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

The implementations of our algorithms are performed in Java language.
The source code and all the resources used to perform this project are available in the following GitHub repository: `https://github.com/JustSalva/ProjectsOfMiningPatternsInData`.

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 *data-set* 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 represents 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 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 to check if an element is present in a HashSet has complexity equal to $O(1)$, and so 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 one 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 decided to use only the following datasets to compute the performances in terms of time and memory with different support values:

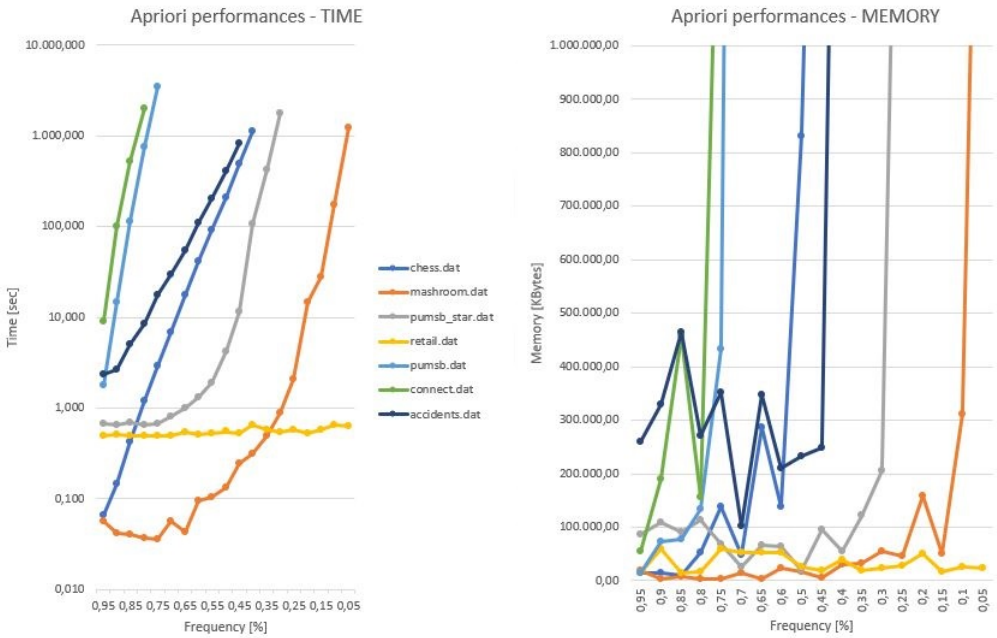
- *chess.dat*
- *mushroom.dat*
- *pumsb.dat*
- *retail.dat*
- *pumsb_star.dat*
- *connect.dat*
- *accidents.dat*

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 from 0.1 to 0.9, but we have noticed that, already from the first data-set and 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, to use a step of 0.05 and to run the performance script until the running time was too long; whenever we noticed that a data-set was taking too long for a specific frequency, we have ignored it for lower frequencies.

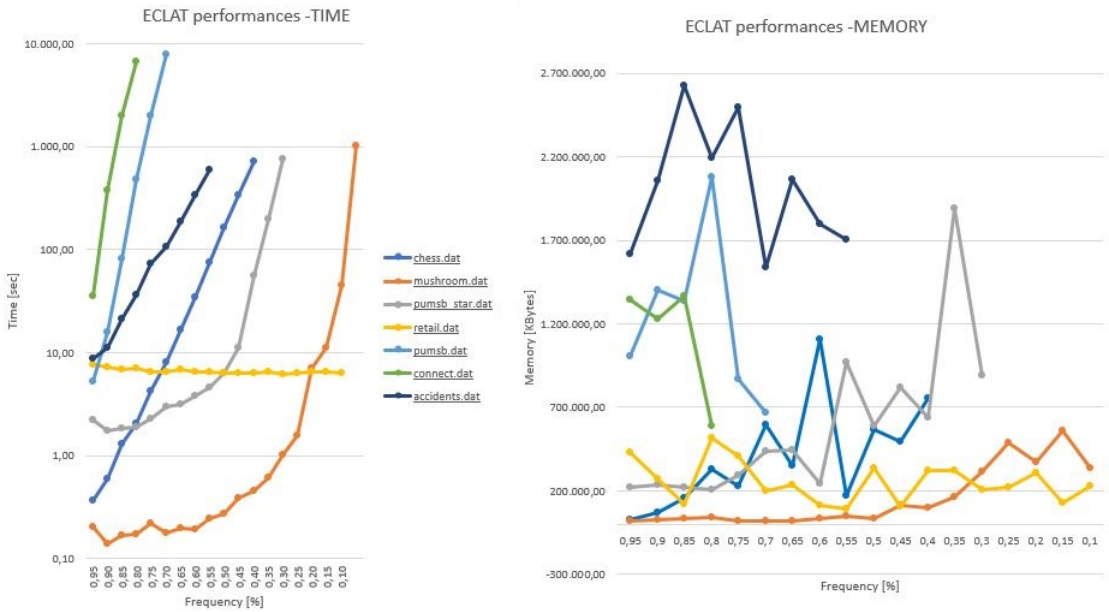
In the following sections, we have plotted the performances in terms of running time and used memory, w.r.t. different frequency values and for all data-sets. The running time graphs are plotted using the vertical axis, that represents the time, with a logarithmic scale.
We have encountered some difficulties in measuring the memory usage of our Java application since the values obtained were not too much coherent among different executions with the same parameters and even forcing the garbage collector to flush the memory was not so useful to tamper this phenomenon. We have then tried to average the results of consequent runs to keep the measurement performances coherent.

2.1 Apriori



As you can notice in the graph, the running time of our algorithm increases as the frequency decreases, for all data-sets, this kind of increase is exponential; for smaller data-sets this kind of phenomenon yields still a small growth (e.g. *retail.dat*), while for data-sets with very long transactions and very similar transactions (e.g. *connect.dat*) the running time increases more rapidly.
We have expected that *accidents.dat* database would be way more time requiring w.r.t.the other data-sets, since it has the highest number of transactions but we have observed that the length of the transactions and the similarities among them had more impact on the running time.
We have noticed that the memory usage grows, more or less, as the frequency decrease and explodes at the same frequencies at which the running time exponential function begins to assume a very high slope (when it starts tending to infinite).

2.2 ECLAT



As you can notice in the graph, the behavior we have obtained with our implementation of ECLAT algorithm is very similar to the Apriori performances, but the exponentials have a slightly smaller growth rate, and so their growth becomes very fast with slightly lower thresholds of frequencies. An exception of this observation is that the data-set *accidents.dat*, that has a way higher number of transactions, probably has an higher impact on the performances due to the *Vertical Representation* usage in this algorithm, and so our ECLAT algorithm performances are worse w.r.t. the Apriori one for this specific data-set. We have expected that the memory usage would have followed more or less the same behavior, but even with multiple measurements the behavior obtained was fuzzy; in particular for all the data-sets, at exception of *connect.dat*, *chess.dat* and *accidents.dat*, we can make the same observations we have done with our Apriori algorithm, but we cannot explain the behavior of these three databases.

2.3 Our system specification

All our measurements have been tested on a laptop with the following specifications:

- Processor: Intel Core i5-6198DU CPU @ 2.30GHz x 4
- RAM: 12 GiB - DDR4
- OS: Ubuntu 18.04.1 LTS
- JRE: Java 8u191

3 Notes

We have added into the last committed jar file on the *INGinious* platform, an implementation of FP-growth algorithm, unfortunately for various reasons, we could not complete the debugging phase, so it is still a work in progress. We will complete it afterwards, even knowing that it will not be evaluated. Inside our zip folder your find, together with this report, the *.dat*, *.mod*, *.run* file of our main model (BaseballProblem), a folder containing the matlab scripts used to plot the results, and two other folders (*threeFeetRule* and *incrementedRealism*) containing the modified models for section 5.1 and 5.2 of this document