POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Parallel MapReduce Algorithm to Efficiently Support Itemset Mining on High Dimensional Data

Original
A Parallel MapReduce Algorithm to Efficiently Support Itemset Mining on High Dimensional Data / Apiletti, Daniele; Baralis, Elena; Cerquitelli, Tania; Garza, Paolo; Pulvirenti, Fabio; Michiardi, Pietro. - In: BIG DATA RESEARCH. - ISSN 2214-5796. - 10:(2017), pp. 53-69. [10.1016/j.bdr.2017.10.004]

Availability:
This version is available at: 11583/2693039 since: 2021-03-30T12:40:55Z

Publisher:
Elsevier Inc.

Published
DOI:10.1016/j.bdr.2017.10.004

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
Elsevier postprint/Author's Accepted Manuscript

© 2017. This manuscript version is made available under the CC-BY-NC-ND 4.0 license
http://creativecommons.org/licenses/by-nc-nd/4.0/.The final authenticated version is available online at:
http://dx.doi.org/10.1016/j.bdr.2017.10.004

(Article begins on next page)

17 April 2024

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

Daniele Apiletti, Elena Baralis, Tania Cerquitelli,
Paolo Garza, Fabio Pulvirenti[a], and Pietro Michiardi[b]

[a]*Dipartimento di Automatica e Informatica*
Politecnico di Torino
Torino, Italy
Email: name.surname@polito.it
[b]*Data Science Department*
Eurecom
Sophia Antipolis, France
Email: pietro.michiardi@eurecom.fr

Abstract

In today's world, large volumes of data are being continuously generated by many scientific applications, such as bioinformatics or networking. Since each monitored event is usually characterized by a variety of features, high-dimensional datasets have been continuously generated. To extract value from these complex collections of data, different exploratory data mining algorithms can be used to discover hidden and non-trivial correlations among data. Frequent closed itemset mining is an effective but computational expensive technique that is usually used to support data exploration. 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, 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. This work

introduces PaMPa-HD, a MapReduce-based frequent closed itemset mining algorithm for high dimensional datasets. An efficient solution has been proposed to parallelize and speed up the mining process. Furthermore, different strategies have been proposed to easily configure the algorithm parameter. 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.
Keywords: high-dimensional data, frequent closed itemset mining, Hadoop Framework

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" [1], have changed dramatically the importance of the intelligent analysis of data. Data mining, together with machine learning [2], 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 [3] and Spark [4]. 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), focusing, on the contrary, on very large datasets in terms of number of transactions. Currently, only single-machine implementations exist to address very long transactions, such as Carpenter [5], 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 [6] 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 [7]. 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 [8]. To effectively deal with those high-dimensional datasets,

novel and distributed approaches are needed.

This work introduces PaMPa-HD [9], a parallel MapReduce-based frequent closed itemset mining algorithm for high-dimensional datasets. PaMPa-HD relies on the Carpenter algorithm [5]. The PaMPa-HD design, 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 2 introduces the frequent (closed) itemset mining problem, Section 3 briefly describes the centralized version of Carpenter, and Section 4 presents the proposed PaMPa-HD algorithm. Section 5 describes the experimental evaluations proving the effectiveness of the proposed technique, Section 6 presents a brief review of the state of the art, and Section 7 discusses possible applications of PaMPa-HD. Finally, Section 8 includes a brief summary of the experimental results and the proposed solution, while Section 9 introduces possible further developments.

2. Frequent itemset mining background

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 1a reports an example of a transactional dataset with 5 transactions. It is used as a

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

running example through the paper.

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 $\{aco\}$ in the running example dataset $\mathcal{D}$ is 2/5 and its tidlist is $\{1, 3\}$. An itemset I is considered frequent if its support is greater than a user-provided minimum support threshold $minsup$.

Given a transactional dataset $\mathcal{D}$ and a minimum support threshold $minsup$, the Frequent Itemset Mining [10] problem consists in extracting the complete set of frequent itemsets from $\mathcal{D}$. In this paper, we focus on a valuable subset of frequent itemsets called frequent closed itemsets [5]. 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 $\{ab\}$ is a frequent itemset (support=2), but it is not closed for the presence of the itemset $\{abls\}$ (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., $tidlist(\{i\})$. Let r be an arbitrary row of TT, $r.tidlist$ denotes the tidlist of row r. Figure 1b reports the transposed representation of the running example reported in Figure 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 instance, consider the transposed table TT reported in Figure 1b. The projection of TT on the tidlist $\{2,3\}$ is the transposed table reported in Figure 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 1c).

3. The Carpenter algorithm

The most popular techniques to perform itemset mining (e.g., Apriori [11] and FP-growth [12]) adopt the itemset enumeration approach (see Section 6 for further discussion). However, itemset enumeration revealed to be ineffective with datasets with a high average number of items per transactions [5]. To tackle this problem, the Carpenter algorithm [5] 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 1a, and more specifically, its transposed version (see Figure 1b). Recall that in the transposed representation each row of the table consists of an item i with its tidlist. For instance, the last

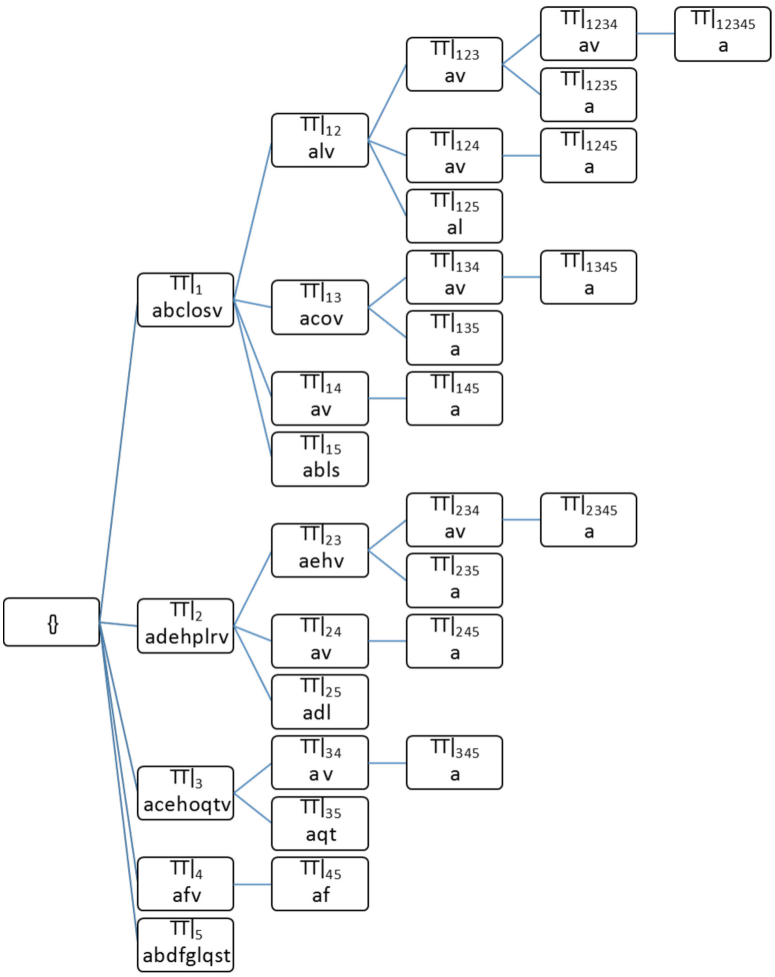

Figure 2: The transaction enumeration tree of the running example dataset in Figure 1a. For the sake of clarity, no pruning rules are applied to the tree.

row of Figure 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 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 1c, matches the node $\{2,3\}$ in Figure 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 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
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 [5], 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, is very challenging in a distributed environment that adopts a shared-nothing architecture, like the one we address in this work.

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 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 2), the centralized Carpenter algorithm extracts the whole set of closed itemsets 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 subprocesses, that autonomously evaluate sub-trees of the search space. Specifically, the whole problem can be split by assigning each subtree rooted in

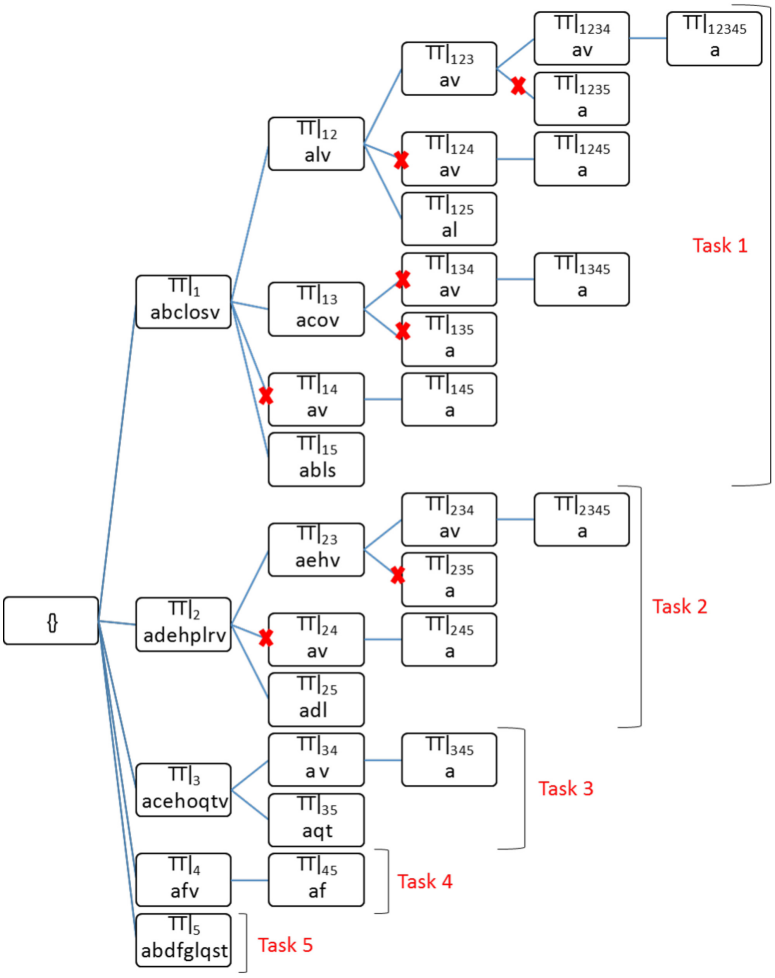

Figure 3: Running toy example: each node expands a branch of the tree independently. For the sake of clarity, pruning rule 1 and 2 are not applied. The pruning rule 3 is applied only within the same task: the small crosses on the edges represent pruned nodes due to local pruning rule 3, e.g. the one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

$TT|_X$, 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 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 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 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 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, in which the small crosses represent the pruning related to the local state memory; and in Figure 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 combina-

tion {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 5, we achieve a better exploitation of the resources.

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 source code of PaMPa-HD is freely available at https://github.com/fpulvi/PaMPa-HD .

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.

Job 1 (Algorithm 2). Each mapper is fed with a transaction of the input

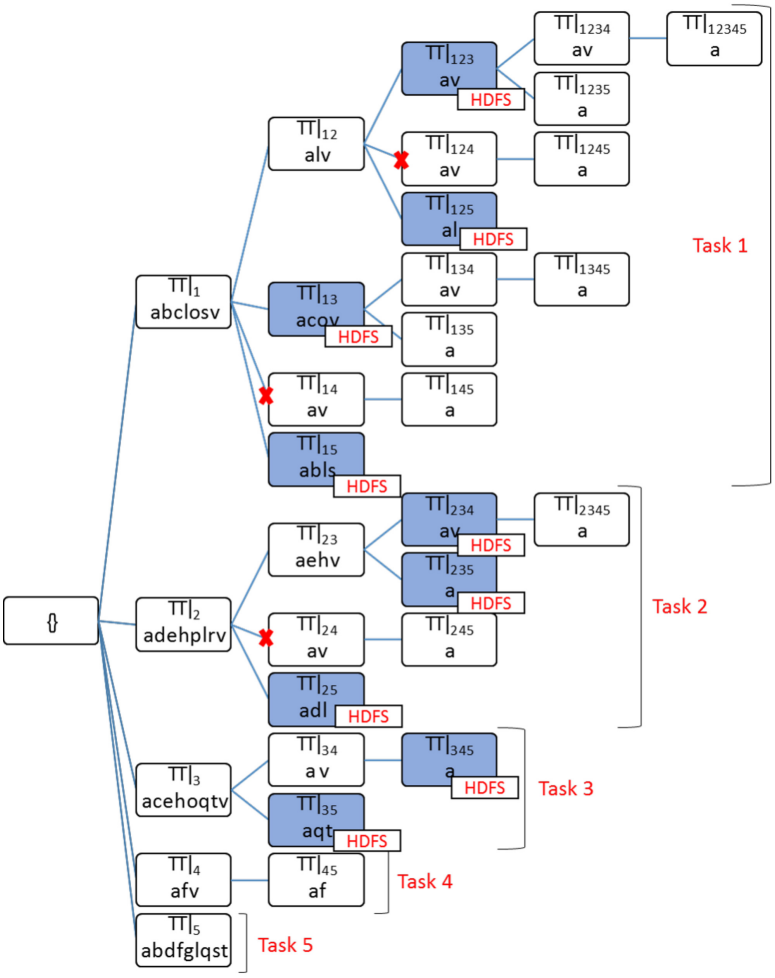

Figure 4: Execution of PaMPa-HD on the running example dataset. For the sake of clarity, pruning rules 1 and 2 are not applied. The dark nodes represent the nodes that have been written to HDFS in order to apply the synchronization job.

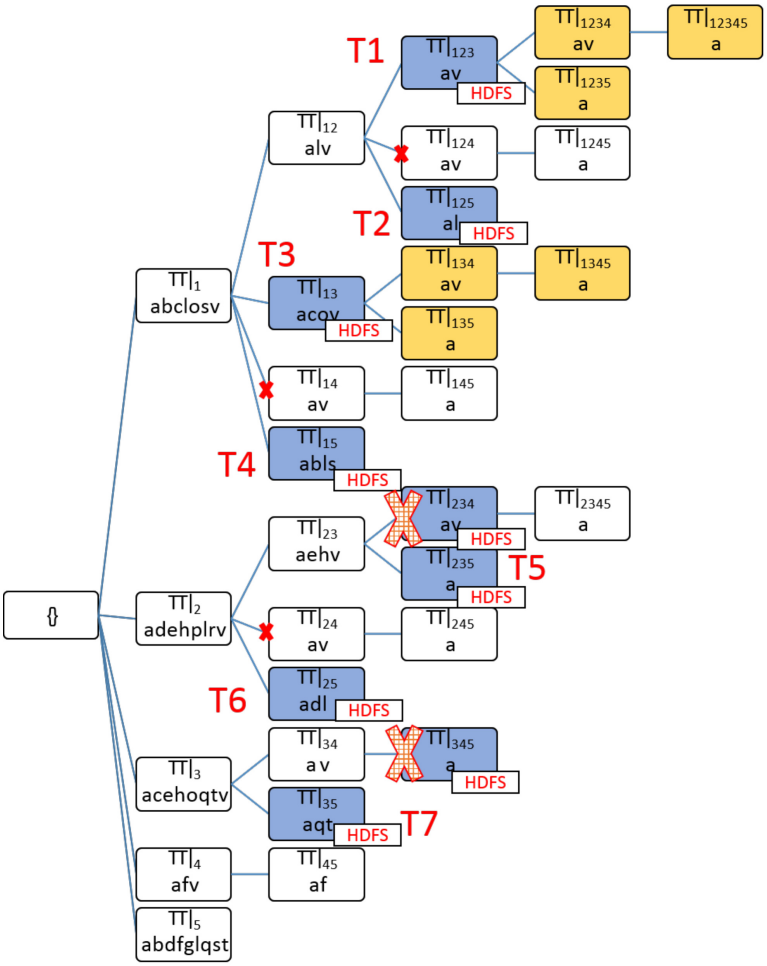

Figure 5: Execution of PaMPa-HD on the running example dataset. For the sake of clarity, pruning rules 1 and 2 are not applied. The big checked crosses on nodes represent the nodes which have been removed by the synchronization job, e.g., the one on node {2 3 4} represents the pruning of node {2 3 4}.

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
```

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.

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 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 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, 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| ≥ 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

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 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 6: Job 1 applied to the running example dataset ($minsup = 1$): local Carpenter algorithm is run from the Transposed Table 6d.

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, 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 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 synchroniza-

tion 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
```

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 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 algo-

rithms. 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 5.2).

Next, we evaluate the speed of the proposed algorithm, comparing it with the state-of-the-art distributed approaches (Section 5.3). Finally, we experimentally analyze the impact of (i) the number of transactions of the input dataset (Section 5.4), (ii) the number of parallel tasks (Section 5.5), and (iii) the resouce utilization and load balancing (Section 5.6).

Experiments have been performed on two real-world datasets. The first is the PEMS-SF dataset [13], 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 [14]). 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 5.4).

The second dataset is the Kent Ridge Breast Cancer [15], 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 [5]). The discretized versions of the real datasets are publicly available at http://dbdmg.polito.it/PaMPa-HD/.

Table 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 MapReduce API. The experiments were performed on two different configurations. The first, *Configuration 1*, consists of a cluster of 5 nodes running the 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. The configuration assumes 17 contemporary independent Yarn containers (tasks) of 6 GB of memory. *Configuration 2* consists of a larger shared Hadoop cluster of 30 nodes 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). We were able to work with 80 contemporary independent Yarn containers (tasks), each one characterized by 4GB of main memory.

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, 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 at task level, due 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, using Configuration 1 and 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 7 are shown the results in terms of execution time and

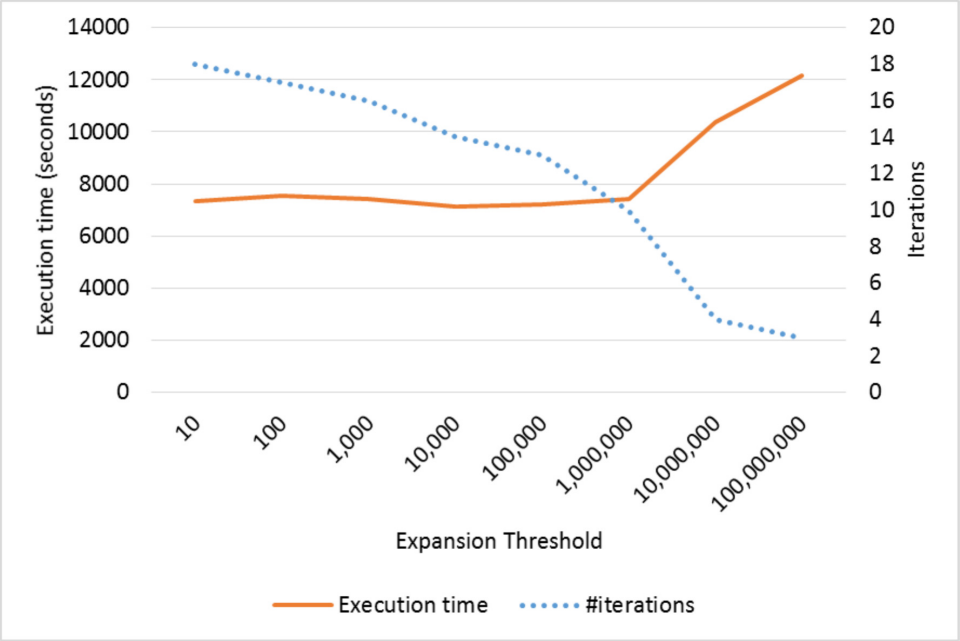

Figure 7: Execution time and number of iterations for different *max_exp* values on PEMS-SF dataset with *minsup*=10. and Configuration 1.

number of iterations (i.e., the number of jobs)[1]. 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.

The same experiment is repeated with the Breast Cancer dataset and a minsup value of 5. As shown in Figure 8, even in this case, the best

[1]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.

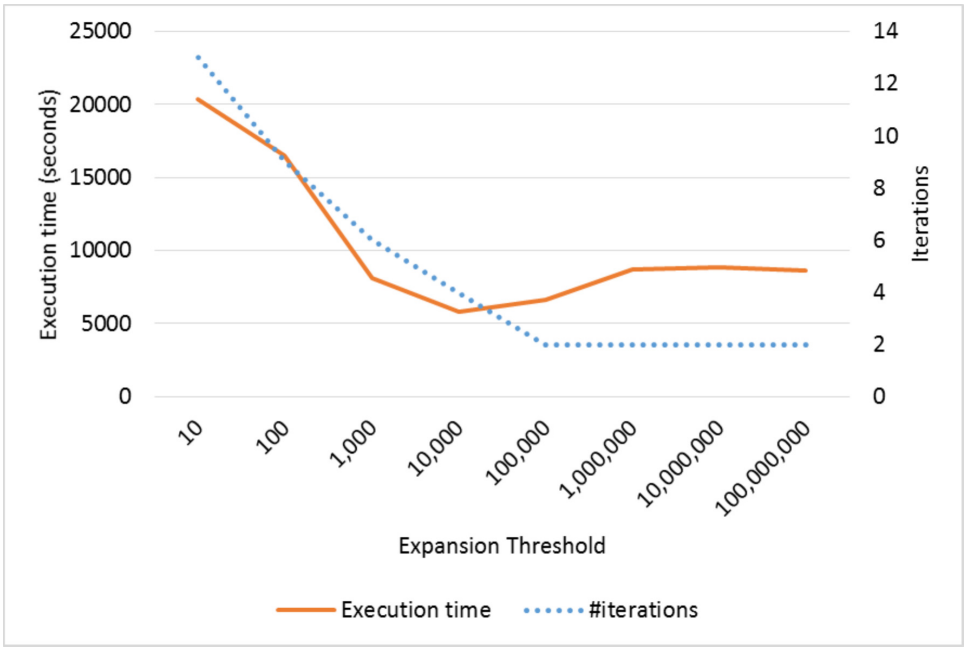

Figure 8: Execution time and number of iterations for different *max_exp* values on Breast Cancer dataset with *minsup*=5. and Configuration 1.

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 *max_exp* choice has a non-negligible impact on the performances of the algorithm. However, as demonstrated by the curves in Figures 7 and 8, it is very dependent on the distribution of the data and on the tree exploration. It is clear how the benefits of a more effective centralized pruning due to higher *max_exp* are enhanced for dense datasets such as PEMS-SF. For this dataset, for a large range of *max_exp* values, the benefits of the additional synchronization are almost completely mitigated by the weaker task level pruning impact and the additional iterations overhead. The motivation is

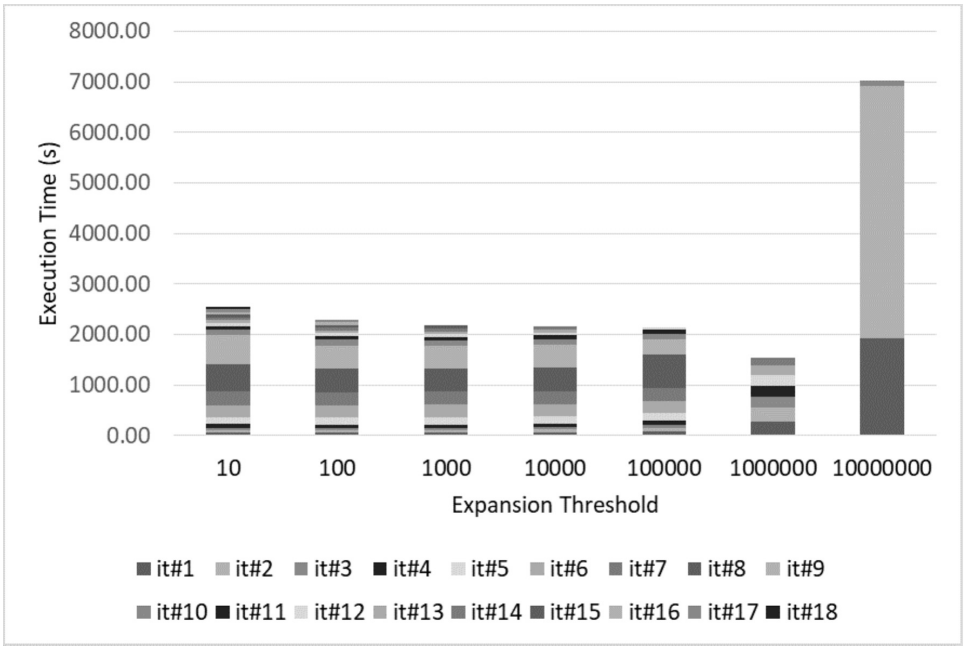

Figure 9: Execution time divided per iteration for different *max_exp* values on PEMS-SF dataset with *minsup*=10. and Configuration 2.

related to the impact of *max_exp* on the number of synchronization phases and the pruning effectiveness of pruning rule 3. When *max_exp* increases, the pruning at task level increases as well, while the number of synchronizations decreases together with their related overhead due the HDFS interactions (the temporary itemsets and related data are stored in HDFS). However, limiting the number of synchronizations has a negative impact on the global pruning effectiveness of pruning rule 3, that is applied less frequently and hence the probability of extracting multiple times the same (useless) itemsets increases. For PEMS-SF (the denser dataset), the overhead given by the synchronization operations is balanced by pruning rule 3 for a large range of values of *max_exp* (up to 1.000.000), because of the enhancend impact of the

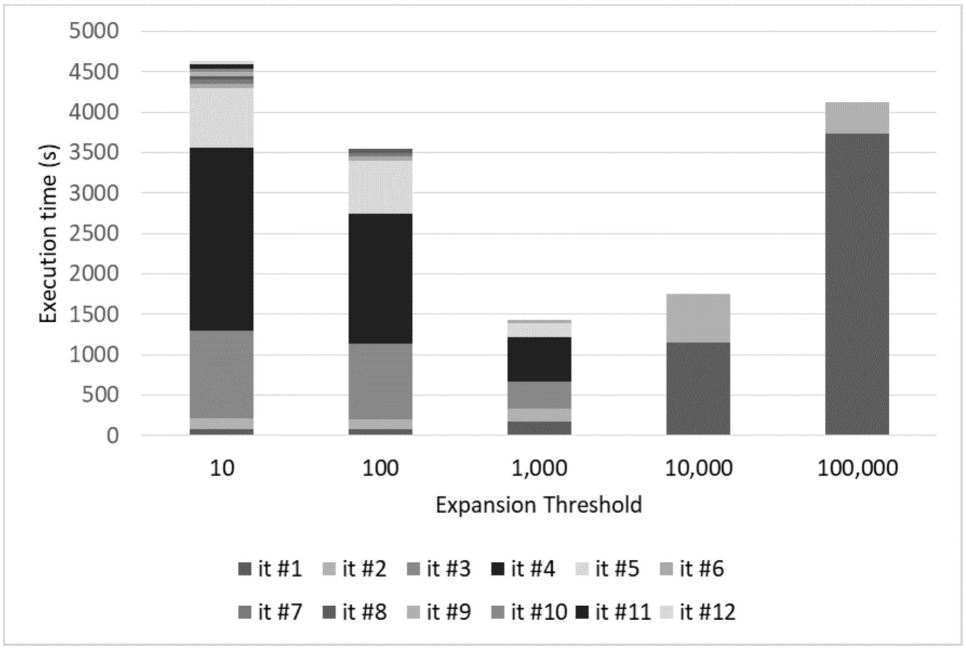

Figure 10: Execution time divided per iteration for different *max_exp* values on Breast Cancer dataset with *minsup*=5. and Configuration 2.

task level pruning in such a dense dataset. Differently, for Breast Cancer (the sparser dataset) , the negative impact of the synchronization phase overhead is initially higher than the positive impact of the application of pruning rule 3 (this is true up to 10.000). The main reason is that, since the dataset is sparse, pruning rule 3 is less effective when short "iterations" are performed (few itemsets are mined and hence the pruning impact of rule 3 is limited).

We run the same experiments with Configuration 2. In Figures 9 and 10 we reported the performance of the algorithm with respect to the expansion threshold parameter, highlighting the length (execution time) of each iteration of the mining. It is clear how the length of the last iterations is strongly reduced with respect to the central ones. Figures 11 and 12, in-

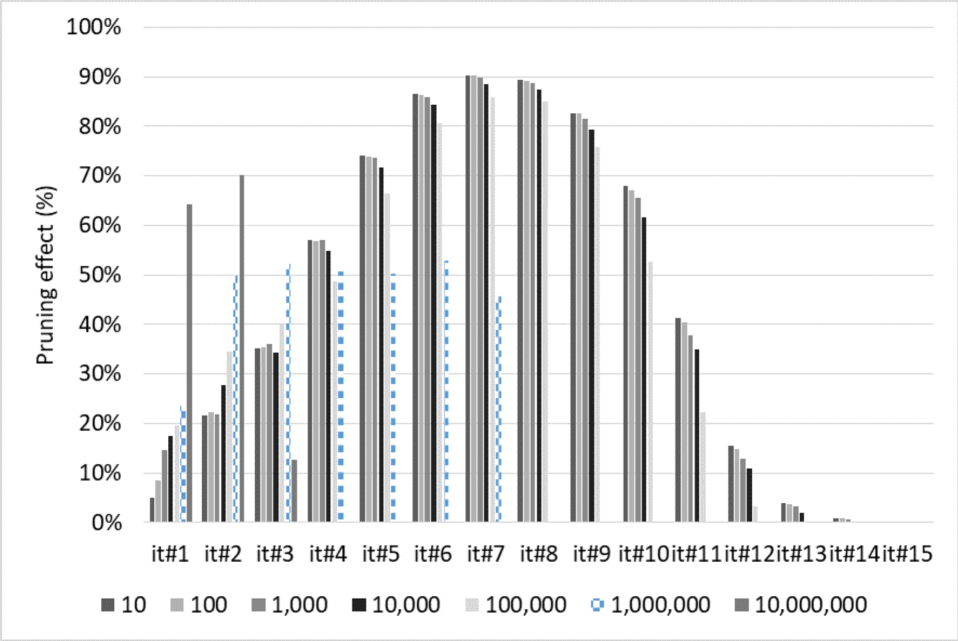

Figure 11: Pruning impact in terms of redundant tables and itemsets produced in each iteration, PEMS-SF dataset with *minsup*=10 and Configuration 2.

stead, plot the pruning impact of the synchronization phase, i.e. the number of elements (tables or closed itemsets) that are deleted. These elements are redundant and their generation is caused by the parallelization which decreases the effect of the centralized pruning. The higher the pruning effect, the more useless elements are produced and, hence, discarded in the synchronization phase. From the trend it is clear how large maximum expansion threshold value configurations are characterized by a greater number of deleted elements between the iterations. On the contrary, low values and frequent synchronization lead to less redundant elements to be deleted and a better optimization of the whole process at the cost of a higher number of iterations. Interestingly, the best configurations for both datasets are the

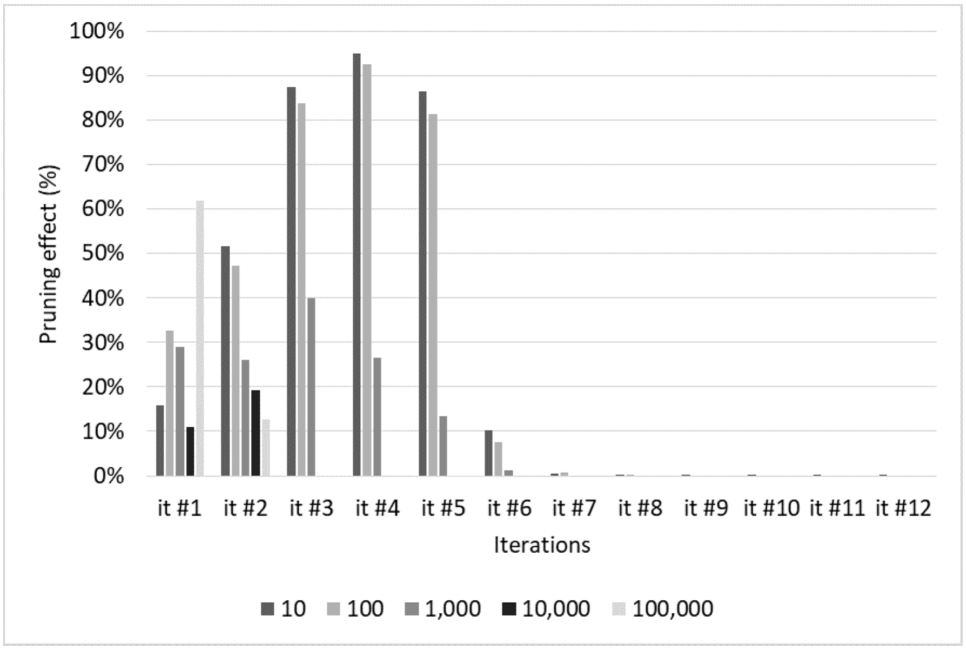

Figure 12: Pruning impact in terms of redundant tables and itemsets produced in each iteration, Breast Cancer dataset with *minsup*=5 and Configuration 2.

ones related the most steady pruning effect along all the iterations. This is particularly evident for PEMS-SF dataset in Figure 11 (scattered bars). The value of *max_exp* impacts also on 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 2 reports the minimum and the maximum execution time of the mining tasks executed in parallel for both datasets, Configuration 1 and for two extreme values of *max_exp*. The load balance is better for the lowest value of *max_exp*.

In the next subsection we introduce and motivate some tuning strategies

Table 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

related to *max_exp*.

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 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 [16]). 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

Table 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

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.

For Configuration 1, *Strategy #1* is the one achieving the best performances for both datasets. Table 4 reports the best performance for each configuration, in terms of relative performance difference with the best results obtained with a fixed *max_exp* parameter. As shown in Table 4, the results among the datasets and the configurations are quite different. It is clear how the higher parallelization degree decreases the effect of the centralized pruning. For this reason, the mining with Configuration 2 should be synchronized more frequently with respect to Configuration 1. Breast Cancer data distribution better fits the growth of the parameter, as shown by the better results with respect to the PEMS-SF dataset. The benefits of

Table 4: Best strategies performance

Configurations	PEMS-SF	Breast Cancer
Configuration 1	Strategy 1 (X = 10, -6,48%)	Strategy 1 (X = 10,000 -19,03%)
Configuration 2	Fixed Max_exp (1,000,000)	Strategy 1 (X = 10, -25,12%)

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 identified best parameter configurations will be used to compare PaMPa-HD with other distributed approaches.

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 [17] available in Mahout 0.9 [18], based on the FP-growth algorithm [12]
2. DistEclat [19], based on the Eclat algorithm [20]
3. BigFIM [19], inspired from the Apriori [11] 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 are discussed in detail in Section 6.

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 [13] and minsup values 35 to 10.[2]

As shown in Figure 13, 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 10 rows or 10%. 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

[2]The algorithms parameters, which will be introduced in Section 6, 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 [19], when possible (i.e. when there were enough 3-itemsets to execute also the second phase of the mining).

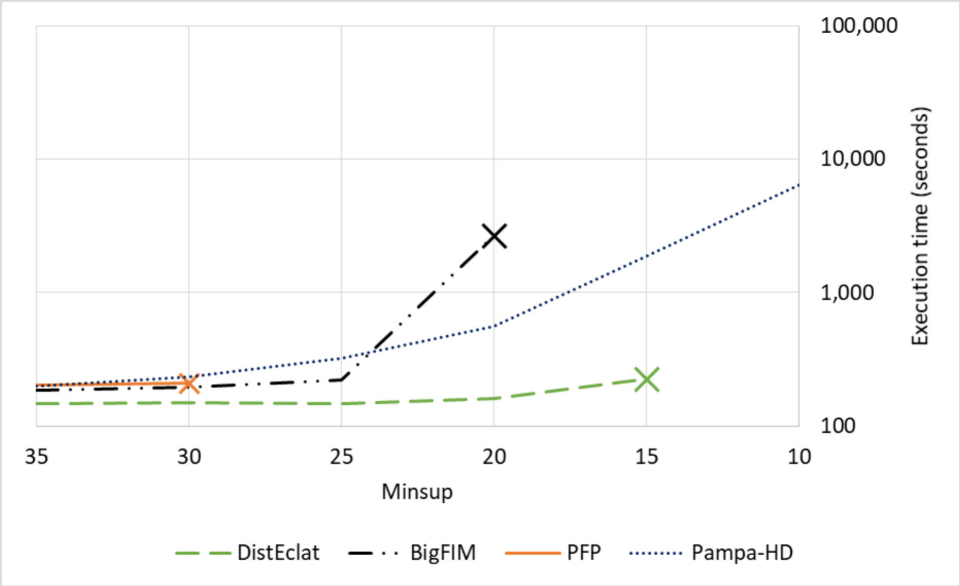

Figure 13: Execution time for different Minsup values on the PEMS-SF dataset (100-rows) and Configuration 1.

a minsup of 20, running out of memory with a minsup of 15. DistEclat, instead, 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 [15]. As reported in Figure 14 (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

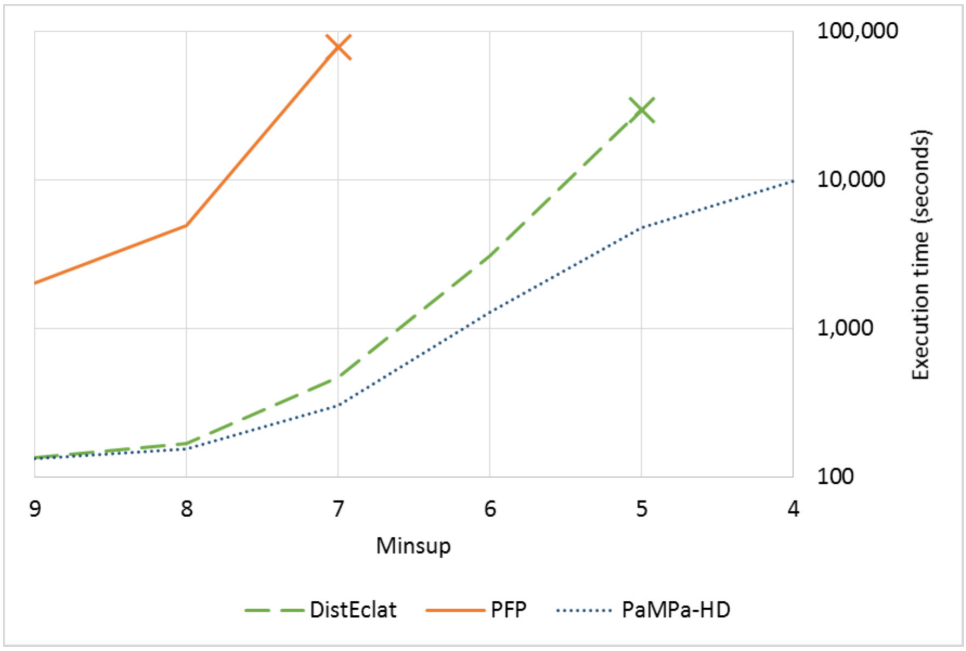

Figure 14: Execution time for different Minsup values on the Breast Cancer dataset and Configuration 1.

days of computation). We repeated the same experiments with Configuration 2. The results (see Figures 15 and 16) are similar to the previous ones, with PaMPa-HD demonstrating to be the most reliable solution. Actually, it is the only approach able to benefit of the higher parallelization degree. In fact, PFP does not show any improvement and it has been interrupted after 12 hours of processing. BigFIM and DistEclat performance are worsened by the less amount of memory available for each process.

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. In some cases characterized by relatively high minsup

thresholds, some of the state of the art approaches perform slightly better than PaMPa-HD, even if they are not specifically designed for high dimensional data. This is due to the fact that high support thresholds limit the number of frequent items, hence the dimensionality of the problem decreases, limiting the advantages of PaMPa-HD. Specifically, the issue with PaMPa-HD is that its design includes an interleaving synchronization job allowing to prune redundant itemsets. When the minsup is high, the benefits of this additional phase are less effective than the cons related to the I/O costs and the iterative architecture. This is more evident with PEMS-SF, due to its density and the production of more intermediate tables. For lower minsup values, PaMPa-HD 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.

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 2, from the root of the tree, it is generated a new branch for each tid

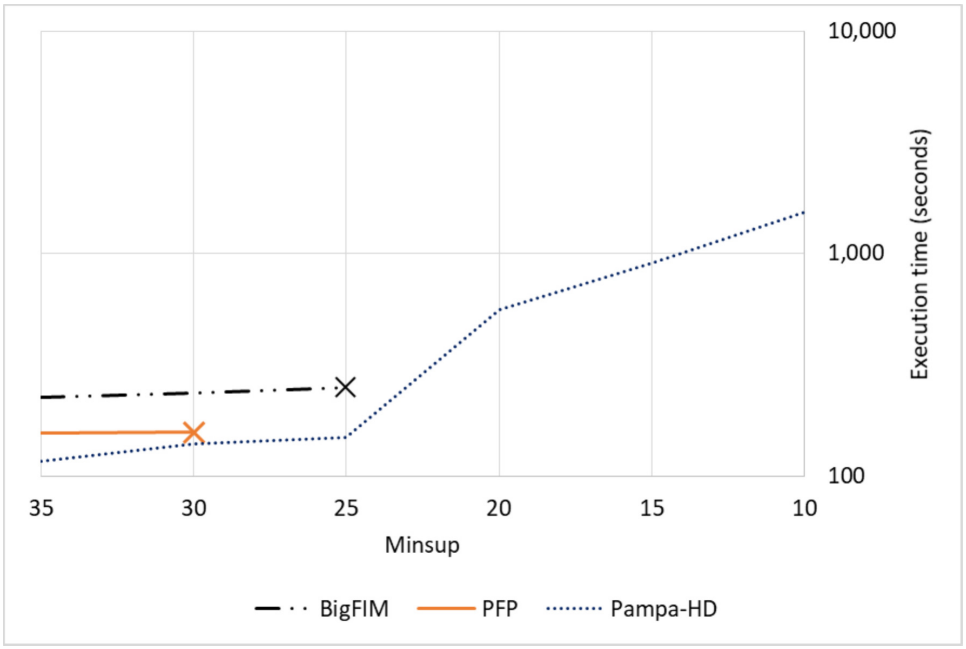

Figure 15: Execution time for different Minsup values on the PEMS-SF dataset and Configuration 2.

(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 Figures 17 and 18 , where, for instance, with a minsup of 35%, the

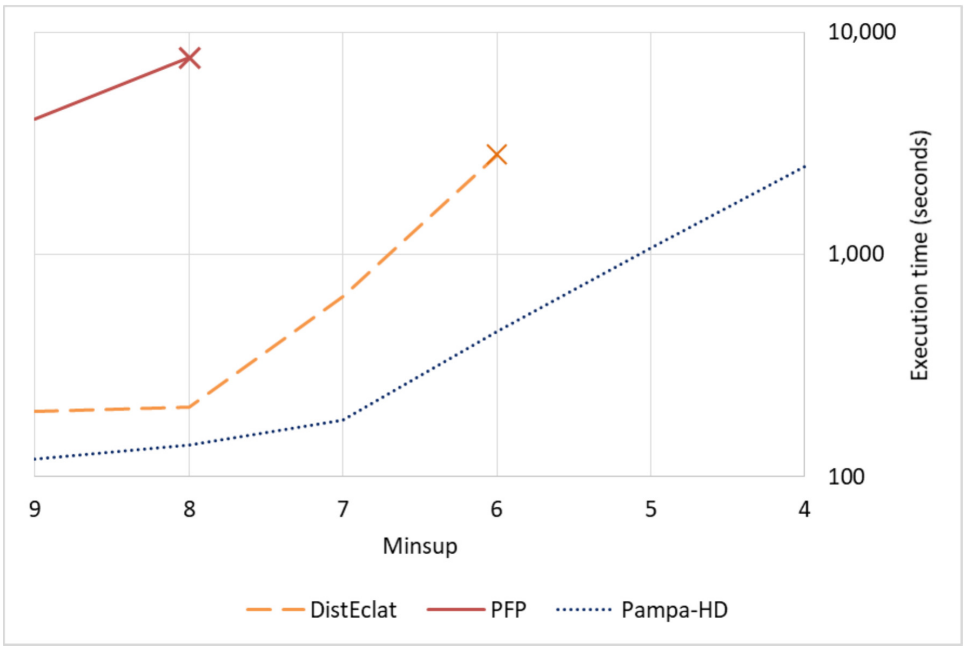

Figure 16: Execution time for different Minsup values on the Breast Cancer dataset and Configuration 2.

execution times related to the 100-rows and the full version of the PEMS-SF dataset differ of almost two orders of magnitude.

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.

5.5. Impact of the parallelization degree

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

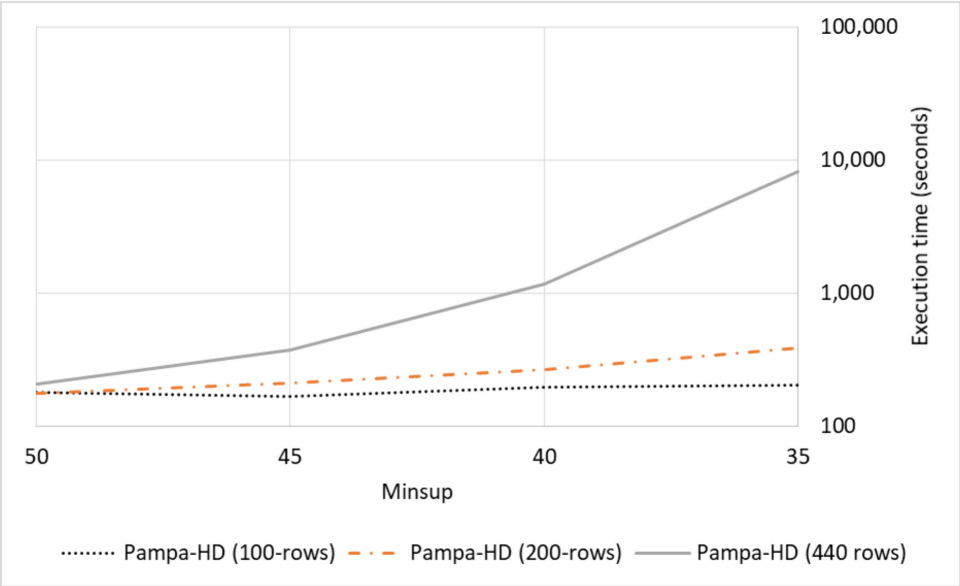

Figure 17: Execution times for different versions of PEMS-SF for PaMPa-HD and Configuration 1.

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 19 and Figure 20, 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 the greatest drop in terms of execution time when the computation passes from 5 to 20 nodes. The experiment of Breast Cancer instead, Figure 20, shows a stronger performance gain. As before, the behavior is related to

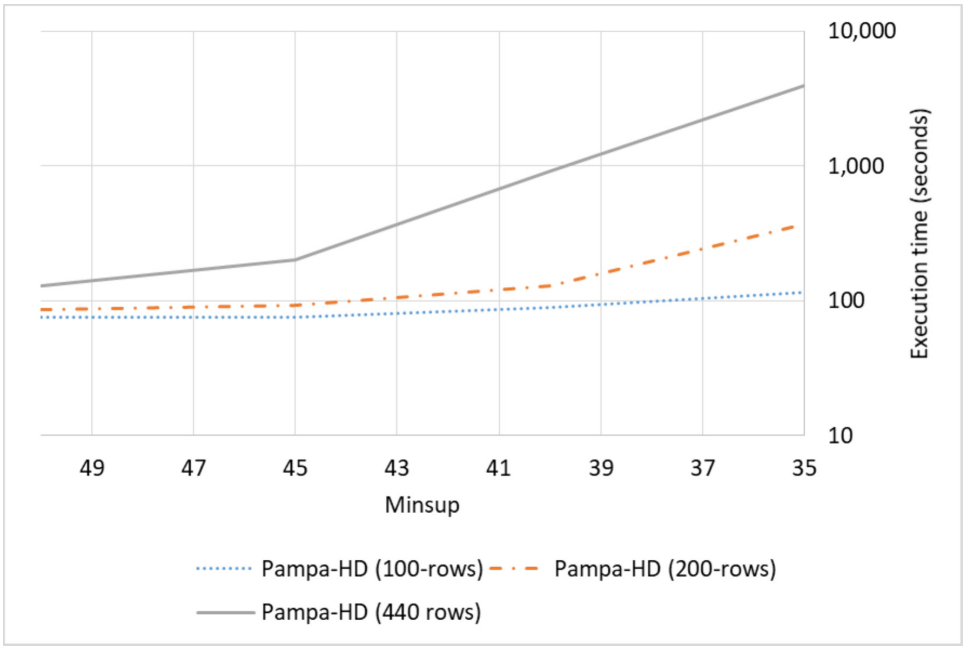

Figure 18: Execution times for different versions of PEMS-SF for PaMPa-HD and Configuration 2.

the dataset data distribution which causes the PEMS-SF mining process generating more intermediate tables. In this case, the advantages related to 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.

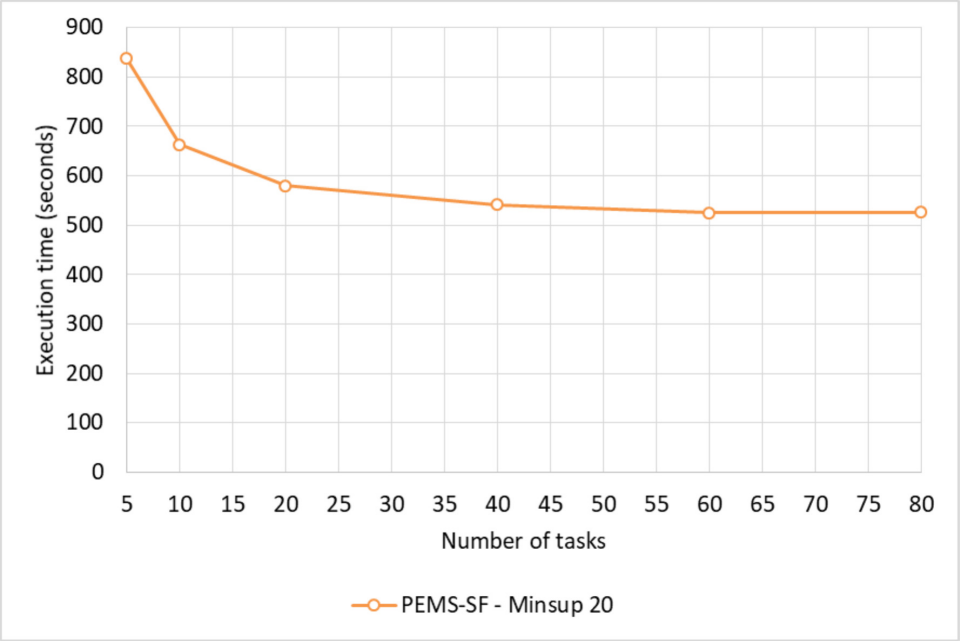

Figure 19: Execution times for PEMS-SF dataset with different number of parallel tasks based on Configuration 2.

5.6. Load Balancing and resources utilization

The last analyses are related to the load balancing and the resources utilization of the algorithm. These issues represent very important factor in such a distributed environment. Communication costs, for instance, are among the main bottlenecks for the performance of parallel algorithms [21]. 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 3). For this reason, each task needs the information

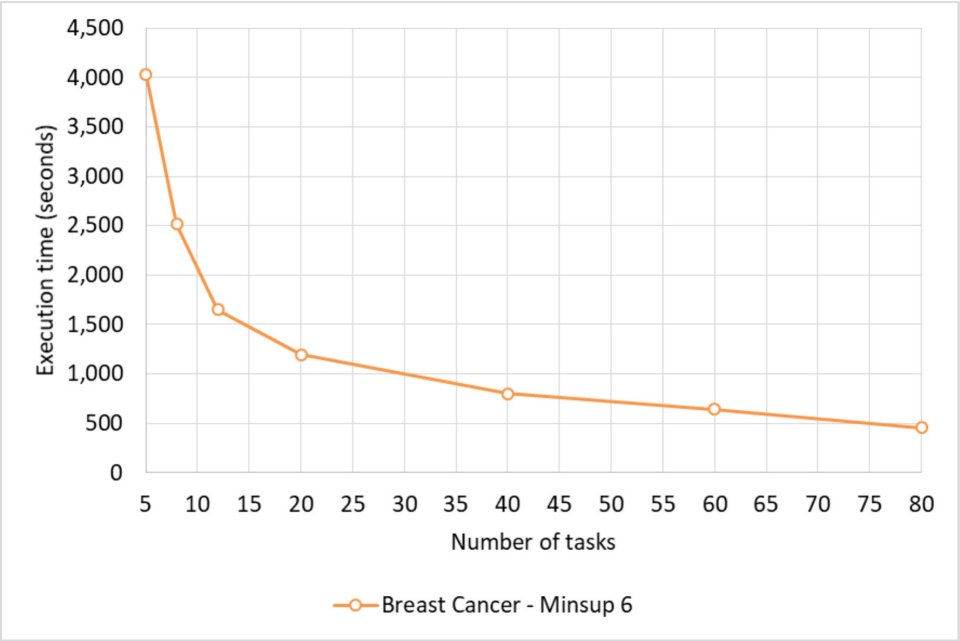

Figure 20: Execution times for Breast Cancer dataset with different number of parallel tasks based on Configuration 2.

(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 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 7 and Figure 8), in some cases additional synchronization phases lead to better execution times, despite their related overhead.

We measured the resource utilization in terms of disk usage (read and write phases of HDFS), network communication, and CPU usage. Please note that the values are normalized with respect to the maximum resource utilization. Specifically, Figure 21 and 22 report the achieved results for the two datasets in an insulated hardware configuration. 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. As regards CPU utilization (Figure 22) the degradation is due to the completion of some of the tasks. The higher *max_exp*, as already mentioned, has the counter effect of decreasing the load balance. The trend is, in fact, more flat for the mining of PEMS-SF dataset (Figure 21), characterized by more frequent synchronizations.

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 unbalanced phase of the job is, not surprisingly, the

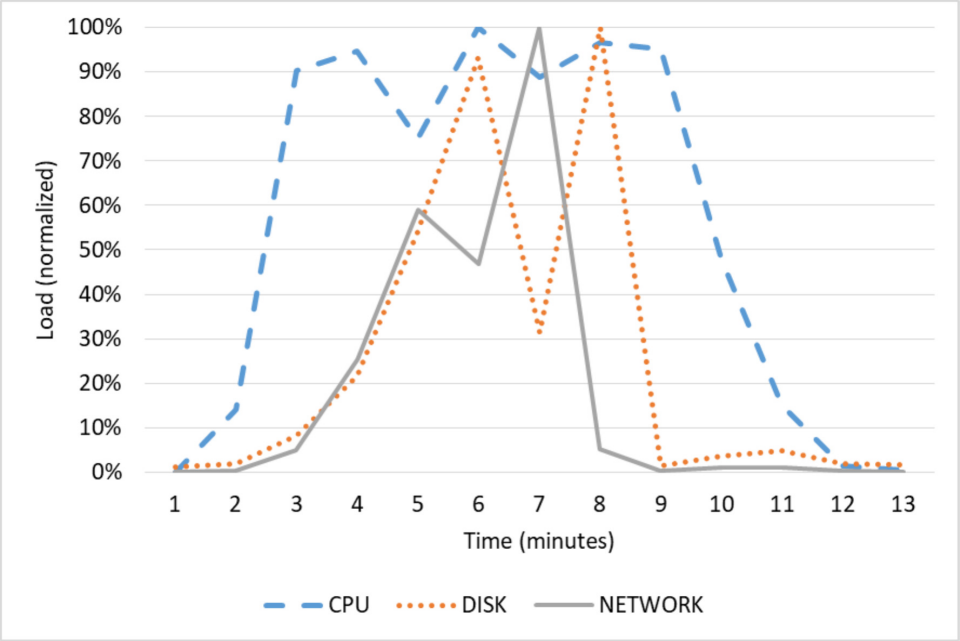

Figure 21: Resource utilization, PEMS-Cancer dataset, minsup=25.

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 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 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.

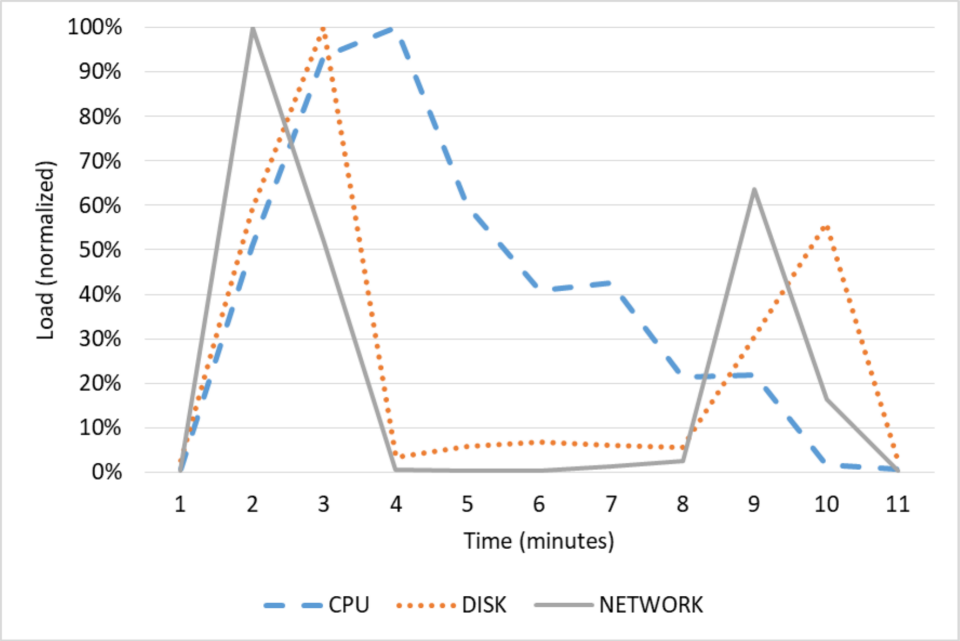

Figure 22: Resource utilization, Breast Cancer dataset, minsup=7.

6. Related work

Frequent itemset mining represents a very popular data mining technique used for exploratory analysis. Its popularity is witnessed by the high number of approaches and implementations. The most popular techniques to extract frequent itemsets from a transactional datasets are Apriori and Fp-growth. Apriori [11] is a bottom up approach: itemsets are extended one item at a time and their frequency is tested against the dataset. FP-growth [12], instead, is based on an FP-tree transposition of the transactional dataset and a recursive divide-and-conquer approach. These techniques explore the search space enumerating the items. For this reason, they work very well for datasets with a small (average) number of items per row, but their running time increases exponentially with higher (average) row lengths [11, 20].

Table 5: Load Balancing, Configuration 1

Dataset	Slowest Task Execution time	Fastest Task Execution time
PEMS-SF (minsup = 20)	3mins 58 sec	3mins 37sec
Breast Cancer (minsup = 6)	20mins 33sec	8mins 42sec

In recent years, the availability of Big Data technologies allowed the implementation of these techniques in distributed environments such as Apache Hadoop [3], based on the MapReduce paradigm [22], and Apache Spark [4]. Parallel FP-growth [17] is the most popular distributed closed frequent itemset mining algorithm. The main idea is to process more sub-FP-trees in parallel. A dataset conversion is required to make all the FP-trees independent. A Spark implementation of Parallel FP-growth has been delivered with MLlib Library [23]. This version extracts all the frequent itemsets and not just the closed ones. BigFIM and DistEclat [19] are two recent methods to extract frequent itemsets. DistEclat represents a distributed implementation of the Eclat algorithm [20] an approach based on equivalence classes (groups of itemsets sharing the same prefixes), smartly merged to obtain all the candidates. BigFIM is a hybrid approach exploiting both the Apriori and Eclat paradigms. BigFIM and DistEclat are divided in two phases. In the first one, the approaches use respectively an Apriori-like and Eclat-like strategy to mine the itemsets up to a fixed k-length. After that, the itemsets are distributed and used as prefixes for the longer itemsets. In the last phase, both approaches use Eclat to extract all the closed itemsets. In addition, [24] introduces another Apriori-based frequent itemset miner. The

contribution of this work is focused on the candidates handling, which are cached in memory between each iteration. In [25], a similar breadth-first approach is introduced, but with the exploitation of a matrix-based pruning in order to significantly reduce the amount of candidates. In [26], the breadth-first exploration manner is combined with the suffix-based candidate generation. Finally, for the environments requiring very fast response, some sampling-based techniques have been presented [27], [28] and [29]. These works are characterized by getting a trade-off between execution time and quality of the results. While the previous works have been designed for use cases characterized by datasets with a large amount of transactions, Carpenter algorithm [5], which inspired PaMPa-HD, has been specifically designed to extract frequent itemsets from high-dimensional datasets, i.e., characterized by a very large number of attributes (in the order of tens of thousands or more). The basic idea is to investigate the row set space instead of the itemset space. The idea of designing a parallel MapReduce algorithm to efficiently support itemset mining on high dimensional data was first introduced in [9]. The PaMPa-HD algorithm significantly enhances the algorithm performance proposed in [9] by providing (i) a more efficent approach to address synchronization phase, reducing the number of MapReduce jobs; (ii) a more efficient visit of the transposed tables; (iii) and a set of self-tuning strategies to speed up the performances through a dynamic modification of the max_exp parameter . Furthermore, this work introduces a wider set of experiment to evaluate, on real datasets, the impact of the number of transaction on the performance, but also communication costs and load balancing, very important in a distributed environment.

This work extends our previous work [9]. The original algorithm exploits an additional independent synchronization job at each iteration. As already described in Section4.1, this implementation includes the synchronization phase in the Mining Job 3. Therefore, the number of MapReduce jobs (with their related overhead) are strongly reduced. Additionally, in order to better exploit the pruning rule in the local Carpenter iteration in each independent task, all the transposed tables are now processed (not only expanded) in depth-first order. This strategy decreases the possibility to explore an useless branch of the tree, i.e. a branch whose results would be completely overwritten by the closed itemsets obtained by branches older in depth-first fashion. For instance, the performance improvement from the previous version, measured with Breast Cancer Dataset (minsup=6) is from 6% to 30% (depending on the number of independent tasks).

7. 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. [30], [31]). Consequentely, many fields of applications which exploits frequent itemset to discover hidden correlations and association rules [32] could benefit of it. The first example is bioinformatics [33] 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 ben-

efit 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 [34], [35]. Collected datasets, that include traffic flows in which the item are flow attributes ([36], [37], [38]), represent an appealing domain where PaMPa-HD can be efficiently exploited. are already a very promising application domain for data mining techniques.

8. Conclusion

This work introduced PaMPa-HD, a novel frequent closed itemset mining algorithm able to efficiently parallelize the itemset extraction from extremely high-dimensional datasets. The algorithm's architecture mitigates the disadvantages of the parallelization of a search-space exploration which strongly benefits of a centralized state. The introduction of the algorithm is followed by an exhaustive experimental analysis. We firstly measured the impact of different parameter configurations and dataset distribution on the execution time and on the whole mining (number of iterations, their duration and the related pruning effect). In order to improve the performance, we explored the introduction of self-tuning techniques, whose efficiency revealed to be strongly related to the dataset distribution. To better assess the efficiency of the algorithm, we compared our approach with the best state of the state-of-the-art algorithms. PaMPa-HD outperformed all of them, by

showing a better scalability than all popular distributed approaches, such as PFP, DistEclat and BigFIM. Despite the algorithm design is strongly focused on high-dimensional use-cases, we measured the impact of the number of transactions on the overall performances. This experiment evidenced the difficulty for row-enumeration-tree-based approaches to deal with large amount of rows. We measured the behavior of our algorithm with different parallelization degrees, i.e. number of parallel tasks. Even in this experiment, leveraging up to 80 different tasks, the performances are skewed by the dataset distribution, since a higher parallelization degree limits the impact of the pruning at task level. Finally, we tracked the resource utilization and the load balancing. Once again, the experiment helped us to understand the impact of the number of synchronizations. Specifically, load balancing degradation demonstrated to be one of the major cons of a high max_exp values.

9. Future work

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 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. Finally, since all the strategies currently focus on the increasing of the max_exp parameter in order to eqaualize the wall-clock time duration of all the iterations, a future work could be more focused on a pruning impact (i.e. number of redundant elements deleted in each synchronization phase) normalization among all the iterations.

Acknowledgement

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

[1] X. Jin, B. W. Wah, X. Cheng, Y. Wang, Significance and challenges of big data research, Big Data Research 2 (2) (2015) 59 – 64, visions on Big Data. doi:http://dx.doi.org/10.1016/j.bdr.2015.01.006.

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

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

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

[5] F. Pan, G. Cong, A. K. H. Tung, J. Yang, 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, KDD '03, ACM, New York, NY, USA, 2003, pp. 637–642. doi:10.1145/956750.956832.

[6] J.-G. Lee, M. Kang, Geospatial big data: Challenges and opportunities, Big Data Research 2 (2) (2015) 74 – 81, visions on Big Data. doi:http://dx.doi.org/10.1016/j.bdr.2015.01.003.

[7] M. Cuturi, UCI machine learning repository. PEMS-SF dataset (2011). URL `https://archive.ics.uci.edu/ml/datasets/PEMS-SF`

[8] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, `http://snap.stanford.edu/data` (Jun. 2014).

[9] D. Apiletti, E. Baralis, T. Cerquitelli, P. Garza, P. Michiardi, 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. doi:10.1109/ICDMW.2015.18.

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

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

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

[13] M. Lichman, UCI machine learning repository (2013).
URL http://archive.ics.uci.edu/ml

[14] California department of transportation.
URL http://http://pems.dot.ca.gov/. Last access: April, 21st 2016

[15] M. L. data set repository, Breast cancer dataset (kent ridge.
URL http://mldata.org/repository/data/viewslug/breast-cancer-kent-ridge-2 Last access: July, 21st 2016

[16] V. Jacobson, Congestion avoidance and control, SIGCOMM Comput. Commun. Rev. 18 (4) (1988) 314–329. doi:10.1145/52325.52356.

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

[18] Apache Software Foundation. Apache mahout:: Scalable machine-learning and data-mining library [online, cited 2016-03-15].

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

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

[21] A. D. Sarma, F. N. Afrati, S. Salihoglu, J. D. Ullman, Upper and lower bounds on the cost of a map-reduce computation, Proc. VLDB Endow. 6 (4) (2013) 277–288. doi:10.14778/2535570.2488334.

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

[23] 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, A. Talwalkar, MLlib: Machine learning in apache sparkarXiv:1505.06807.

[24] H. Qiu, R. Gu, C. Yuan, Y. Huang, Yafim: A parallel frequent itemset mining algorithm with spark, in: Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International, IEEE, 2014, pp. 1664–1671.

[25] F. Zhang, M. Liu, F. Gui, W. Shen, A. Shami, Y. Ma, A distributed frequent itemset mining algorithm using spark for big data analytics, Cluster Computing 18 (4) (2015) 1493–1501.

[26] Y.-h. Liang, S.-y. Wu, Sequence-growth: A scalable and effective frequent itemset mining algorithm for big data based on mapreduce framework, in: 2015 IEEE International Congress on Big Data, IEEE, 2015, pp. 393–400.

[27] M. Riondato, E. Upfal, Mining frequent itemsets through progressive sampling with rademacher averages, in: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2015, pp. 1005–1014.

[28] S. Gole, B. Tidke, Frequent itemset mining for big data in social media using clustbigfim algorithm, in: Pervasive Computing (ICPC), 2015 International Conference on, IEEE, 2015, pp. 1–6.

[29] X. Wu, W. Fan, J. Peng, K. Zhang, Y. Yu, Iterative sampling based frequent itemset mining for big data, International Journal of Machine Learning and Cybernetics 6 (6) (2015) 875–882. doi:10.1007/s13042-015-0345-6.

[30] R. Vimieiro, P. Moscato, A new method for mining disjunctive emerging patterns in high-dimensional datasets using hypergraphs, Information Systems 40 (2014) 1 – 10. doi:http://dx.doi.org/10.1016/j.is.2013.09.001.

[31] P. Bermejo, L. de la Ossa, J. A. Gmez, J. M. Puerta, Fast wrapper feature subset selection in high-dimensional datasets by means of filter re-ranking, Knowledge-Based Systems 25 (1) (2012) 35 – 44, special Issue on New Trends in Data Mining. doi:http://dx.doi.org/10.1016/j.knosys.2011.01.015.

[32] B. Kamsu-Foguem, F. Rigal, F. Mauget, Mining association rules for the quality improvement of the production process, Ex-

pert Systems with Applications 40 (4) (2013) 1034 – 1045. doi:http://dx.doi.org/10.1016/j.eswa.2012.08.039.

[33] J. Nahar, T. Imam, K. S. Tickle, Y.-P. P. Chen, Association rule mining to detect factors which contribute to heart disease in males and females, Expert Systems with Applications 40 (4) (2013) 1086 – 1093. doi:http://dx.doi.org/10.1016/j.eswa.2012.08.028.

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

[35] B. Claise, Cisco systems netflow services export version 9. rfc 3954 (informational) (2004).

[36] D. Apiletti, E. Baralis, T. Cerquitelli, S. Chiusano, L. Grimaudo, Searum: A cloud-based service for association rule mining, in: Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TRUSTCOM '13, IEEE Computer Society, Washington, DC, USA, 2013, pp. 1283–1290. doi:10.1109/TrustCom.2013.153.

[37] D. Brauckhoff, X. Dimitropoulos, A. Wagner, K. Salamatian, Anomaly extraction in backbone networks using association rules, Networking, IEEE/ACM Transactions on 20 (6) (2012) 1788–1799. doi:10.1109/TNET.2012.2187306.

[38] D. Apiletti, E. Baralis, T. Cerquitelli, V. D'Elia, Characterizing network traffic by means of the netmine framework, Comput. Netw. 53 (6) (2009) 774–789. doi:10.1016/j.comnet.2008.12.011.