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,

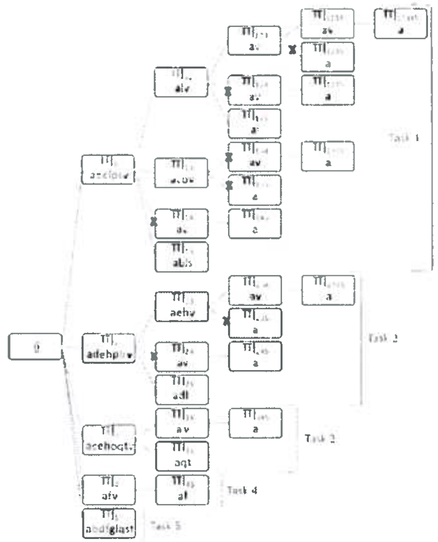

Figure 3: Running toy example: each node expands a branch of the tree independently. Pruning rule 1 and 2 are not applied. The pruning rule 3 is applied only within the same task: the red crosses on the edges represent pruned nodes due to local pruning rule 3, e.g. the one on node {2 4} represents the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

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.

SONO ARRIVATA QUI

lo eliminerei

, a parallel Map-Reduce algorithm of Carpenter [ ],

itemset mining

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~~ is 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

PaMPa-HD exploits the pruning rules 1 and 2 and a slight variation of the pruning rules 3 discussed ~~presented~~ in Section 3. Furthermore, it has been designed to achieve a good load balancing and robusteness to memory-issues - Aggiungere qui una frase x indicare come sono stati indirizzati questi 2 aspetti.

"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 ae 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 ae 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~~ (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 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_{ae} can be:

- a frequent closed itemset ae extracted during the subtree exploration of the node $TT_{3,4}$.
- a transposed table associated to the itemset ae 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_a, it is kept the node $TT_{1,2,3}$ (see Figure 5)

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 ae 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

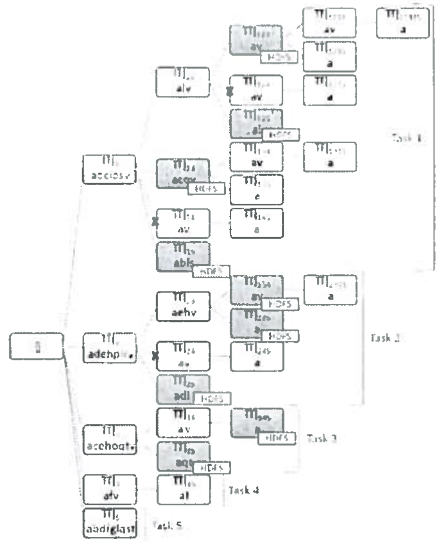

Figure 4: Execution of PaMPa-HD on the running example dataset. For sake of clarity, pruning rules 1 and 2 are not applied. The dark nodes represent the node that have been written to hdfs in order to apply the synchronization job local pruning, e.g.,

node associated with the transaction ids combination $\{1, 2, 3\}$. The use of this approach allows the parallel execution of the mining process, providing at the same time a very high reliability dealing with heavy enumeration trees, which can be split and pruned according to pruning rule 3.

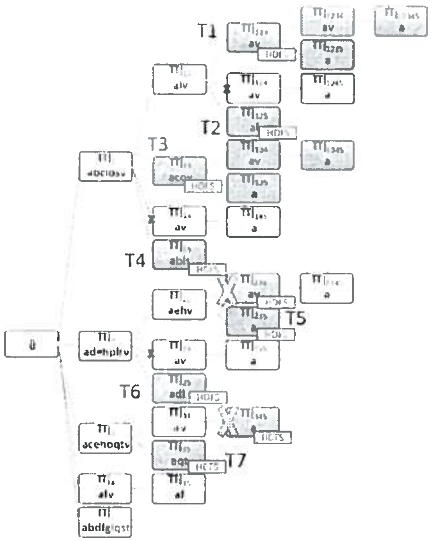

Figure 5: Execution of PaMPa-HD on the running example dataset. For sake of clarity, pruning rules 1 and 2 are not applied. The big checked crosses on nodes represent the nodes which have been removed by the synchronization job, e.g., the one on node {2 3 4} represents the pruning of node {2 3 4}.

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).

Aggiungere le label Algorithm 1.

PaMPa-HD pseudo code

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**

distribution of the input dataset

The first job, as whose pseudo-code is reported in Algorithm 2, is ~~developed to~~ distributes the input dataset to the indepen-

The second job performs the synchronization of the partial results and exploits the pruning rules. At the end, ~~while~~ the last ~~job~~ ~~addresses~~ interleaves the carpenter execution with the synchronization phase.

~~In the first job (Algorithm 2)~~

and partial

dent 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 1.1~~ ~~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 $(t_{i+1}, t_{i+2}, \ldots, t_n)$ greater than the considered tid t_i. (lines 2-7 in Algorithm 2)

Andare a capo Scrivere il nome del job in bold e poi seguire con la descrizione

Title job 1 (Algorithm 2) ... descrizione

- 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 XX-YY in Algorithm 2) 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

(see Section 3 for further details)

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.

Questa sezione mi sembra molto lunga. Consiglio di spezzarla in 3 paragrafi. Come inizio di ogni paragrafo scrivere in bold il nome del job (vedi proposta label x i diversi algoritmi.

Title job 1 (in bold) ______

Title job 2 (in bold) ______

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| ≥ 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
```

Algorithm 2: Distribution of the input dataset (job 1) and local and partial execution of the Carpenter algorithm

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.

bold

Title job 2. (Algorithm 3) ~~Algorithm 3~~

~~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 oldest version in a depth first exploration. For instance, referring to our running example in Figure 5, in the bucket of the itemset *ae* are collected the entries related to the nodes T_{124} 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.

Title job 3 (Algorithm 4)

This is

~~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

~~Algorithm 4~~

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.

Inserirei alcuni riferimenti alle linee dello pseudocodice riportato tipo (lines XX-YY in Algorithm Z)

Job 2 Pseudo code

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

Algorithm 3: ~~Round~~ Synchronization phase and exploitation of the pruning rules (job 2)

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 ← null
    for each itemset or table T of T[ ] do
        tidlist ← ExtractTidlist(T)
        if tidlist previous of oldest in a Depth-First Search then
            oldest ← T
        end if
    end for
    Output(<itemset + oldest>)
end procedure
```


Algorithm 4: ~~A~~ Interleaving of the Carpenter execution and the synchronization phase (job 3)

SONO ARRIVATA QUI