Ecole Polytechnique de Louvain

LINGI2364: Mining Patterns in Data

Project 1: Implementing Apriori

Group 11

Alessandra Rossaro (01211800), Matteo Salvadore (01731800)

Version 1.0 - 21/10/2018

AY 2018-2019

1 Implementations

1.1 Apriori

To implement the Apriori algorithm we have created ad hoc implementation of an HashTree that is used to contain the internal structure of the Apriori search. Every node of the HashTree is characterized with an HashMap containig its children nodes and the frequency of the node. We have chosen this data structure in order to simplify the accesses of the algorithm to the elements of the SearchTree since the complexity of the access is $O(1)$.
Since the Apriori algorithm, in order to expand a new level, needs the previous one, during the Database reading process, we have saved in a TreeMap where the keys are all the patterns with a singular element and the values are the corresponding supports, computed incrementally during the reading process.
This is the only improvement that we have performed on the *DataSet* class that we have renamed into *AprioriDataSet*. For the next levels the search is performed accordin to the Apriori algorithm specifications.

We have then implemented the following optimizations:

- We generate candidates according to the *Merging itemsets technique*: when we expand nodes to generate new candidates we take the nodes with same father of the node that is being expanded and we select as candidates only those have keys with value higher w.r.t. the current node's key.
- After the nodes generation we prune the infrequent candidates by computing their frequency. Those nodes are not considered for the new levels' expansion.
- Our HashTree data structure is implemented as a *Prefix Tree* to make the merging more efficient, the order of the items in the tree is the natural Integer order.

1.2 ECLAT

To implement the ECLAT algorithm we have redefined the *DataSet* class into *ECLATDataSet* class, since this algorithm needs to read database and to convert it into its *Vertical Representation*. This new class loads the entire database in memory and then it creates the *Vertical Representation* using a TreeMap that contains as key an Integer that represents a singular element and as value an HashSet of Integers that represent the numbers of transactions in which the singular element appears. We have decided to use this kind of data structure since just using an array to store the transaction numbers was inefficient. With our improvement the access complexity to insert an item in an HashSet is $O(1)$ and to access to the TreeMap is still $O(1)$.
This kind of data structure is used whenever the algorithm projects the Database on a new item.
We have implemented the ECLAT algorithm following the specifications of *Depth-First Search* algorithms explained in the slides of the course.

We have implemented the following optimizations:

- As explained before, we have used an HashSet in our data structure to store the list of transaction numbers in which the singular item appears. Our decision was the result of a sequence of attempts to choose the most performant data structure: our first choice was an ArrayList that we have discarded because the execution time spent to transform the Database in the *Vertical Representation* was too high, the same same problem occurred with our second choice: a vector of Integer.

- Thanks to the HashSet we can perform the intersection more efficiently since, instead of scanning two lists, we can just take the shorter list and search for the singular items in the longer one exploiting the fact that checking if an element is present in a HashSet is $O(1)$, then the total complexity of an intersection is $O(l)$ where l is the length of the smaller HashSet. It is worth noting that comparing the shorter HashSet with the longer minimizes the number of comparisons to be performed.
- Before the computation of the intersections among a line of the *Vertical Representation* and an item, we skip the computation with the Hashset with a support lower than minimum support. This is equivalent to the perform *pruning* on the SearchTree.

2 Performances

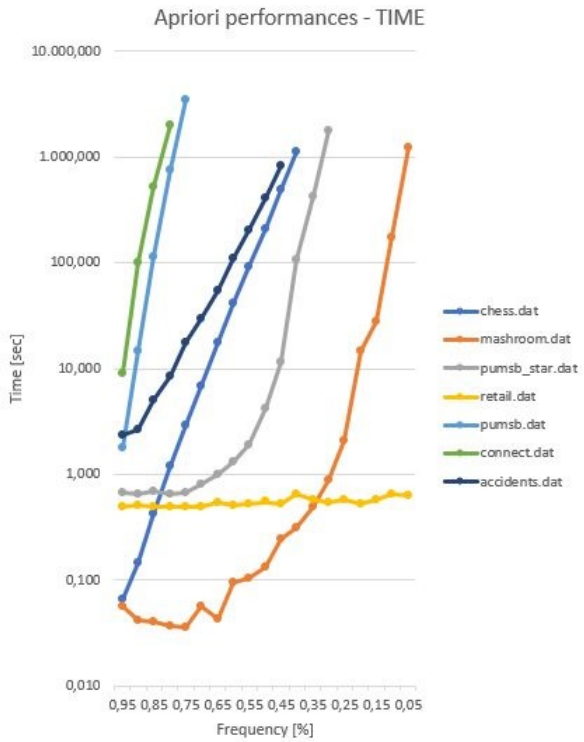
We have decide to use only the following datasets to compute the performances in terms of time and memory with different support values:

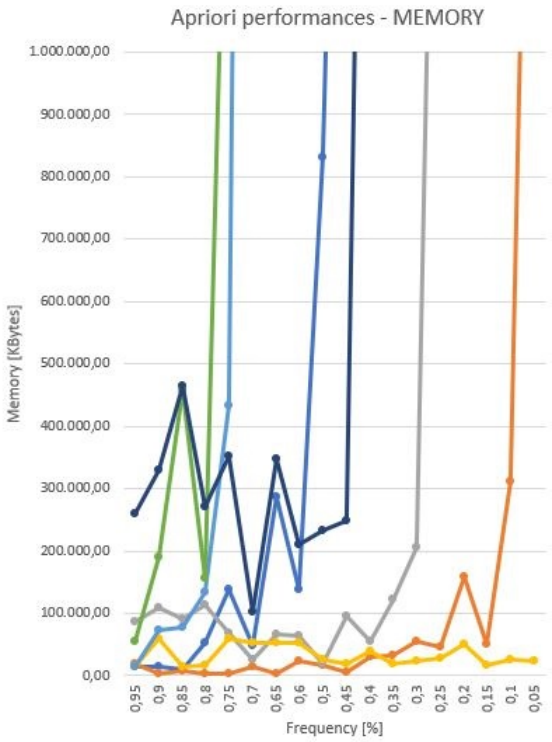
- *chess.dat*
- *mushroom.dat*
- *pumsb.dat*
- *pumsb_star.dat*
- *connect.dat*
- *retail.dat*
- *accidents.dat*

The datasets are ordered according to the ascending number of rows.
We have decided to discard *toy.dat* because of its dimension: the results of the performances computed were inconsistent w.r.t. the overhead (in terms of memory occupied and elapsed time) by the JVM.

The first computation of performances was with a range of values between 0.1 and 0.9, but we have noticed that already from the first dataset, for a frequency equal to 0.4, the running time was too high to be computed from our laptop.
For this reason we have decided to use a frequency that starts from 0.95 and to use a step of 0.05 and to run the performance script until the running time were too long; whenever we noticed that a data-set were taking too long for a specific frequency we've ignored it for lower frequencies.

2.1 Apriori





2.2 ECLAT