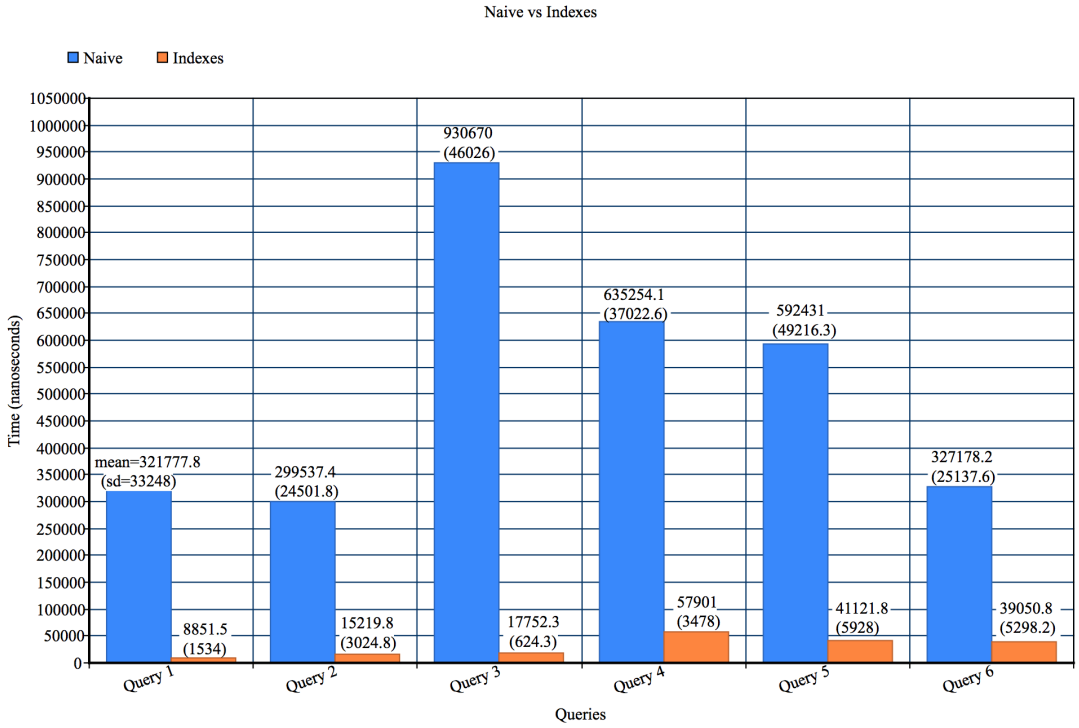
Big Data Analytics Assignment 3

Grigoris Konstantinis, r0488834

April 16, 2018

1 Index Structures

In the following image you can find the time of Naive (sequential) execution of the queries vs the execution of the queries using index structures. For the standard deviation I used the formula for a sample. Both mean time and standard deviation were rounded to 1 decimal digit. The mean time in nanoseconds is on top of the bar and the standard deviation inside the parenthesis.



The first query is equivalent one BitMap.select operation and is 35 times faster than the naive query. The second query which is equivalent to one BitMap.select operation and one BitSet count operation is 20 times faster. This makes sense since for both naive queries the complexity is $O(N)$ while for the 1st indexed query the complexity is $O(1)$ and for the second $O\left(1 + \left\lceil \frac{n}{64} \right\rceil\right) = O\left(\left\lceil \frac{n}{64} \right\rceil\right)$ (BitSet.cardinality() is $O\left(\left\lceil \frac{n}{64} \right\rceil\right)$). The third query which is equivalent to one BitSlice sum operation, is more than 50 times faster than the sequential query. The fourth query required a BitSlice.range and a BitMap.count operation. The result suggests that the BitSlice.range which is $O(encodingLength)$ is much slower compared to the BitMap.select operation which is $O(1)$. Finally the queries 5 and 6 despite requiring cloning operations of the BitSets are still very fast. For the query 6 it should be noted that despite being more complex, the naive implementation can check both conditions in the same loop and that makes it approximately as slow as query 1 and query 2, the complexity did not affect its run time much. On the other hand the indexed query, require more time for the more complex queries compared to the simpler ones (query 1 & 2), that makes sense as more operations are needed. For more complex queries the difference between naive queries and index queries tend be smaller, though still very large differences continue to exist (8 times faster).

2 Movie Recommendations

2.1 Implementation details

Two different formulas for computing Pearson's correlation were used. The first was the following

$$s_{xy} = \frac{\sum(x_i - E[X]) \cdot (y_i - E[Y])}{(n-1)s(X)s(Y)}$$

To speed it up, the means $E[X]$ were precomputed once, using all x_i resulting in a O(n) algorithm.

A second version of the Pearson correlation was also used where $E[X]$ and $E[Y]$ were only computed based on the common movie ratings. Follows the proof for a $O(N)$ equivalent formula

$$s_{xy} = \frac{\sum(x_i - E[X]) \cdot (y_i - E[Y])}{(n-1)s(X)s(Y)} =$$

$$\frac{1}{(n-1)s(X)s(Y)} \cdot \Big[\sum x_i y_i - \sum x_i E[Y] - \sum y_i E[X] + \sum E[X]E[Y]\Big] =$$

$$\frac{1}{(n-1)s(X)s(Y)} \cdot \Big[\sum x_i y_i - nE[X]E[Y] - nE[Y]E[X] + nE[X]E[Y]\Big] =$$

$$\frac{1}{(n-1) \cdot \sqrt{\frac{1}{n-1}\sum(x_i - \overline{x})^2} \cdot \sqrt{\frac{1}{n-1}\sum(y_i - \overline{y})^2}} \cdot \Big[\sum x_i y_i - nE[X]E[Y]\Big] =$$

$$\frac{1}{\sqrt{\sum(x_i - \overline{x})^2} \cdot \sqrt{\sum(y_i - \overline{y})^2}} \cdot \left[\sum x_i y_i - \frac{\sum x_i \sum y_i}{n}\right] =$$

$$\frac{n\sum x_i y_i - \sum x_i \sum y_i}{n \cdot \sqrt{\sum x_i^2 - \sum 2xiE[X] + \sum E[X]^2} \cdot \sqrt{\sum y_i^2 - \sum 2yiE[X] + \sum E[Y]^2}} =$$

$$\frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - 2n^2E[X]^2 + n^2E[X]^2} \cdot \sqrt{n\sum y_i^2 - 2n^2E[Y]^2 + n^2E[Y]^2}} =$$

$$\frac{\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \cdot \sum x_i^2 - n^2E[X]^2} \cdot \sqrt{n \cdot \sum y_i^2 - n^2E[Y]^2}} =$$

$$\frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n \cdot \sum x_i^2 - (\sum x_i)^2} \cdot \sqrt{n \cdot \sum y_i^2 - (\sum y_i)^2}}$$

Taking advantage (1) of the fact the number of different movie IDs is a relatively small number, and (2) each user has rated a small number of movies (around 100), to efficiently find the common elements between the two lists instead of using hashing (i.e. a HashMap), a lookup table was used. Code snippet follows

```
    float[] lookUpArray;
    float lookUpArrayFlag;

    public double correlation(List<MovieRating> xRatings, List<MovieRating> yRatings) {

        //The smaller list should become the "Map"!
        if (xRatings.size() > yRatings.size()) {
            List<MovieRating> temp = xRatings;
            xRatings = yRatings;
            yRatings = temp;
        }

        //Ratings 0.5 to 5, definitely not flag value -1
        for (MovieRating rating : xRatings) {
            lookUpArray[rating.getMovieID()] = (float) rating.getRating();
        }

        for (MovieRating rating :yRatings){
            if(lookUpArray[rating.getMovieID()] != lookUpArrayFlag){
                //Common Element
            }
        }

        //Set arr to original state
        for (MovieRating rating : xRatings) {
            this.lookUpArray[rating.getMovieID()] = this.lookUpArrayFlag;
        }

    }
```

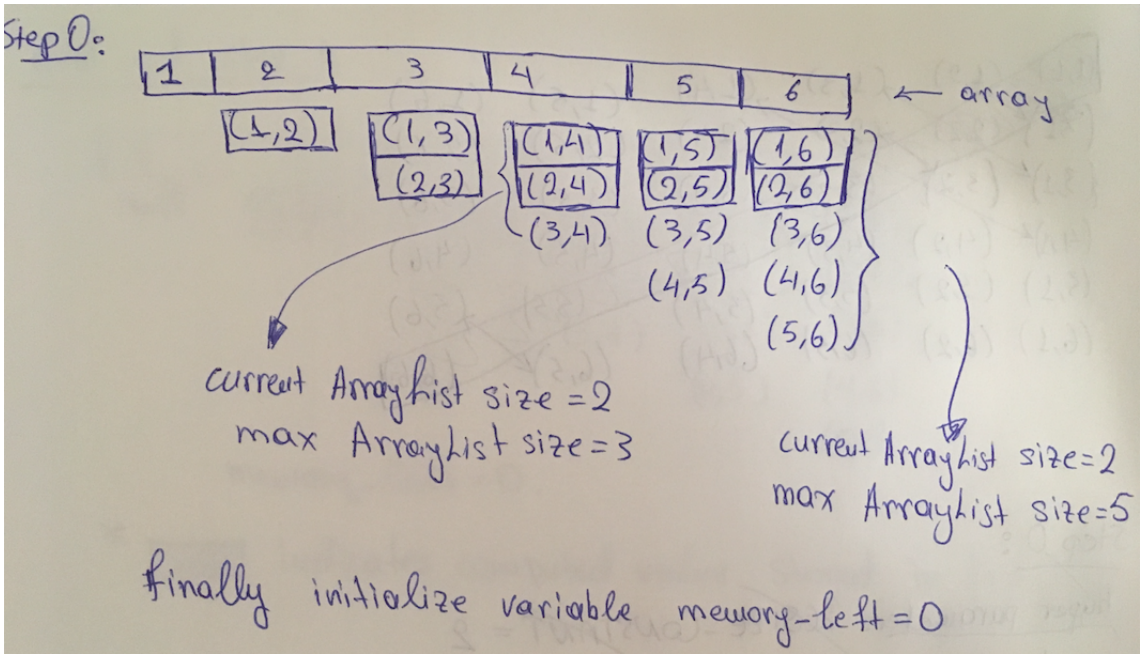
Instead of declaring HashMap and creating lots of Objects a lookUpArray is declared only once somewhere in the program with Flag values. Then every time the correlation method is called, first the array is populated only using the ratings of one list and then using the second list I check if the value is not null, that means a common element was found. Since the number of elements of each list is small, restoring the array to its original state with looping is faster than declaring a new array all the times. In reality though, in my code a different version is used, I make use of the fact cor(1,2), cor(1,3) ... cor(1,69000) will be computed so instead of setting and unsetting the lookUpArray for the user 1 all the time, it will only happen once. Coding snippet follows that explains it better.

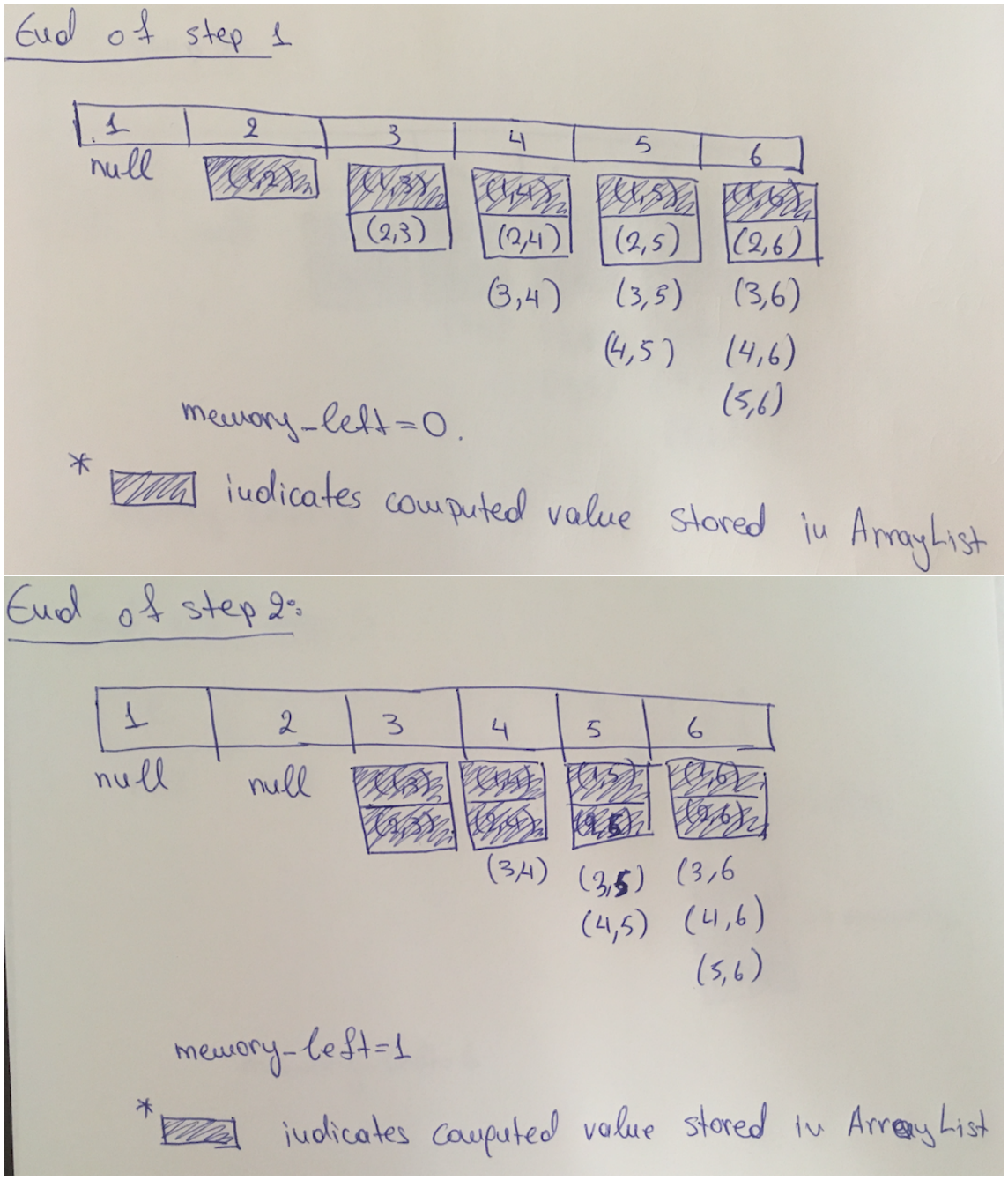
```
for (int i = 0; i < userIDs.size(); i++) {
        List<MovieRating> xRatings = usersToRatings.get(userIDs.get(i));
        for (MovieRating rating : xRatings) {
            lookUpArray[rating.getMovieID()] = (float) rating.getRating();
        }
        for (int j = i + 1; j < userIDs.size(); j++) {
            List<MovieRating> yRatings = usersToRatings.get(userIDs.get(j));
            double cor = correlation2(lookUpArray, yRatings);
        }
        for (MovieRating rating : xRatings) {
            lookUpArray[rating.getMovieID()] = lookUpArrayFlag;
        }
    }
```

One property of Pearson correlation is that $Cor(X, Y) = Cor(Y, X)$, this means I only need to store half of the correlations. The assignment specifies that we need up to 4 decimal digits precision, this means instead of using double I can use short and amplification of 10.000 in order to save memory. Even then the memory requirements are too high (almost 5 Gigabytes for 70.000 users). In order to overcome this problem and maintain an efficient algorithm I created a specialized data structure for this assignment that stores intermediate results in the computer's disk and loads them later when needed. As I do not have time to create graphics before the deadline, the algorithm will be explained with photos and an example of computing correlations of 6 users.

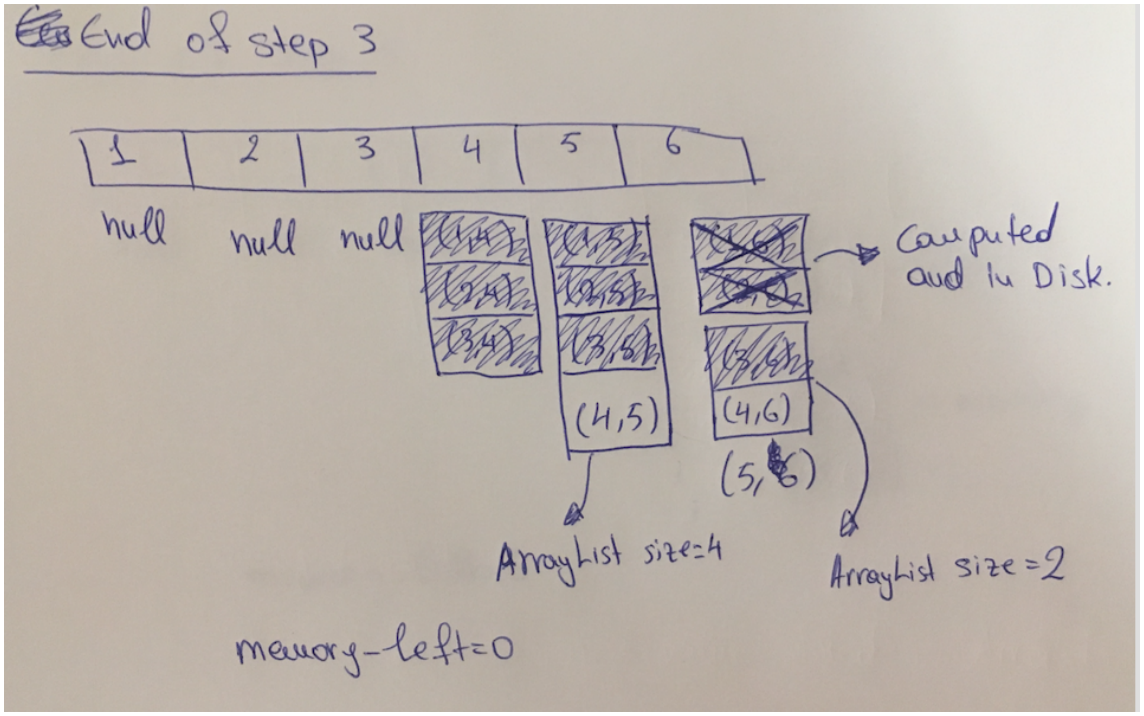
The data structure requires a hyper parameter INITIAL_SIZE that is determined based on the available RAM of the machine (users gives this). In the step 0, an array of size numOfUsers is created. Each cell of the array contain a primitive array list, and its array list is declared with initial capacity Math.min(INITIAL_SIZE, maxArrayListSize). The maxArrayListSize is the maximum size we expect the ArrayList to get, that is easily determined as for user n the max ArrayList size is gonna be $n - 1$. Finally a variable memoryLeft is declared with value 0.

At every step I load from RAM/Disk previous relevant correlations i.e. for user i I load all $Cor(j, i), 1 <= j <= i - 1$ and print those correlations to the matrix.csv, then I delete those correlations from the RAM and increment the memoryLeft. I compute and store all correlations $Cor(i, j) i + 1 <= j <= numOfUsers$ to both matrix.csv and ArrayList. If ArrayList is at max capacity I check if memoryLeft>0 and if yes I resize the array, if not I first store all the contents of the ArrayList to disk. And then since the ArrayList is not empty I store the correlation in the empty ArrayList. Photos follow showing the procedure.

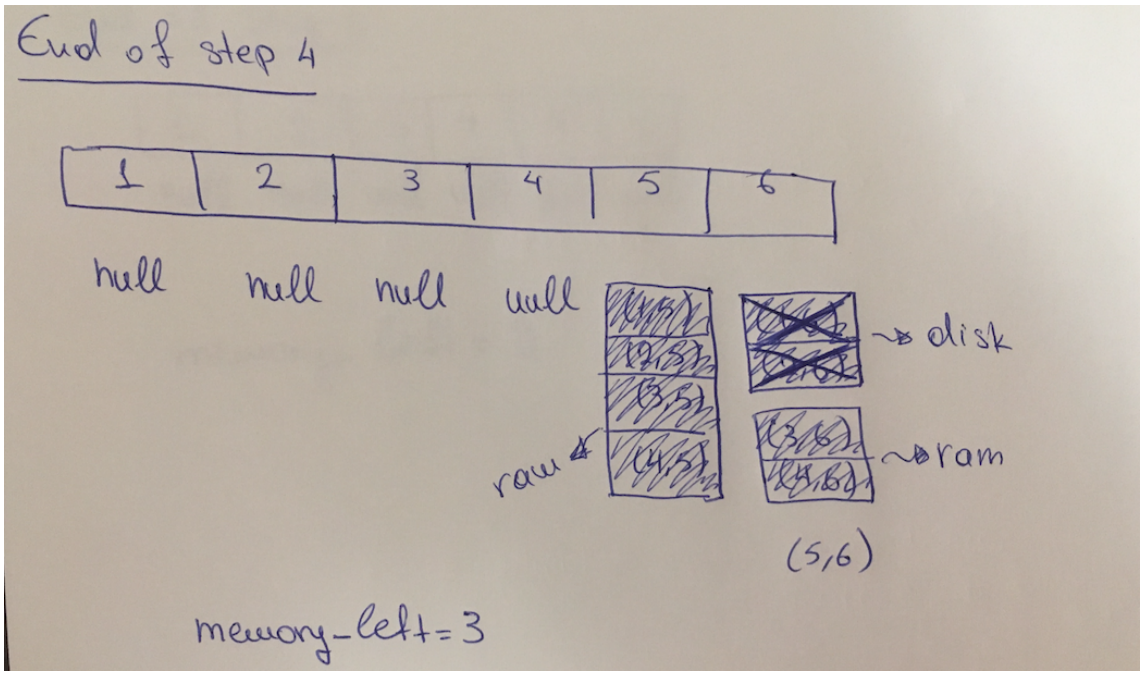


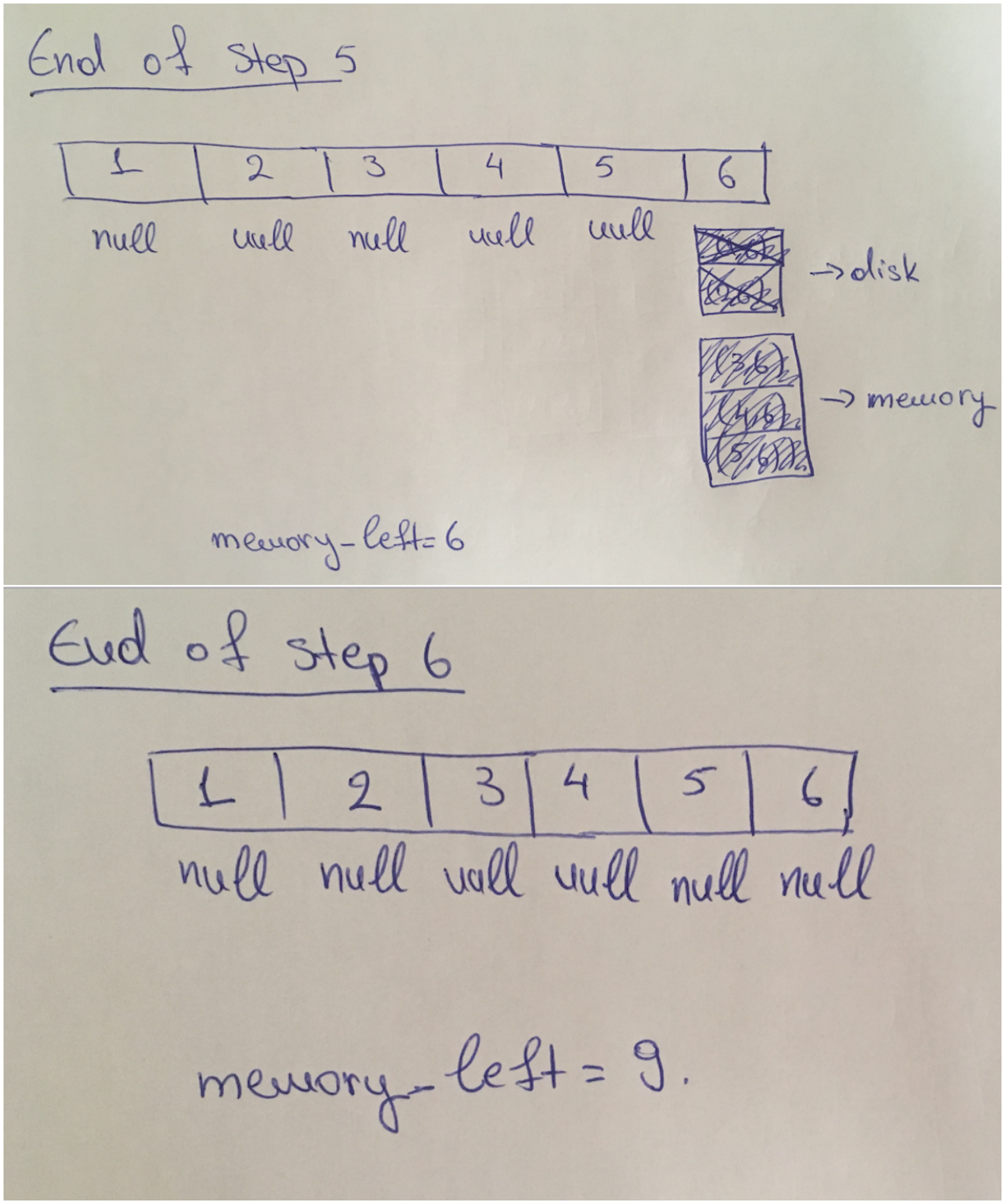


At the end of step 2 the array, at cell 2 no longer points to the ArrayList as a result some memory was freed. I increment the variable memory_left by the size of the ArrayList, in this case 1, thats why at the end of the step memory_left=1



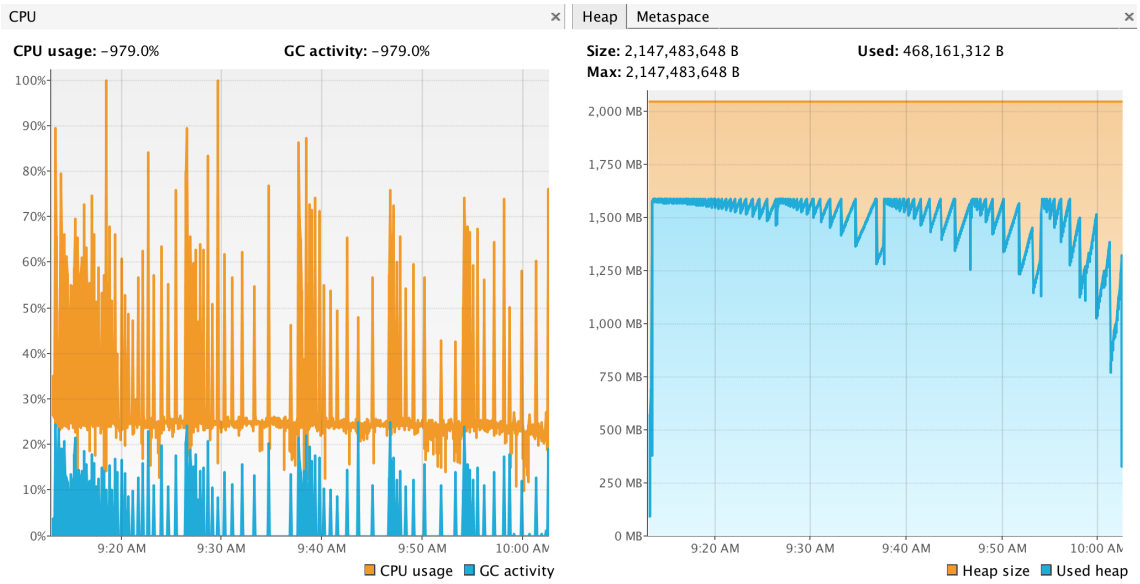
In the 3rd step the algorithm reads everything in the ArrayList at position 3 and prints it to the matrix.csv file. After that since the algorithm no longer needs that ArrayList it sets the array to null and increments the memory_left +=2. As a result memory_left=3. Next it computes the value Cor(3,4) and attempts to save it in the ArrayList in the 4th cell. The ArrayList at that time is already at its full capacity, the algorithm checks if the memory_left is greater than 0, and then resize the ArrayList to min(maxArrayListSize, currentCapacity + 3), in this case min(3,5) = 3, as a result the ArrayList gets larger and the memory_left is decreased by 1 to 2. The correlation is saved and then the algorithm proceeds to compute Cor(4,5) and the same procedure is used as before. After computing Cor(4,5) we no longer have free_memory and the Cor(4,6) cannot be saved. First the algorithm dumps to the disk all the contents of the ArrayList in position 6 and then stores the value in the now free ArrayList.





The algorithm reads from the disk and memory all computed correlations and prints them to the matrix.csv file. Even though its not obvious in this example all ArrayLists that are saved in the disk, are read back to the RAM only once during the execution of the program.

A photo follows with the memory and CPU profiling of the algorithm with the hyperparameter INITIAL_SIZE=10500 and -Xmx2G -Xms2G.



CPU samples | Thread CPU Time

Snapshot | Thread Dump

Hot Spots – Method	Self Time [%]	Self Time	Self Time (CPU)	Total Time	Total Time (CPU)
Pearsons2.**computeCorrelationsWithPrecomputedMeans**		3,050,205... (95.2%)	3,050,205 ms	3,194,989 ms	3,194,989 ms
SerializableListManager$myCustomArrayList.**loadFromFile**		85,348 ms (2.7%)	85,348 ms	85,348 ms	85,348 ms
SerializableListManager$myCustomArrayList.**saveToFile** ()		49,213 ms (1.5%)	49,213 ms	49,213 ms	49,213 ms
MovieHandler.**readData** ()		7,689 ms (0.2%)	7,689 ms	8,691 ms	8,691 ms
SerializableListManager$myCustomArrayList.**add** ()		7,305 ms (0.2%)	7,305 ms	56,518 ms	56,518 ms
SerializableListManager.**<init>** ()		2,142 ms (0.1%)	2,142 ms	2,142 ms	2,142 ms
MovieHandler.**computeMovieAverageRatings** ()		1,002 ms (0%)	1,002 ms	1,002 ms	1,002 ms
SerializableListManager.**addElementToList** ()		742 ms (0%)	742 ms	57,260 ms	57,260 ms
Pearsons2.**correlationWithPrecomputedMeansFastLookU**		32.4 ms (0%)	32.4 ms	32.4 ms	32.4 ms
SerializableListManager$myCustomArrayList.**get** ()		0.000 ms (0%)	0.000 ms	85,348 ms	85,348 ms
SerializableListManager.**getElementFromList** ()		0.000 ms (0%)	0.000 ms	85,348 ms	85,348 ms
com.intellij.rt.execution.application.AppMainV2$1.**run** ()		0.000 ms (0%)	0.000 ms	0.000 ms	0.000 ms
MovieHandler.**<init>** ()		0.000 ms (0%)	0.000 ms	8,691 ms	8,691 ms
Pearsons2.**main** ()		0.000 ms (0%)	0.000 ms	3,203,680 ms	3,203,680 ms

More than 95% of the time is spent computing correlations, while just 2.7% and 1.5% is spent for saving/loading intermediate results to/from the disk.

2.2 Collaborative filtering formulas

The collaborative filtering formula that performed best as the following.
($\hat{r}$ = predicted rating, NN = nearest neighbours, s= Pearson correlation, x=user, i=movie)

$$\hat{r}_{xi} = \overline{x} + \frac{\sum_{y \in NN} s_{xy} \cdot (r_{yi} - \overline{y})}{\sum_{y \in NN} s_{xy}} \tag{1}$$

In case of undefined correlation the prediction is the average rating of the user.

2.3 Experiments

Hyper parameters choices when generating the matrix are, use **precomputedMeans** (true/false), and **minimum number of commonly rated movies** sufficient to define a correlation, if the number is less than that then Float.Nan is returned, the motivation behind this is if 2 users with just 2 commonly rated movies have a high correlation it won't really mean much about how similar those users really are. The final hyper parameter is given when making the movie prediction in MovieRunner.java, **kNN** the number of nearest neighbors to consider.

First the hyper parameters for generating the matrix are presented and the time taken to generate the matrix.

initialSize=10000 (hyper parameter of the data structure)		
minCommonRatedMovies	precomputedMeans	Time (mm:ss)
2	False	33:45
10	False	31:10
20	False	33:39
40	False	31:47
60	False	32:21
2	True	33:53
10	True	33:22
20	True	30:12
40	True	32:13
60	True	31:18

As expected no significant difference between the run times for different minCommonRatedMovies. Also both ways of computing the correlation yield similar run times since both methods are one-pass.

precomputedMeans=false			
kNN	minCommonRatedMovies	Time (mm:ss)	RMSE
100	2	9:38	1.0703
500	2	9:58	1.0569
1000	2	11:31	1.0474
100	10	7:51	0.9644
500	10	7:24	0.9644
1000	10	11:15	0.9540
1500	10	15:44	0.9471
100	20	5:52	0.9604
500	20	5:38	0.9603
1000	20	9:47	0.9541
1500	20	13:41	0.9505
100	40	2:39	0.9862
500	40	5:19	0.9747
1000	40	7:44	0.9726
1500	40	10:14	0.9719

precomputedMeans=true			
kNN	minCommonRatedMovies	Time (mm:ss)	RMSE
100	2	13:38	1.0718
500	2	11:08	1.0652
1000	2	12:55	1.0596
100	10	5:42	0.9836
500	10	7:42	0.9571
1000	10	9:16	0.9468
1500	10	13:59	0.9404
100	20	3:34	0.9743
500	20	5:48	0.9536
1000	20	9:49	0.9475
1500	20	13:37	0.9454
100	40	2:09	0.9798
500	40	6:14	0.9747
1000	40	7:35	0.9673
1500	40	10:53	0.9668

Generally the trend for predictions is the more nearest neighbors used and the smaller the minimum number to define a correlation the slower the predictions.

The prediction accuracy appears to get better when the min number to define a correlation is 10-20. Larger numbers force the algorithm to predict the default value a lot and in the end not work well, as there are not so many users who have actually seen and rated 40 or more common movies and therefore users have fewer neighbors.

On the other hand very small numbers i.e. 2 will make users that are not really correlated in reality to appear very similar just by chance. I expect 2 users that have very similar tastes to have watched many common movies and have a high correlation but not a correlation of 1.0. When using a large kNN then the users that are not truly related will have a larger influence to the prediction because they have a larger correlation. When using a small kNN though the results are very bad since this time the truly similar users will be pushed out of the neighborhood completely leaving as neighbors only the users with just 2 commonly rated movies that have correlation of 1.0 by chance. Finally it should also be noted that those few common movies most of the times will be just popular movies everyone has watched, i.e. Harry Potter, and therefore the correlation is completely useless.

Between the two different methods of computing the correlation I notice better RMSEs when making predictions with precomputed means.

Overall the best configuration appears to be precomputedMeans=true, minCommonRatedMovies=10, kNN=1500.