking," *Knowledge-Based Systems*, vol. 25, no. 1, pp. 35 – 44, 2012, special Issue on New Trends in Data Mining. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095070511100027X

[74] B. Kamsu-Foguem, F. Rigal, and F. Mauget, "Mining association rules for the quality improvement of the production process," *Expert Systems with Applications*, vol. 40, no. 4, pp. 1034 – 1045, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0957417412010007

[75] J. Nahar, T. Imam, K. S. Tickle, and Y.-P. P. Chen, "Association rule mining to detect factors which contribute to heart disease in males and females," *Expert Systems with Applications*, vol. 40, no. 4, pp. 1086 – 1093, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S095741741200989X

[76] Cisco, "Netflow." [Online]. Available: http://www.cisco.com/c/en/us/products/ios-nx-os-software/ios-netflow/index.htmlLastaccess:July,15th2015

[77] D. Brauckhoff, X. Dimitropoulos, A. Wagner, and K. Salamatian, "Anomaly extraction in backbone networks using association rules," *Networking, IEEE/ACM Transactions on*, vol. 20, no. 6, pp. 1788–1799, Dec 2012.

[78] D. Apiletti, E. Baralis, T. Cerquitelli, and V. D'Elia, "Characterizing network traffic by means of the netmine framework," *Comput. Netw.*, vol. 53, no. 6, pp. 774–789, Apr. 2009. [Online]. Available: http://dx.doi.org/10.1016/j.comnet.2008.12.011

[79] ——, "Characterizing network traffic by means of the netmine framework," *Computer Networks*, vol. 53, no. 6, pp. 774–789, 2009.

[80] E. Baralis, L. Cagliero, T. Cerquitelli, V. D'Elia, and P. Garza, "Expressive generalized itemsets," *Information Sciences*, vol. 278, pp. 327–343, 2014.

[81] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "Blinc: multilevel traffic classification in the dark." in *SIGCOMM*, 2005, pp. 229–240.

[82] J. Erman, M. Arlitt, and A. Mahanti, "Traffic classification using clustering algorithms," in *MineNet '06*, 2006, pp. 281–286.

[83] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2005.

[84] R. Srikant and R. Agrawal, "Mining generalized association rules," in *VLDB 1995*, 1995, pp. 407–419.

[85] E. Baralis, L. Cagliero, T. Cerquitelli, V. D'Elia, and P. Garza, "Support driven opportunistic aggregation for generalized itemset extraction," in *IEEE Conf. of Intelligent Systems*, 2010, pp. 102–107.

[86] E. Baralis, L. Cagliero, T. Cerquitelli, S. Chiusano, and P. Garza, "Frequent weighted itemset mining from gene expression data," in *BIBE*, 2013, pp. 1–4.

[87] L. Cagliero, T. Cerquitelli, P. Garza, and L. Grimaudo, "Misleading generalized itemset discovery," *Expert Syst. Appl.*, vol. 41, no. 4, pp. 1400–1410, 2014.

[88] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[89] K. Sriphaew and T. Theeramunkong, "A new method for finding generalized frequent itemsets in association rule mining," in *Proceeding of the VII Intern. Symposium on Computers and Communications*, 2002, pp. 1040–1045.

[90] E. Baralis, L. Cagliero, T. Cerquitelli, and P. Garza, "Generalized association rule mining with constraints," *Inf. Sci.*, vol. 194, pp. 68–84, 2012.

[91] J. Han and Y. Fu, "Mining multiple-level association rules in large databases," *IEEE Transactions on knowledge and data engineering*, vol. 11, no. 7, pp. 798–805, 1999.

[92] D. Kunkle, D. Zhang, and G. Cooperman, "Mining frequent generalized itemsets and generalized association rules without redundancy," *J. Comput. Sci. Technol.*, vol. 23, no. 1, pp. 77–102, 2008.

[93] L. Cagliero, "Discovering temporal change patterns in the presence of taxonomies," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 3, pp. 541–555, 2013.

[94] M. Barsky, S. Kim, T. Weninger, and J. Han, "Mining flipping correlations from large datasets with taxonomies," *Proc. VLDB Endow.*, vol. 5, no. 4, pp. 370–381, Dec. 2011.

[95] T. Wu, Y. Chen, and J. Han, "Re-examination of interestingness measures in pattern mining: a unified framework," *Data Min. Knowl. Discov.*, vol. 21, no. 3, pp. 371–397, Nov. 2010.

[96] L. Grimaudo, M. Mellia, and E. Baralis, "Hierarchical learning for fine grained internet traffic classification," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2012 8th International*. IEEE, 2012, pp. 463–468.