PaMPa-HD: EXTENSION - TO MODIFY

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

Frequent closed itemset mining, a data mining technique for discovering hidden correlations in transactional datasets, are among the most complex exploratory techniques in data mining environment. Even this domain has been involved, thanks to the spread of distributed and parallel frameworks, to the development of scalable approaches able to deal with the so called Big Data. Unfortunately, most of the them 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 parallel MapReduce-based frequent closed itemset mining algorithm for high dimensional datasets, inspired from Carpenter algorithm. The experimental results, performed on two real life high-dimensional use cases, show the efficiency of our approach. **Shall we mention that is an extension work?**

Keywords:

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", have changed dramatically the importance of the intelligent analysis of data. 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 [1] and Spark [2]. 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 [3], and no distributed implementations at all.

Nevertheless, many researchers in scientific domains such as bioinformatics or networking, often require to deal with this type of data. 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. Some smart-cities 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 [4]. 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 [4, 5].

This work introduces PaMPa-HD, a parallel MapReduce-based frequent closed itemset mining algorithm for high-dimensional datasets, based on the Carpenter algorithm. PaMPa-HD outperforms the single-machine Carpenter implementation and the best state-of-the-art distributed approaches, in both execution time and minimum support threshold. Furthermore, the implementation takes into account crucial design aspects, such as load balancing and robustness to memory-issues.

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 provides a brief review of the state of the art, and Section 7 discusses possible applications of PaMPa-HD. Finally, Section 8 introduces future works and conclusions.

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.,

$\mathcal{D}$	
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 $\mathcal{D}$

$t_i \subseteq \mathcal{I}$) and it is identified by a transaction identifier (tid_i). Figure 1a reports an example of a transactional dataset with 5 transactions. The dataset reported in Figure 1a is used as a 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 [6] 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 [3]. Closed itemsets allow representing the same information of traditional frequent itemsets in a more compact form.

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 6a 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 6a. 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

As discussed in section 6, the most popular techniques (e.g., Apriori [7] and FP-growth [8]) adopt the itemset enumeration approach to mine the frequent itemsets. However, itemset enumeration revealed to be ineffective with datasets with a high average number of items per transactions [3]. To tackle this problem, the Carpenter algorithm [3] 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 6a). As already described in Section 2, in the transposed representation each row of the table consists of an item i with its tidlist. For instance,

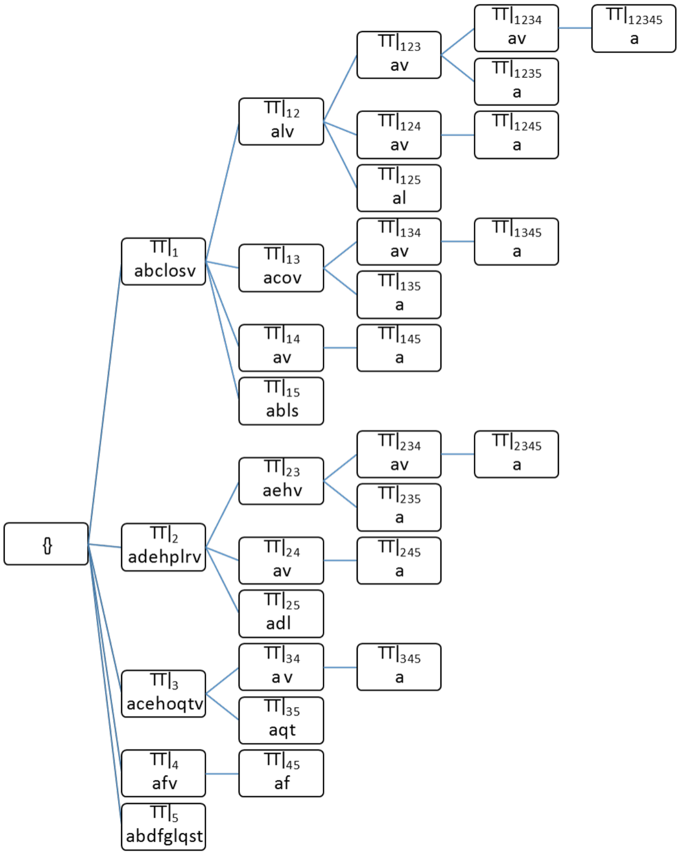

Figure 2: The transaction enumeration tree of the running example dataset in Figure 1a. For the sake of clarity, no pruning rules are applied to the tree.

the last row of Figure 6a points that item v appears in transactions 1, 2, 3, 4.

Carpenter builds a transaction enumeration tree where 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 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, 2) the conditional transposed table $TT|_X$, 3) the set of frequent closed itemsets found up to the current step of the tree search, and 4) the enforced minimum support threshold (*minsup*). Based on the theorems reported in [3], 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. At 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 architectures, like the ones we address in this work.

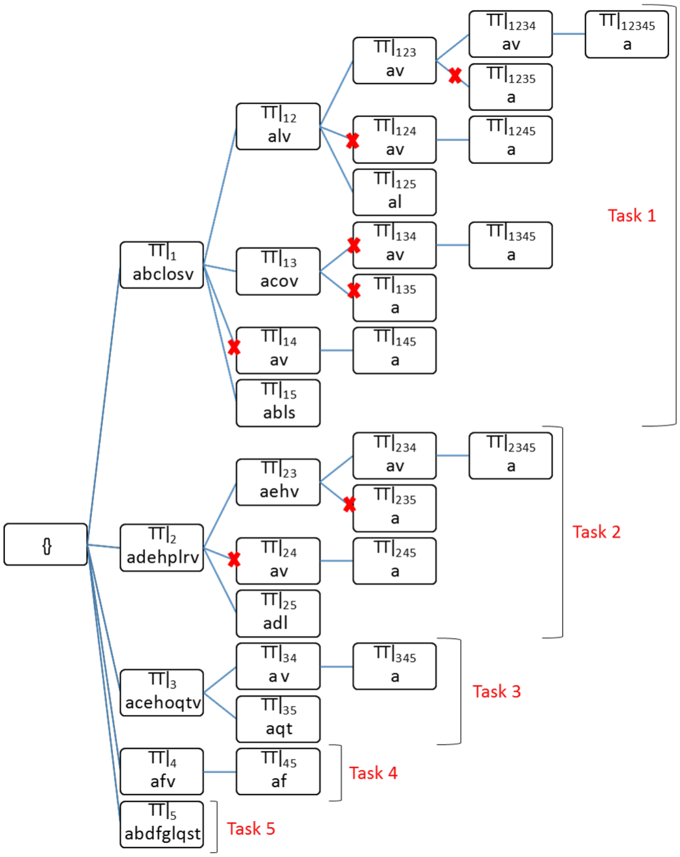

Figure 3: Running toy example: each node expands a branch of the tree independently. Pruning rule 1 and 2 are not applied. The pruning rule 3 is applied only within the same task: the red crosses on the edges represent pruned nodes due to local pruning rule 3, e.g. the one on node {2 4} represents the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

4. The PaMPa-HD algorithm

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. Carpenter also prunes part of the search space by applying the three pruning rules illustrated above. The PaMPa-HD algorithm proposed in this paper 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 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.

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 of the reduction of the search space, as detailed in Section 5, its inapplicability leads to a negative impact on the execution time of the distributed algorithm as described so far. 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 node 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 toghether 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, as shown in Figure 5, from the bucket B_{av}, it is kept the node $TT_{1,2,3}$.

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 ignore the frequent closed itemsets obtained in the previous iteration, which are just processed in the synchronization phase. 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. 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

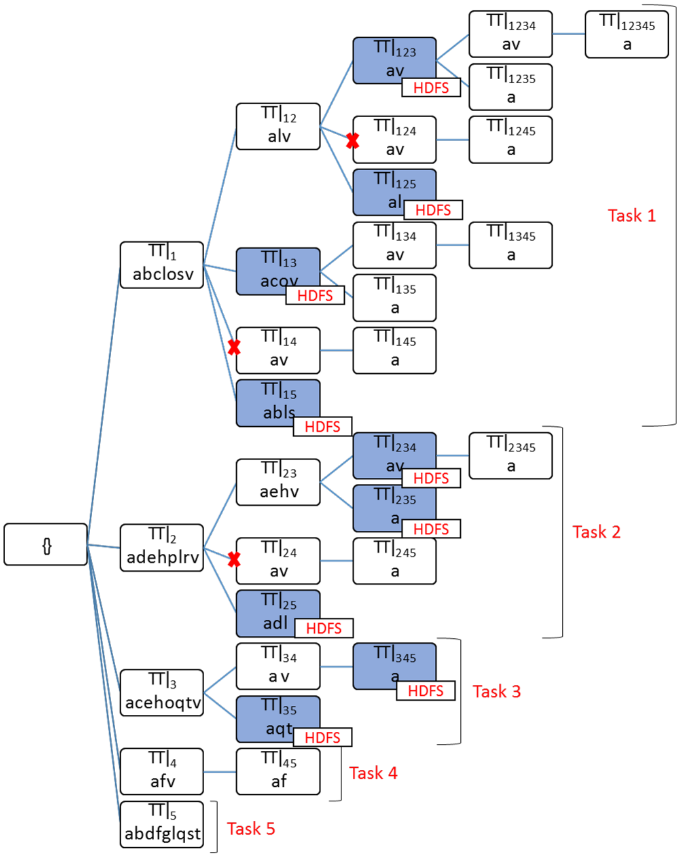

Figure 4: Execution of PaMPa-HD on the running example dataset. For sake of clarity, pruning rules 1 and 2 are not applied. The dark nodes represent the node that have been written to hdfs in order to apply the synchronization job

at the same time a very high reliability dealing with heavy enumeration trees, which can be split and pruned according to pruning rule 3.

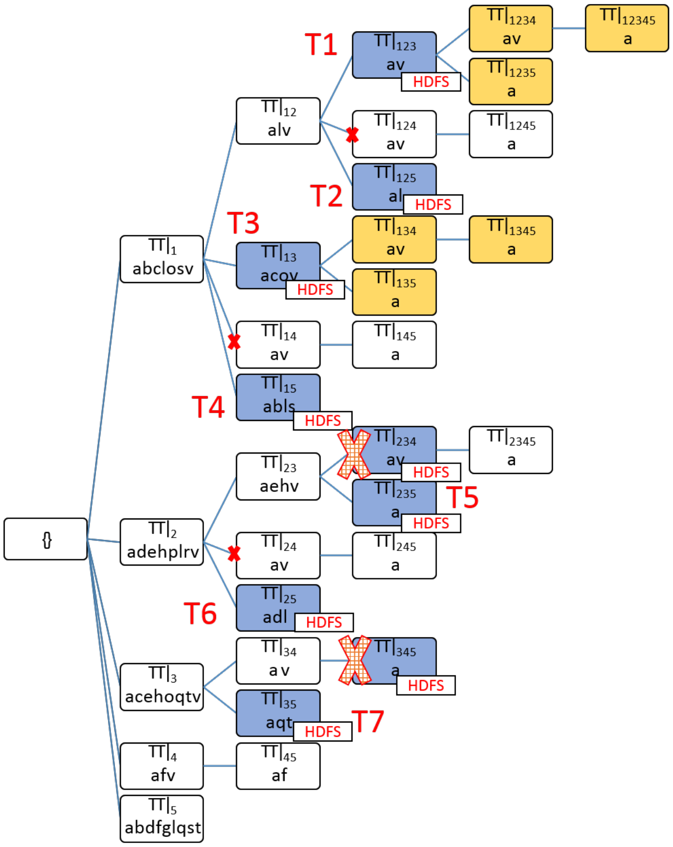

Figure 5: Execution of PaMPa-HD on the running example dataset. For sake of clarity, pruning rules 1 and 2 are not applied. The big checked crosses on nodes represent the nodes which have been removed by the synchronization job, e.g., the one on node {2 3 4} represents the pruning of node {2 3 4}.

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 (Figure 4.1).

PaMPa-HD pseudo code

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

The first job is developed to distribute the input dataset to the independent tasks, which will run a local version of the Carpenter algorithm. 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 4.1, 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.

- 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. The tables in Figure 6 illustrate the processing of a row of the initial Transposed representation of D. Reducers run a local Carpenter implementation from the input tables. 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 job is run. As already described in Section 3, Carpenter recursively processes a transposed table expanding it in a depth-first manner. At each iteration of the Carpenter subroutine, a counter is increased. When the count is over the given maximum expansion threshold, the main routine is not invoked anymore. In this case, all the intermediate results are written to HDFS.

1. the transposed table is composed using the tidlists from each key-value and a local Carpenter job is run
2. each recursion of the Carpenter subroutine increases a counter which is compared to the expansion threshold before each recursion
3. if the count is below the threshold another Carpenter recursion is scheduled
4. else, Carpenter main routine is not invoked anymore but all the intermediate results are written to HDFS

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. This information is required by the synchronization phase in order to establish which element is the eldest in a depth first exploration.

Job 1 Pseudo code

```
procedure MAPPER($minsup; item_i; tidlist\ TL$)
    for $j = 0$ to $|(TL)| - 1$ do
      tidlist $TL_{greater}$ : set of tids greater than
      the considered tid $t_j$.
        if $|TL_{greater}| \geq minsup$ then
            output <key= $t_j$; value= $TL_{greater}$, item>
        else Break
        end if
    end for
end procedure
procedure REDUCER($key = tid\ X, value = tidlists\ TL[\ ]$)
    Create new transposed table $TT|_X$
    for each tidlist $TL_i$ of $TL[\ ]$ do
        add $TL_i$ to $TT|_X$ (populate the transposed table)
    end for
    while $max\_exp$ is not reached do
        Run $Carpenter(minsup; TT|_X)$
    end while
        $Output{<}itemset;\ tidlist + Transposedtable\ I\ rows{>}$
    for each frequent closed itemset found do
        $Output({<}itemset; tidlist + support{>})$
    end for
end procedure
```

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 6a

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 key3

$TT|_{\{3\}}$

item	tidlist
a	4,5
c	-
e	-
h	-
o	-
q	5
t	5
t	5
v	4

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

Figure 6: Job 1 applied to the running example dataset: local Carpenter algorithm is run from the Transposed Table 6d.

After this phase, the synchronization job is launched (Job 2 pseudo code). It 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 itemset, the mappers output a pair of the form <key=itemset;value=tidlist,minsup>; for each tables the mappers out a pair of the form <key=itemset;value=tidlist,table_content>. The shuffle and sort phase delivers to the reducers the pairs aggregated by keys. The reducers, which matches the buckets introduced in Section 4, compare the entries and emit, for the same key or itemset, only the eldest version in a depth first exploration. For instance, referring to our running example in Figure 5, in the bucket of 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 are discarded with a pruning very similar to the one related to a centralized memory at the cost of a very MapReduce-like job.

Finally, the last MapReduce job can be seen as a mixture of the two previous jobs. As shown by Job 3 pseudo code, 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. From that moment, the tables are not expanded but sent to the reducers. Please note that the tree exploration processing the initial transposed tables in a depth-first order is more similar to a centralized architecture, enhancing the impact of the pruning rule 3. 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. At the end of the task, all the frequent closed found are sent to the reducers. 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), the proposed PaMPa-HD approach is able to apply pruning rule 3 and handle high-dimensional datasets, otherwise not manageable due to memory issues.

Job 2 Pseudo code

```
procedure MAPPER(Frequent Closed itemset;
        Transposed table)
    if Input $I$ is a table then
        $itemset \leftarrow ExtractItemset(I)$
        $tidlist \leftarrow ExtractTidlist(I)$
        $Output(<itemset; tidlist + table\ I\ rows>)$
    else (i.e. input $I$ is a frequent closed Itemset)
        $itemset \leftarrow ExtractItemset(I)$
        $tidlist \leftarrow ExtractTidlist(I)$
        $support \leftarrow ExtractSupport(I)$
        $Output(<itemset; tidlist + support>)$
    end if
end procedure
procedure REDUCER($key = itemset$;
        $value = itemsets\ \&\ tables\ T[\ ]$)
    $oldest \leftarrow null$
    for each itemset or table $T$ of $T[\ ]$ do
        $tidlist \leftarrow ExtractTidlist(T)$
        if tidlist previous of oldest in a Depth-First Search then
            $oldest \leftarrow T$
        end if
    end for
    $Output(<itemset + oldest>)$
end procedure
```

Job 3 Pseudo code

```
procedure MAPPER(Frequent Closed itemset;
        Transposed table)
    if Input $I$ is a frequent closed itemset then
        save $I$ to local memory
    else (i.e. input $I$ is a Transposed Table)
        while  $max\_exp$ is not reached do
            Run $Carpenter(minsup; TT|_X)$
        end while
            $Output(<itemset;\ tidlist + table\ I\ rows>)$
    end if
    for each frequent closed itemset found do
        $Output(<itemset; tidlist + support>)$
    end for
end procedure
procedure REDUCER($key = itemset$;
        $value = itemsets\ \&\ tables\ T[\,]$)
    $oldest \leftarrow null$
    for each itemset or table $T$ of $T[\,]$ do
        $tidlist \leftarrow ExtractTidlist(T)$
        if  $tidlist$ previous of $oldest$ in a Depth-First Search then
            $oldest \leftarrow T$
        end if
    end for
    $Output(<itemset + oldest>)$
end procedure
```

5. Experiments

Next, we perform a set of experiment to evaluate the performance of the proposed algorithm. Firstly, we measure the performance impact of the maximum expansion threshold, evaluating the quality of a set of proposed strategies. After that, we measure the efficiency of the proposed algorithm, comparing it with the state of the art distributed approaches (Section 5.3). Subsequently, we focus on more technical aspects of our approach. Specifically, we experimentally analyze the impact of the number of transactions of the input dataset on the performance of PaMPa-HD (Section 5.4). Then, we measure the performance impact with respect to the number of parallel tasks (see Section 5.5). Finally, we analyze in Section 5.6 the communication costs and load balancing behavior, which are very important in such a distributed context.

We perform the experiments on two real life datasets. The first real dataset is the **PEMS-SF** dataset [9], 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 [10]). Each transaction represents the daily traffic rate of 963 lanes, sampled every 10 minutes. It is characterized of 440 rows and 138672 attributes (6 x 24 x 963), and it has been discretized in equi-width bins of size 0.001. Because of the nature of PaMPa-HD, which is designed to cope with high-dimensional datasets characterized by a small number of transactions, we have used several down-sampled version (in terms of number of rows) of the dataset to measure the impact of the number of transactions on the performance of the algorithm.

The second real dataset is the **Kent Ridge Breast Cancer** [11], 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, floating point). Data have been discretized with an equal depth partitioning using 20 buckets (similarly to [3]).

The discretized version of the real dataset and the synthetic dataset generator are publicly available at http://dbdmg.polito.it/PaMPa-HD/. **TO DO: aggiungere versione finale discretized pemsf**

Table 1: Datasets

Dataset	Number of transactions	Number of different items	Average 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. 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.

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 in this phase to globally apply pruning rule 3 and reduce the search space. Low values of max_exp threshold increases 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 set of experiments. In the first one we perform the mining on the PEMS-SF (100 transactions) dataset with a minsup 10, by varying max_exp from 100 to 100,000,000. The minsup value has been empirically selected in order to let the mining problem being deep enough to show different performance. In Figure 7 are shown the results in terms of execution time and number of iterations (i.e., the number of jobs)[1]. It is clear how the max_exp parameter can

[1] Please note that in all the experiments, for sake of clarity, the confidence intervals (obtained after a sufficient number of executions and with complementary level of significance

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 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 2 reports the minimum and the maximum execution time of the mining tasks executed in parallel for both the 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

of 95%) are omitted from the graphs.

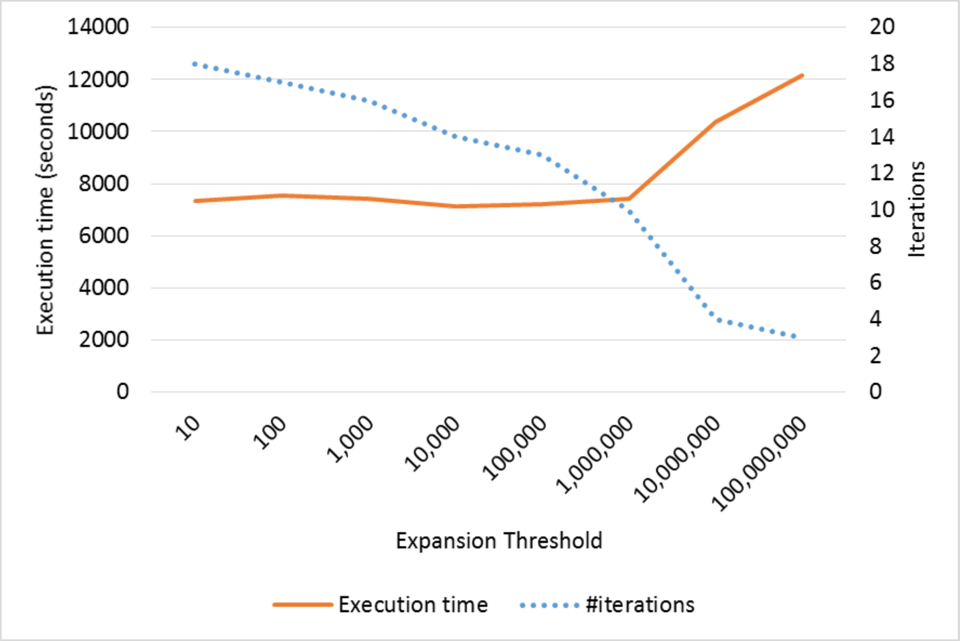

Figure 7: Execution time and number of iterations for different *max_exp* values on PEMS-SF dataset with *minsup*=10.

the algorithm. However, as demonstrated by the curves in Figures 7 and 8, it is very dependent on the use case and distribution of the data. In the next subsection we introduce and motivate some tuning strategies related to *max_exp*.

5.2. Proposed strategies

This section introduces some heuristic strategies related to the *max_exp* parameter. The aim of this experiment is to identify an heuristic technique which is able to deliver good performances without the need by the user to tune up the the *max_exp* parameter. Before the introduction of the techniques, let us motivate the reasons behind their design. Because of the

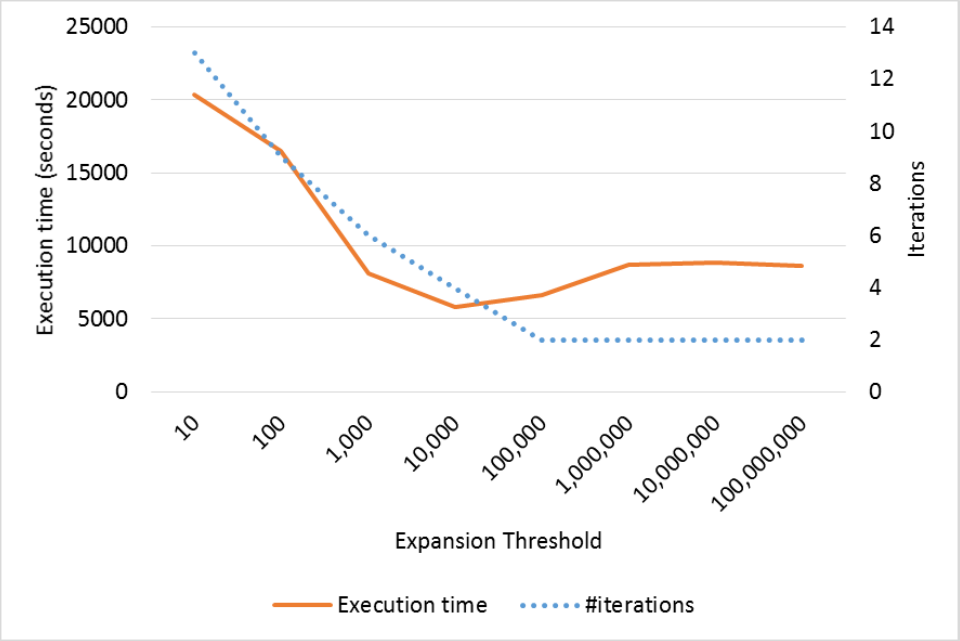

Figure 8: Execution time and number of iterations for different *max_exp* values on Breast Cancer dataset with *minsup*=5.

enumeration tree architecture, 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 projected on a tid. For these reasons, the tables of the initial mining phase are the most heavy to be processed. On the other hand, the number of nodes to process 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

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

are not more tids in the node transposed table). These reasons motivated us to introduce some strategies that assume a maximum expansion threshold that is increased with the number of iterations. These strategies start with very low values in very initial iterations (i.e. when the nodes are more heavy to be processed) and increase max_exp during the mining phases.

The strategy #1 is the most simple: the max_exp is increased with a factor of X at each iteration. For instance, if the 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 have tried to leverage informations about the execution time of each iteration and the pruning effect (i.e. the percentage of transposed tables / nodes that are pruned in the synchronization job). Specifically, strategy #2 consists in increasing, at each iteration, the max_exp parameter with a factor of $X^{T_{old}/T_{new}}$, given T_{new} and T_{old} the execution time of the previous two jobs. The motivation is to balance the growth of the parameter in order to achieve a stable execution times among the iterations. For strategy #3, we take into account the relative number of pruned tables. Indeed, this value cannot be easily interpreted. An increasing pruning percentage means that there are a lot of tables that are generated uselessly. However, an increasing trend 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. Finally, strategy #4 is inspired by the congestion control of TCP/IP (a data transmission protocol used by many Internet applications [12]). Precisely, the *max_exp* is handled like the congestion window size (i.e. the number of packets that are sent without congestion issues). This strategy, called "Slow Start", assumes two types of growing of the window size: 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 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 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 (for this reason $X \geq 10$). 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 has to build 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

Table 3: Strategies

Strategy #1(X)	Increasing at each iteration with a factor of X
Strategy #2(X)	Increasing at each iteration with a factor of $X^{T_{old}/T_{new}}$
Strategy #3(X)	Increasing at each iteration with a factor of $X^{Pr_{old}/Pr_{new}}$
Strategy #4	Slow start, with a fast increase factor of X

times of the last iterations which are often shorter than the first ones. **Fabio & Paolo:Non siamo sicuri che convenga inserire questa parte sul time out. Michiardi: I guess it is ok: mechanisms like speculative execution work similarly, hence to me the approach is not shocking**. The increasing *max_exp* value introduced by the described strategies, however, leads to a degradation of the load balancing between the parallel tasks of the job. To limit this issue, we have introduced a timeout of 1 hour. After that, all the tasks will be forced to run the synchronization job. From the algorithmic point of view, this is not a loss, since the the tables are expanded in a depth-first fashion. The last tables, hence, are the ones with the highest probability to be pruned. Although, in this way, we are limiting to 1 hour the amount of time in which we are not completely exploiting the resources of the commodity cluster (i.e. only few very long tasks running). A value of 1 hour has been empirically proved to be a good trade-of between load balancing and a good leveraging of the centralized memory pruning.

Table 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. In Table 4 are resumed 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 10,000. The max value has been increased in the cases in which the performance suggest a decreasing execution time trend.

Since the best performance is achieved with values of 10 and 100,000 respectively for PEMS-SF and Breast Cancer datasets (Figures 9 and 10), we will use this configuration for the experiments comparing PaMPa-HD with other distributed approaches. **Fabio: queste figure sono indispensabili?**

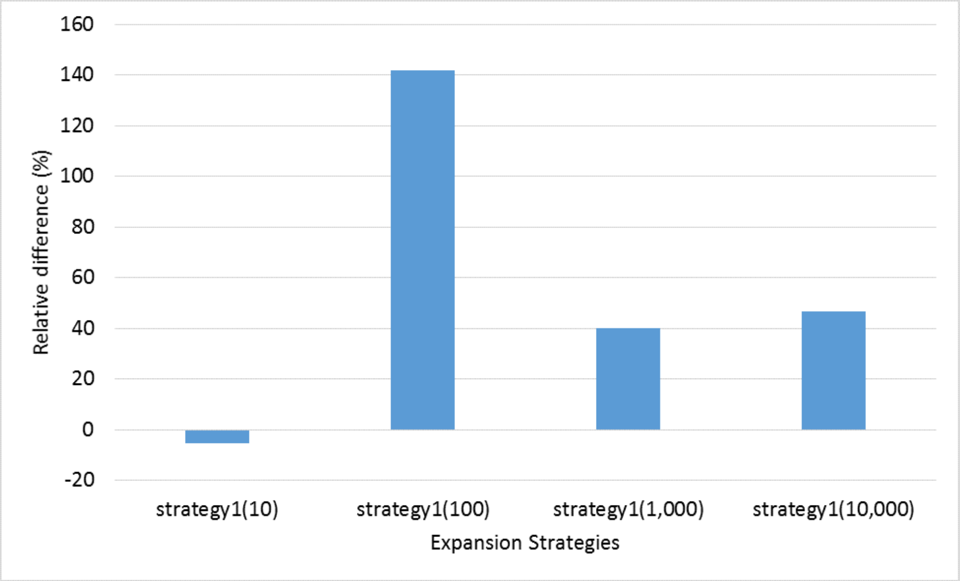

Figure 9: Relative gains on Pems-SF dataset with *minsup*=10, Strategy1 and different X values.

Michiardi: in my opinion either: i) we omit them and only report # in the text ii) we put a table. Fabio: Se approvate, le eliminerei e specifico la percentuale vincente nel testo, visto che la figura 9 pessima e con quel picco cosi' alto potrebbe suscitare domande scomode. In questo modo si elimina anche il dubbio se invertire gli esperimenti coi due dataset come suggeriva Paolo The difference may be caused by the characteristics of the dataset: evidently, PEMS-SF dataset benefits of more synchronization phases.

5.3. Running time

After the identification of a good trade-of strategy in the previous section, we have used it to analyze the efficiency of PaMPa-HD comparing it with three distributed state-of-the-art frequent itemset mining algorithms:

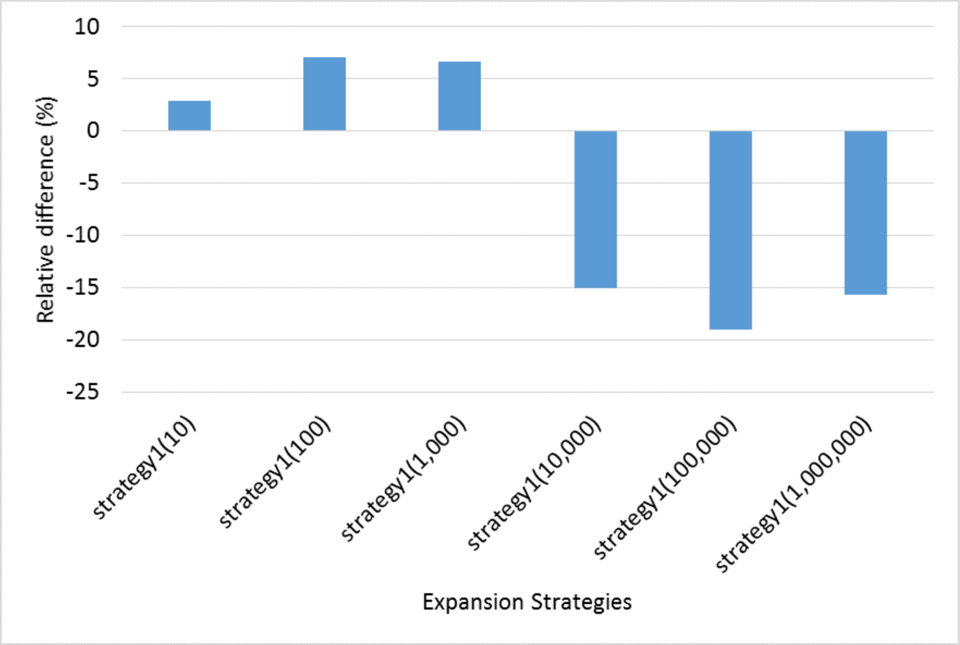

Figure 10: Relative gains on Breast Cancer dataset with *minsup*=5, Strategy1 and different X values.

1. Parallel FP-growth [13] available in Mahout 0.9 [14], based on FP-growth algorithm [8]
2. DistEclat [15], based on Eclat algorithm [16]
3. BigFIM [15], inspired from Apriori [7] and DistEclat

This set of algorithms represents the most cited implementation 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. For details about the algorithms, see Section 6.

The first set of experiments has been performed with the 100-rows version PEMS-SF dataset [9] and minsup values 35 to 5.[2]

As shown in Figure 11, 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, 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 [11]. As reported in Figure 12 (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 either 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 it is always slower than PaMPA-HD (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 [15], when possible (i.e. when there were enough 3-itemsets to execute also the second phase of the mining).

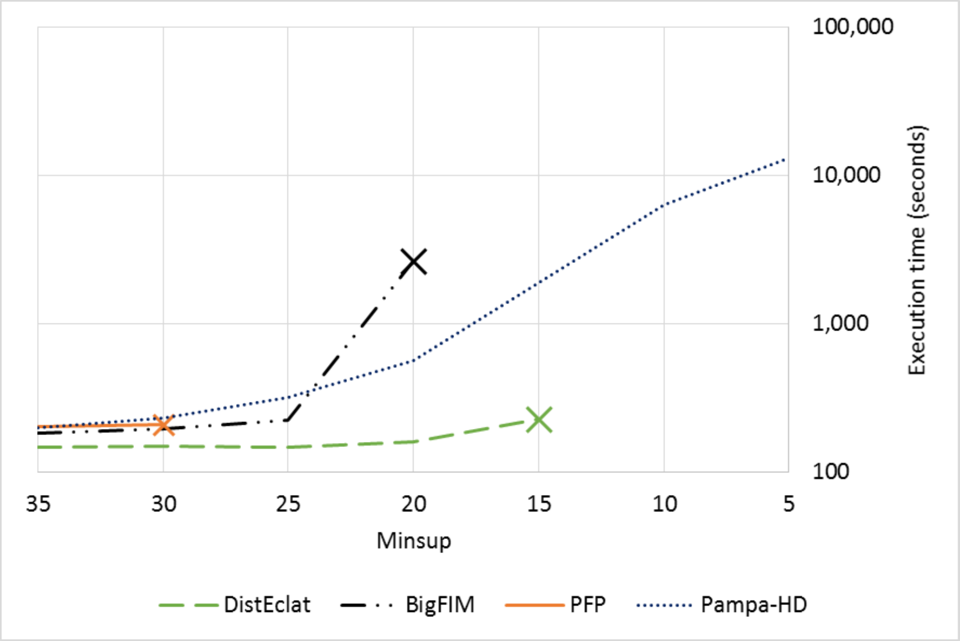

Figure 11: Execution time for different Minsup values on the PEMS-SF dataset (100-rows).

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 algorithm, the next experimental subsections will experimentally describe the behavior of PaMPa-HD with respect to the number of transactions, number of independent tasks, communication costs and load balancing.

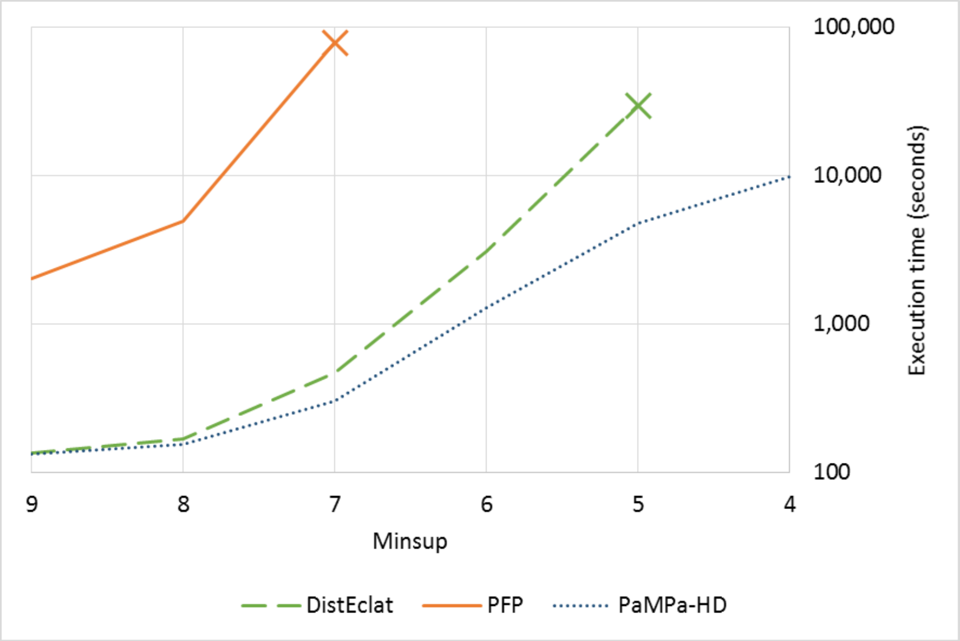

Figure 12: Execution time for different Minsup values on the Breast Cancer dataset.

5.4. Impact of the number of transactions

This set of experiments measures the impact of the number of transactions on PaMPa-HD performances. At this aim, it will be used the PEMS-SF datasets 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 (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 of the 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 13, 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.

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 number of nodes

The impact of the number of independent tasks involved in the algorithm execution is not trivial issue. Adding a task to the computation would not only delivers more resources such as memory or CPU. An additional task leads to split the chunk of the enumeration tree that is explored by each task. On 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 14 and Figure 15, it is reported the behavior of PaMPa-HD with a mining

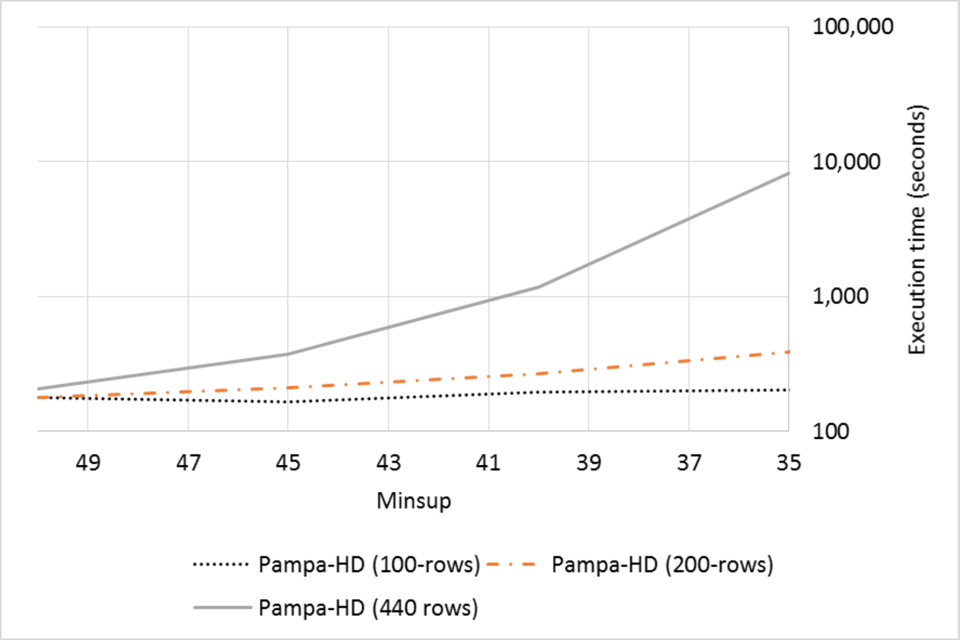

Figure 13: Execution times for different version of the PEMS-SF for PaMPa-HD.

process on the datasets PEMS-SF and Breast Cancer. The minsup values, respectively of 20 and 6, have been chosen in order to be deep enough to show performance differences among the different degree of parallelism. 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 15, shows a stronger performance gain. **(Lo stesso esperimento per quanto riguarda PEMS l'ho fatto con un minsup pi basso e la linea era ancora pi orizzontale** The behavior is related to the dataset data distribution. For PEMS-SF dataset, the advantages related to additional independent nodes into the mining is mitigated by the loss from the point of view of state local pruning phase inside the nodes.

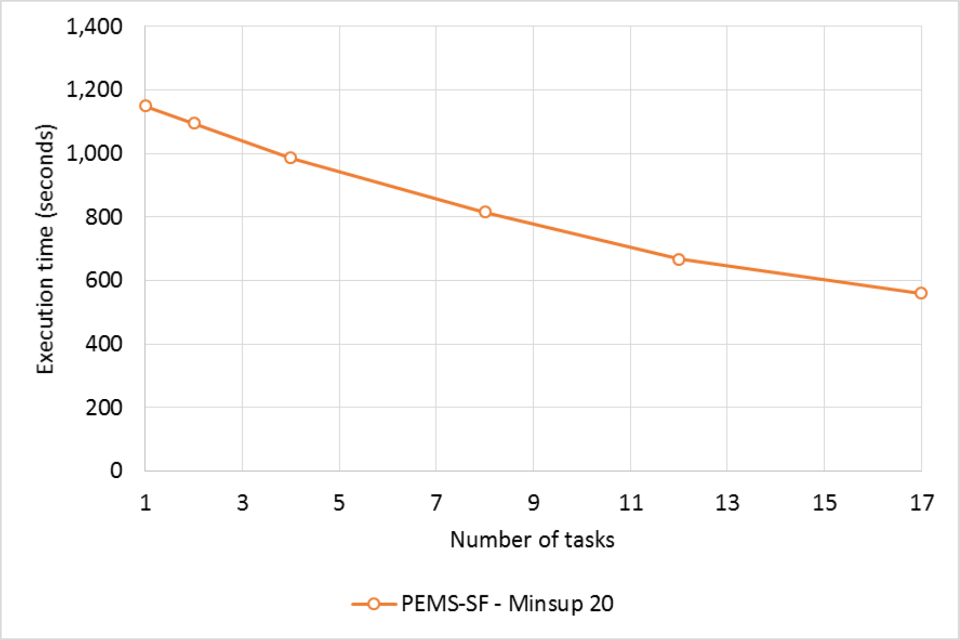

Figure 14: Execution times for PEMS-SF datasets with different number of parallel tasks.

With additional nodes, in fact, 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 environment. In problems characterized by any kind of ”state” benefit (in this case local pruning inside the tasks), a higher degree of parallelism does not lead to better performance a priori. **(Spero che si capisca, Michiardi mi aveva chiesto di riformulare)**

5.6. Load Balancing and communication costs

The last analysis are related to the load balancing and the communication costs of the algorithm. These issues are very underestimated but they represent very important factor in such a distributed environment. Com-

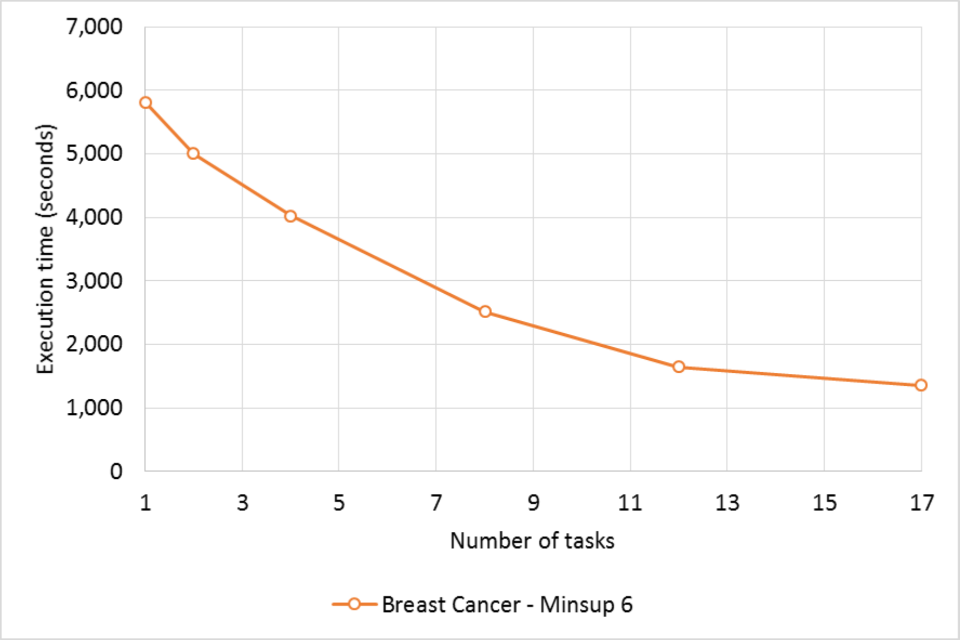

Figure 15: Execution times for Breast Cancer datasets with different number of parallel tasks.

munication costs represent are among the main bottlenecks related to the performance of parallel algorithms [17]. A bad-balanced load among the independent tasks leads to few long tasks that block the whole job.

PaMPa-HD, being based on Carpenter algorithm, as shown in the previous sections, 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 (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 tasks. 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, burdens of the I/O costs. In fact, in order to prune some useless tables and improve the performances, 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 the *max_exp* (Figure 7 and Figure 8), in some cases additional synchronization phases leads to better execution times, despite their related overhead. In Figure 16 and Figure 17 it is shown the communication cost during a mining process. 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 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 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 mapper phase of the Job 3. This job is iterated until the mining is complete and it is the one more involved by the increasing of increasing of

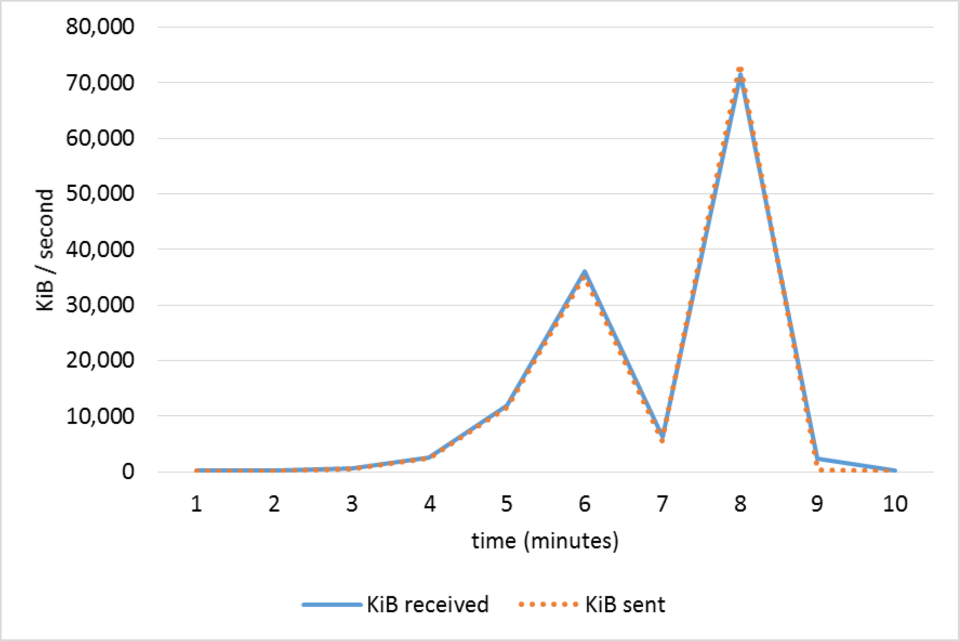

Figure 16: Received and sent data in the commodity cluster network during PEMS-SF dataset mining, minsup=20.

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, as shown by 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.

6. Related work

Questo tutto vecchio, rifrasare? Frequent itemset mining represents a very popular data mining technique used for exploratory analysis. Its

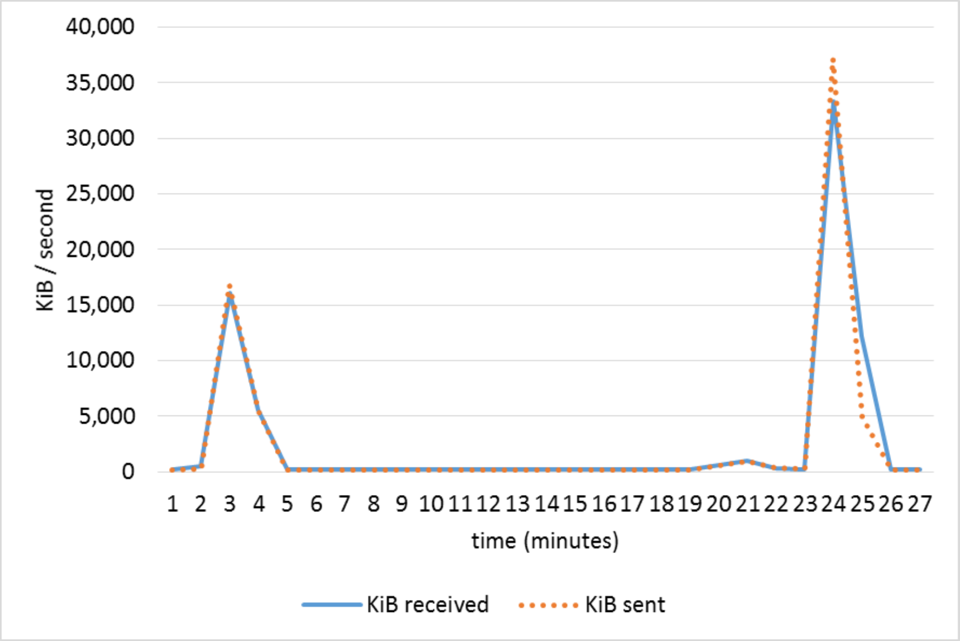

Figure 17: Received and sent data in the commodity cluster network during Breast Cancer dataset mining, minsup=6.

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 [7] is a bottom up approach: itemsets are extended one item at a time and their frequency is tested against the dataset. FP-growth [8], 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 [7, 16].

In recent years, the explosion of the so called Big Data phenomenon has

Table 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

pushed the implementation of these techniques in distributed environments such as Apache Hadoop [1], based on the MapReduce paradigm [18], and Apache Spark [2]. Parallel FP-growth [13] 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 [19]. This version extracts all the frequent itemsets and not just the closed ones. BigFIM and DistEclat [15] are two recent methods to extract frequent itemsets. DistEclat represents a distributed implementation of the Eclat algorithm [16] 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 Apriori and Eclat paradigms. BigFIM and DistEclat are divided in two phases. In the first one, the approaches use respectively an Apriori-like an 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. The last phase of the mining, both the approaches uses Eclat to extract all the closed itemsets. In addition, [20] 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.
While the previous works have been designed for use cases characterized by datasets with a large amount of transactions, Carpenter algorithm [3], 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. A detailed introduction to the algorithm is presented in section 3. This work extends our previous work [21]. The original algorithm assumes a slightly different architecture, assuming an additional independet 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.

7. Applications

Questo tutto vecchio, rifrasare? Lo eliminiamo? Michiardi: solo se abbiamo problemi di spazio Since PaMPa-HD is able to process extremely high-dimensional datasets we believe it is suitable for many application (scientific) domains. The first example is bioinformatics: researchers in this environment 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 environments are two important application 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 [22], [23]. Datasets, in which the transactions represent flows and the item are flows attributes, are already a very promising application domain for data mining techniques [24],[25], [26].

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. Experimental results shos its scalability and its performance in coping with real datasets characterized by up to 8 millions different items and, above all, an average number of items per transaction over a hundred thousands, on a small commodity cluster of 5 nodes. PaMPa-HD outperforms state-of-the-art algorithms, by showing a better scalability than all the state of the art distributed approaches such as PFP, DistEclat and BigFIM. Further developments of the framework can be related to the introduction of new pruning rules related to specific use cases inside the algorithm. This pruning, so far related to the post processing phase, would avoid the processing of useless data.

Acknowledgement

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

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

[2] 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.

[3] 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.
URL `http://doi.acm.org/10.1145/956750.956832`

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

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

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

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

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

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

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

[11] 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, 15th 2015

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

[13] 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.

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

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

[16] 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.

[17] 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.
URL http://dx.doi.org/10.14778/2535570.2488334

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

[19] 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 spark (May 2016). arXiv:1505.06807.
URL http://arxiv.org/abs/1505.06807

[20] 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.

[21] 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.
URL `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7395755`

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

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

[24] 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.
URL `http://dx.doi.org/10.1109/TrustCom.2013.153`

[25] 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.

[26] 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.

URL http://dx.doi.org/10.1016/j.comnet.2008.12.011