# Frequent Item Sets

## Introduction:

In this project I implemented various Algorithms to data mine two datasets, the retail data which consisted of 88,162 baskets, and the larger Netflix data set which had 480,188 baskets. I implemented the Apriori algorithm using both the Triangular array and the Triples method, PCY, Multistage, Sampling and SON as well as its Distributed version.

## Experiment:

Here are the numbers of frequent items and frequent pairs of the two datasets using 1% and 2% threshold

| Dataset | Threshold | Frequent Items | Frequent Pairs |
|---|---|---|---|
| Retail | 1% | 70 | 58 |
| Retail | 2% | 190 | 22 |
| Netflix | 1% | 3221 | 1220351 |
| Netflix | 2% | 2099 | 529133 |

The numbers were identical across all algorithms except for the Sampling.

**Sampling Algorithm results:**

1) Data size = 20%
2) Sample Threshold = Threshold **/ 6**
   These parameters minimized the false negatives

| Dataset | Threshold | Frequent Items | Frequent Pairs | False Positives | False Negatives |
|---|---|---|---|---|---|
| Retail | 1% | 97 | 109 | 56 | 5 |
| Retail | 2% | 36 | 22 | 8 | 0 |
| Netflix | 1% | 3575 | 1483208 | 262857 | 0 |
| Netflix | 2% | 2396 | 671669 | 142536 | 0 |

1) Data size = 20%
2) Sample Threshold = Threshold **/ 5**
   These parameters gave few false positives and few false negatives, but still a good approximation of the actual numbers

| Dataset | Threshold | Frequent Items | Frequent Pairs | False Positives | False Negatives |
|---|---|---|---|---|---|
| Retail | 1% | 71 | 81 | 30 | 7 |
| Retail | 2% | 23 | 25 | 3 | 0 |
| Netflix | 1% | 3227 | 1220528 | 14198 | 14021 |
| Netflix | 2% | 2099 | 530090 | 5610 | 4653 |

# Notes

I have measured the running time of these algorithms using the retail and Netflix datasets, I have tried to optimize the code to run efficiently and save space and time, here are some of the things that I did to achieve the optimized results.

1. The data type I used to count is Short instead of Integer, this is because the maximum support is going to be 9604 on the Netflix dataset using 2% threshold. 9604 can be represented as 2 bytes short, which saves us 2 bytes per count as opposed to using 4 bytes integers.

2. I do not have to keep counting occurrences of items after they pass the minimum support, this will minimize “get” and “put” operations when using an algorithm that has a Hash Map as its data structure, since there will be collisions and the operations will not be in constant time.

3. I made my own Pair class since Java does not have one, the class assumes pairs are in lexicographical order such that Pair(x,y); x < y.

4. I implemented the CompareTo method for the pairs such that pairs are ordered by X first then by Y, this will make the hash map faster in case of collisions since it will convert to a tree instead of a linked list, where the search will be in O(log N) times instead of O(n).

5. I have noticed that primitives are immutable in Java, so instead of the Hash Map being of type <Pair, Short>, I made it of type <Pair, MutableShort>, where MutableShort is a wrapper object that contains the count, and a method to get the value and another method to increment, this reduces the number of “Put” operations and gives us faster access time when incrementing the count. [1]

6. I have also noticed that the modulo (%) operation is slow when hashing the pairs to a bucket, so instead I made the array length a power of 2 number, where the modulo can be calculated using Bitwise AND operation, this saves time in the PCY algorithm. [2]

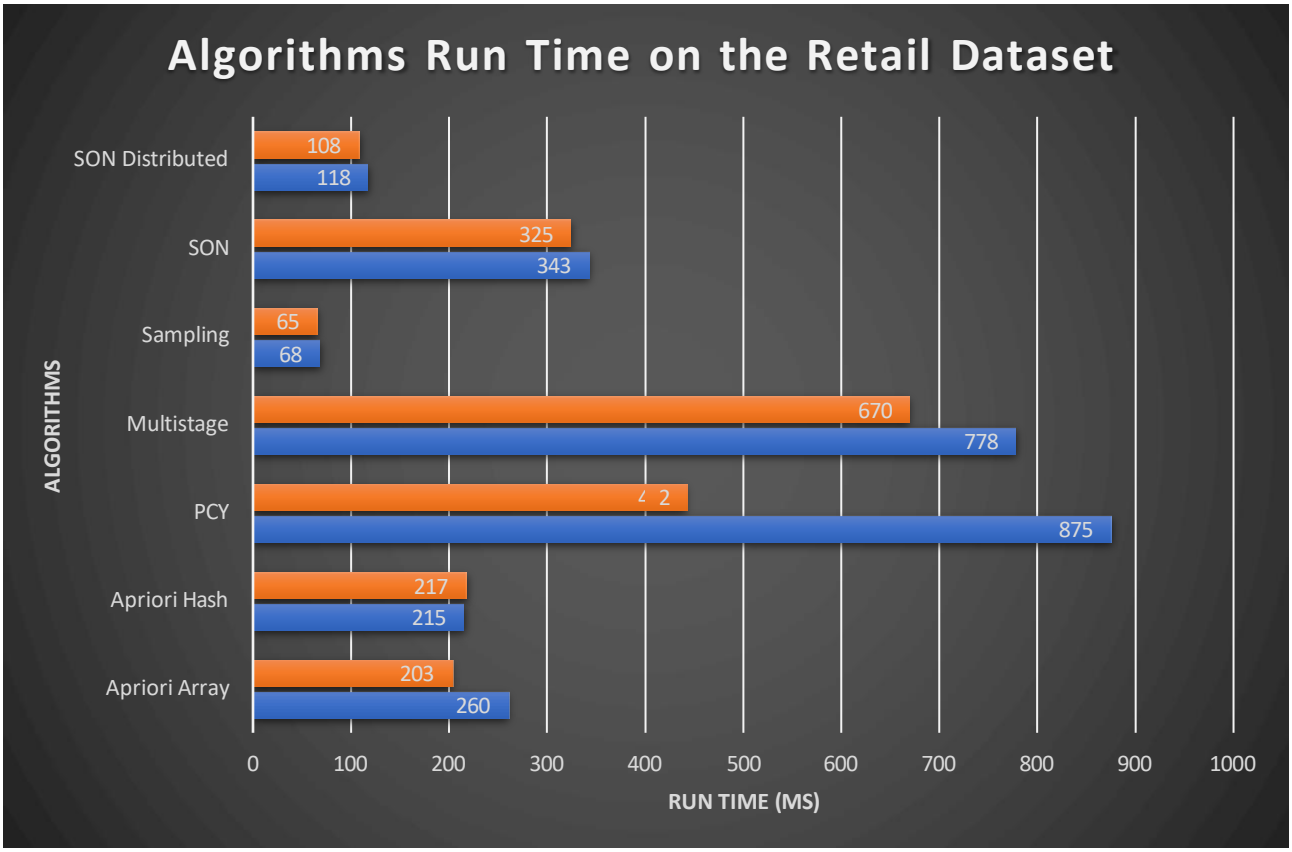
7. The first hash function I am using for the PCY algorithm is the one used in the triangular method array to convert the matrix in to 1 dimensional array: “(((y) * (y - 1)) / 2) + ((x))”. [3] the other function in the multistage is The Szudzik pairing function which also maps two numbers to a unique number. [4].

8. There are 67108864 buckets of Shorts, so the buckets consume 134217728 Bytes of memory on the first pass which is 128 MB.

9. SON Distrusted algorithm works by running the chunks in parallel and counting the pairs in different threads then combining the count in the main thread. Each thread has its own data structure to store the count to avoid race conditions.

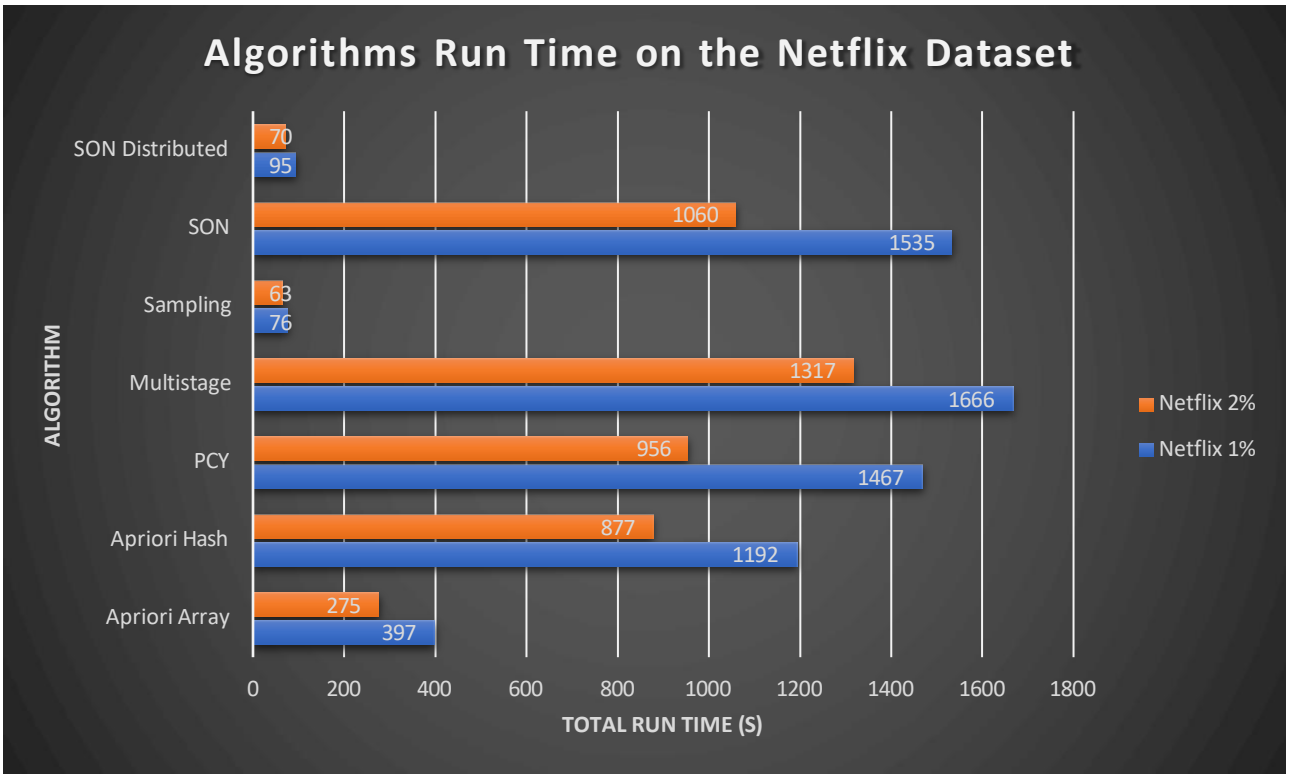
# Retail Run Time:

| Algorithm | Retail 1% (ms) | Retail 2% (ms) |
|---|---|---|
| Apriori Array | 260 | 203 |
| Apriori Hash | 215 | 217 |
| PCY | 875 | 442 |
| Multistage | 778 | 670 |
| Sampling | 68 | 65 |
| STN | 343 | 325 |
| STN Distributed | 118 | 108 |



# Netflix Run Time:

| Algorithm | Netflix 1% (S) | Netflix 2% (S) |
|---|---|---|
| Apriori Array | 397 | 275 |
| Apriori Hash | 1192 | 877 |
| PCY | 1467 | 956 |
| Multistage | 1666 | 1317 |
| Sampling | 76 | 63 |
| STN | 1535 | 1060 |
| STN Distributed | 95 | 70 |

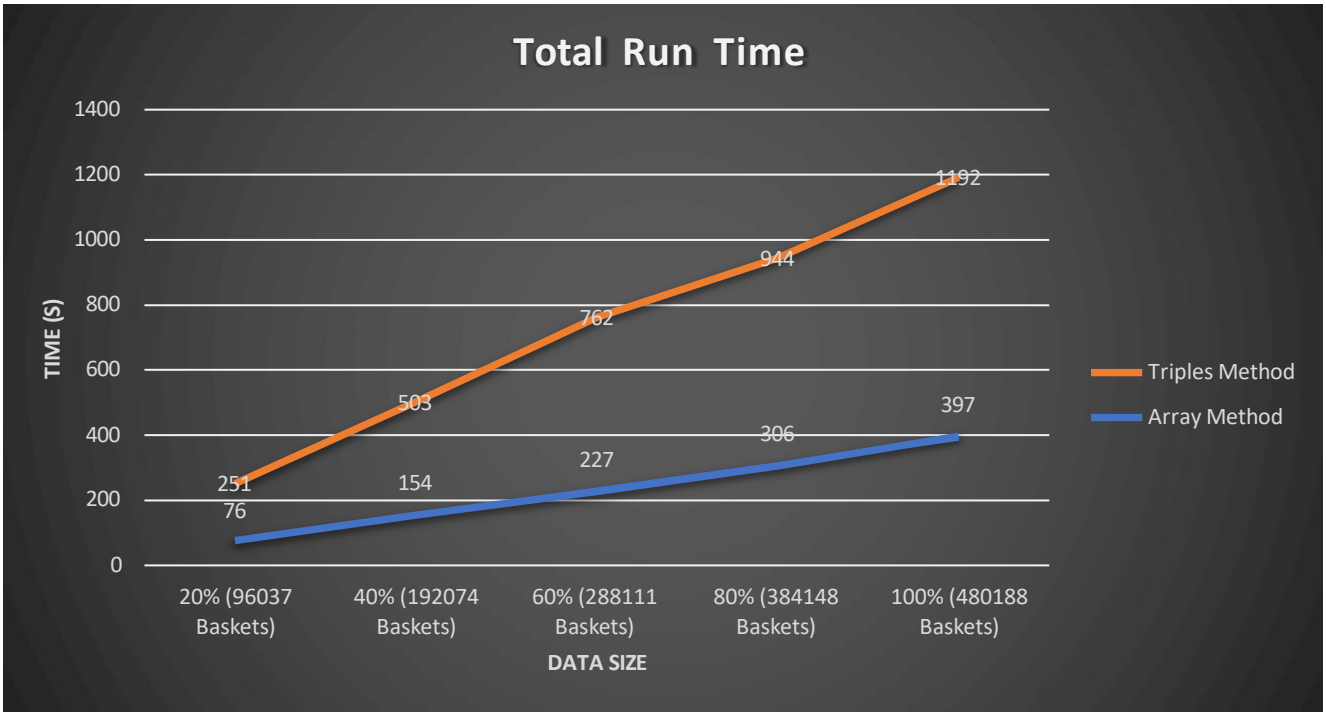


# Scalability
## (Apriori Array Method)

| Netflix Data Size | First Pass | Second Pass | Total |
|---|---|---|---|
| 20% (96037 Baskets) | 1 | 75 | 76 |
| 40% (192074 Baskets) | 3 | 151 | 154 |
| 60% (288111 Baskets) | 4 | 222 | 227 |
| 80% (384148 Baskets) | 6 | 300 | 306 |
| 100% (480188 Baskets) | 7 | 389 | 397 |

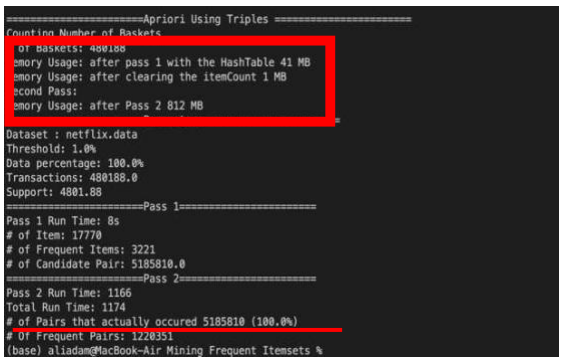
## Apriori Triples Method:

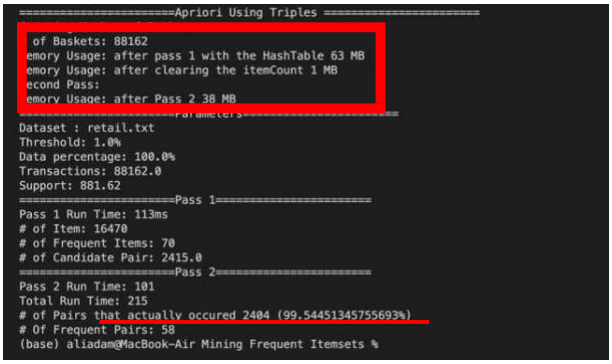
| Netflix Data Size | First Pass | Second Pass | Total |
|---|---|---|---|
| 20% (96037 Baskets) | 1 | 249 | 251 |
| 40% (192074 Baskets) | 3 | 499 | 503 |
| 60% (288111 Baskets) | 4 | 757 | 762 |
| 80% (384148 Baskets) | 6 | 938 | 944 |
| 100% (480188 Baskets) | 8 | 1184 | 1192 |



# Observations:

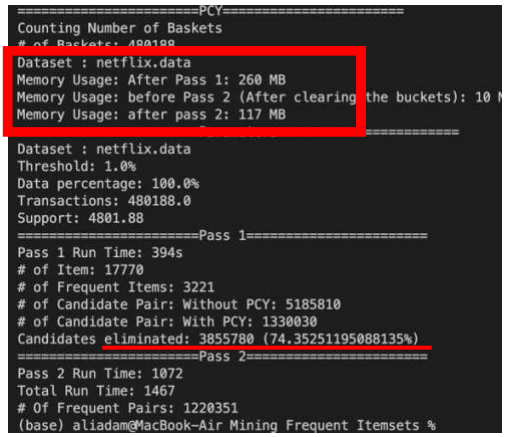
1. **Triangular vs Triples Method:** The triangular array method is much faster than the triples method and uses less memory, I believe it is faster due to the way hash maps are implemented in java and due to the high amount of collisions, it takes a lot of time to find the pair and retrieve its value compares to accessing an array index, this leads me to believe that the hash table buckets are turning into linked lists or trees and its taking longer than constant time. As for memory:

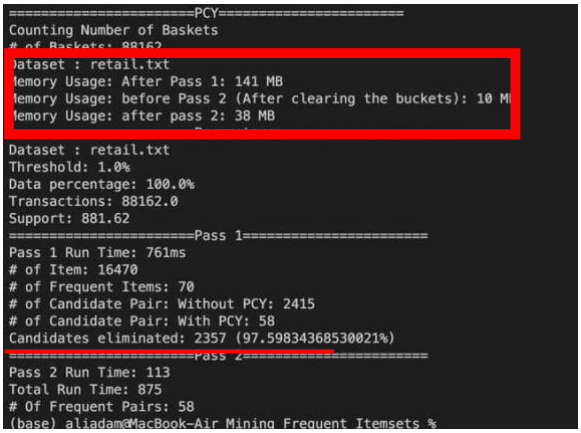



All candidate pairs appear in the data set, so the triples method is not efficient since it takes up 3 times as much space compares to the array if not 4 times due to the collisions and storing the next pointer in the linked list.

2. **Apriori vs PCY**: When comparing Apriori and PCY, I will be comparing the Apriori that uses the Triples method since it uses a hash map also which is fairer to compare with PCY. I noticed that overall PCY is slower than the Apriori due to the long run time of pass 1, However, PCY's second pass is faster than Apriori's because it has less candidate pairs to count, and uses significantly less memory because it eliminates about 74% of candidate pairs



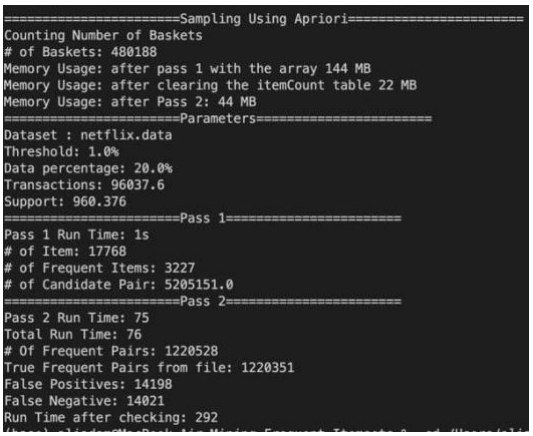

We can see the memory used is 117 mb vs 812 mb after pass 2 in the Netflix dataset after eliminating 74% of the candidate pairs. If the dataset was larger, then the Apriori will be very

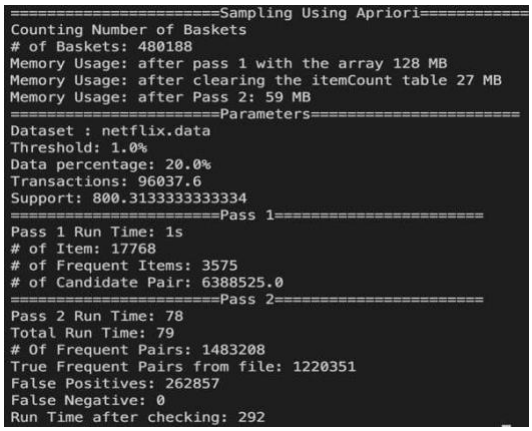
slow on the second pass as it will use up all the free memory and resorts to I/O operations from the hard disk while the PCY will have more space and can do all the computation in the main memory.

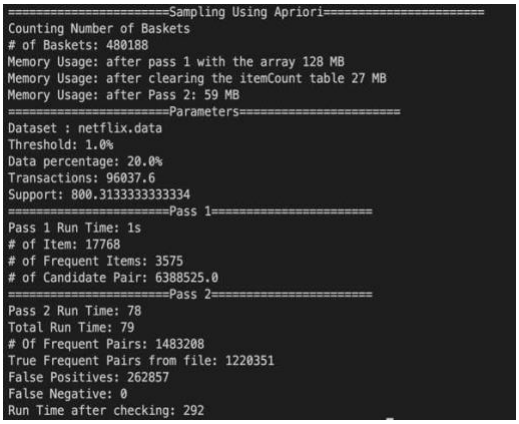
3. **Sampling:** The sampling algorithm runs very quickly as its only processing a 20% sample of the dataset, I used different combinations of dataset size and sample thresholds and I found two suitable ones:

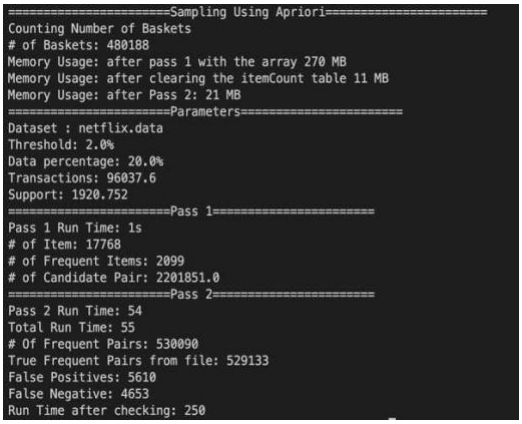
   **20% Sample and (Threshold/5) Sample threshold**: This gave me **5610 false positives** and **4653 false negatives** using the 2% global threshold. And **14198 false positives** and **14201 false negatives** using the 1% global threshold If we care about minimizing false positives and we are fine with missing few true positives pairs, then this gives us a very close result.

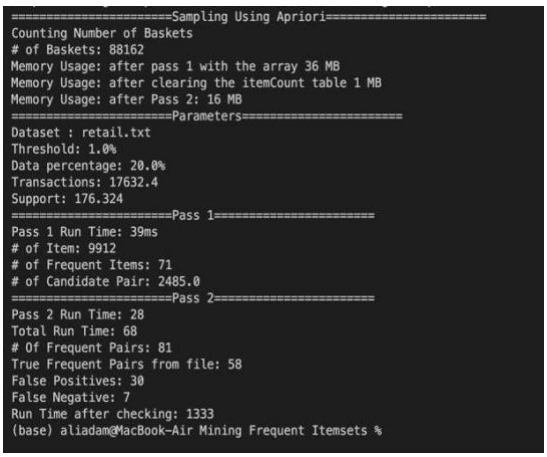
   **20% Sample and (Threshold/6) Sample threshold**: This makes the threshold smaller to capture more pairs that were false negatives, but this will increase the number of false positives by a magnitude. There were no false negatives, but I got **262857 false positives** using 1% threshold and **142536** using 2%. If we are trying to capture all true positives this will be better since I got 0 false negatives on the Netflix data set and it runs very quickly.

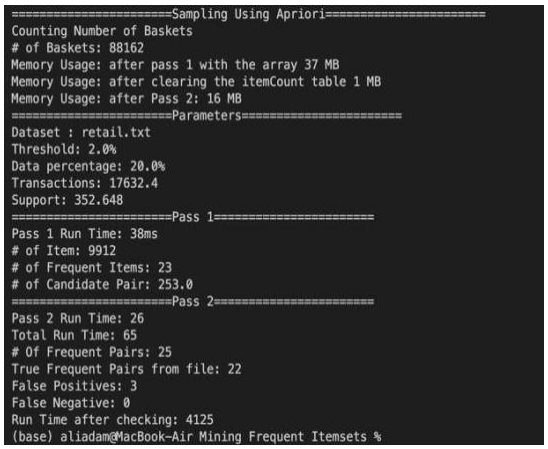


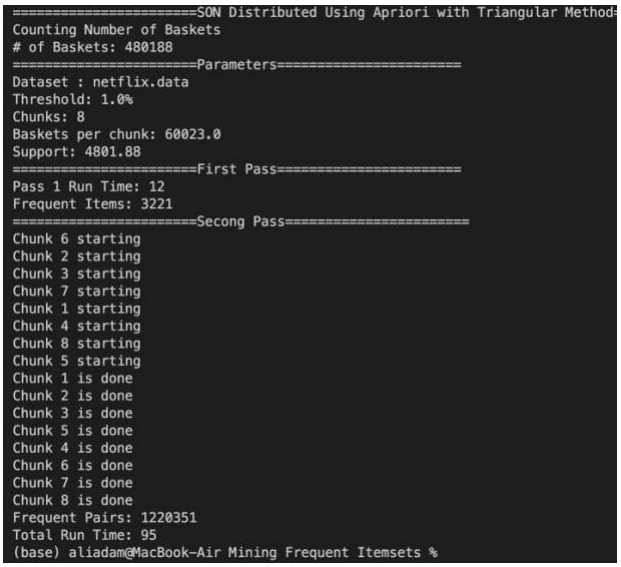



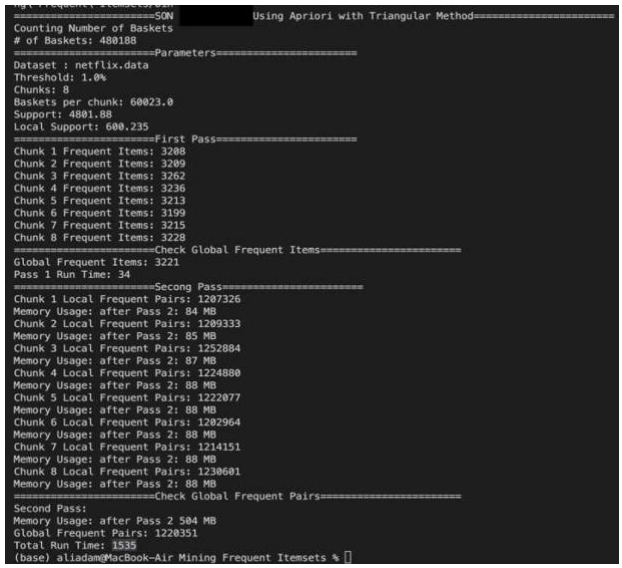





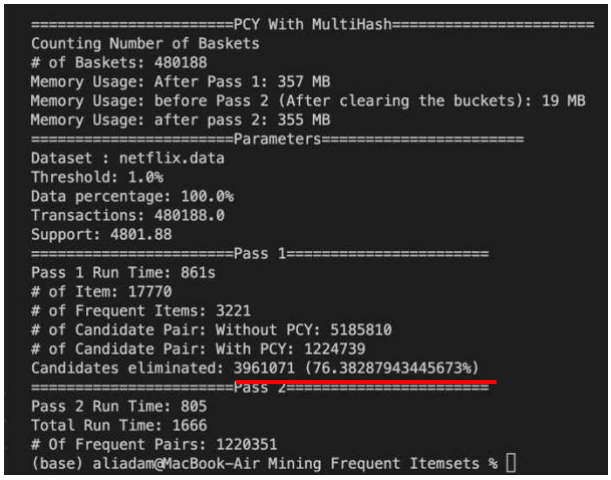



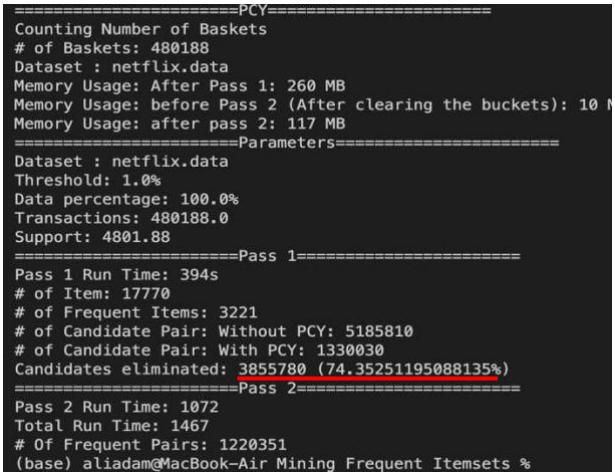

4. **SON:** The version that processes each chunk at a time will save us space because at any instant of time the amount of memory used is 1/8$^{th}$ of that in Apriori, since I divided the file into 8 chunks. Although it's a bit slower than Apriori, but on larger scale it will not run out of memory space.
    a. **Son Distributed:** This is the fastest implementation of the algorithm in this project, since each chunk is independent, I processed each one in parallel in different threads then combined the count in the main thread, this utilizes all CPU Cores that were idle in Apriori and it's running the entire data set in less than 100 seconds. Although this uses more memory space than processing each chunk at a time, it uses the triangular array to its still using less space than the triples method.



5. **Mutli-Stage:** The goal of the multistage algorithm is to eliminate more candidate pairs on the stages before the second pass using another independent hash function, on the second stage I was counting pairs that only hashed to a frequent bucket from the first pass, then saving those that are hashing to 2 frequent buckets for pass 2.




The multistage algorithm was only able to eliminate 2% more candidates, I believe this is due to the distribution of the data in the Netflix dataset since I used two independent hash functions. It is taking longer to optimize but only eliminate 2% so it is not that much improvement.

# Conclusion:

I learned how each algorithm works, the advantages and disadvantages, when to use each one. Overall the distributed SON was the fastest since it uses all cores to process the data, then comes the Apriori that uses the array. As for memory, PCY consumed the least amount of memory on the second pass because it has less overhead than mutli stage, since Multistage didn't eliminate much more candidates but we had to save its bitmap. Sampling gave us very close results and was a good approximation of the data.

Citations:

**[1]:** Code Utility. "Optimization - Most Efficient Way to Increment a Map Value in Java." *Code Utility*, 18 Mar. 2022, https://codeutility.org/optimization-most-efficient-way-to-increment-a-map-value-in-java-stack-overflow/.

**[2]:** Newland, Chris. "High Performance modulo Operation." *RSS News*, https://www.chrisnewland.com/high-performance-modulo-operation-317.

**[3]:** Leskovec, Jure, et al. *Mining of Massive Datasets*. Cambridge Univ. Press, 2014.

**[4]:** *An Elegant Pairing Function - Szudzik*. http://szudzik.com/ElegantPairing.pdf.

# How Each Algorithm Works

## Apriori:

The algorithm starts by reading the file and counting the number of lines, which will be the number of baskets. This will be used in order to calculate the support.

**First Pass:** Read file line by line, the line will be a string, which gets parsed into an array of strings by the white space as a separator. Then we loop through that array, and convert each string into an integer, which represents the item. Using a Hash Map, we keep count of the occurrences of items.

After the first pass is complete and we have the count of each item, we loop through the hash map entries to determine which items have a support more than the minimum support. The hash map is summarised as a bitmap to save space. The Hash Map takes 8 bytes of space per item count (4 byte for the item, 4 Bytes for the count value), where the bitmap takes up 1 bit of space per item (Indexes represent item numbers, and the value at that index is either True or false which is a 1 bit value). The bitmap would take approximately 1/64 of the hash map space.

Now we map the frequent items into 1,2,3,. ...N, where N is the number of frequent items. This renumbering will help make the triangular array much smaller, saving space. The mapping is saved in 2 hash maps, one to map to the new numbering, and another to map back to the original item number.

> **Triangular Array Method:** I implemented the Apriori using the triangular array method by generating candidate pairs from the frequent items in a double loop, using the formula:
>
> Since there are a maximum of 480188 baskets, and the max support is 2%, then I only need to count up to 9603, I can use the Short data type instead of Int, which takes up 2 bytes instead of 4, saving 50% space in memory per pair.
>
> **Triples Method:** I also implemented the triples method in another file, which uses a hash map instead of an array. This method will take up 6 bytes of space per entry (4 bytes for the pair of shorts, and 2 bytes for the count value).

**Second Pass:** Read the file again line by line and parse the string into the array of items, check if that item is frequent using the bitmap, if it is, we add it to a temporary list of frequent items in that basket. After finding the frequent items in the basket, generate the candidate pairs in a double loop and increment the count in the data structure.

After the second pass we loop through the data structure, and check for the pairs that have support higher than the minimum support and add them to a list after mapping the items back to the original numbering.

# PCY:

This algorithm utilises the empty space in memory on the first pass, it uses an array of shorts (2 bytes) to hash pairs into, hoping to eliminate some of the candidate pairs so we don't have to count them on the second pass, saving space in memory.

This algorithm needs to eliminate ⅔ of the candidate pairs to save space, because we are forced to use the triples method to count the pairs, since we don't know which items are eliminated by the buckets.

**First Pass:** This step will be the same as the Apriori, with one additional step, after incrementing the counts of items in the basket, we generate the pairs of that basket, and hash the pair into a bucket, incrementing the count in that bucket

The buckets are summarised in the bitmap as well as the frequent items, to save space for the second pass.

After the first pass, we get the frequent items, then generate the candidate pairs, but before adding them into the hash map, we check if that pair hashes to a frequent bucket that has a count equal or higher than the min support.

**Second Pass:** Just as in the Apriori second pass, count the pairs in the hash map, but now we might have less pairs to count, which should save us space.The rest is the same as Apriori.

# Multistage:

This algorithm works the same way as the PCY, but uses 2 passes where pairs are hashed on each pass to buckets using independent hash functions. The functions used are:

1. From the Mining of the Massive Dataset book, this function is used in the triangular method to map 2 numbers into a unique number
2. The Szudzik Function is another function that also pairs 2 numbers into a unique number

Those 2 hash functions are independent, so their mappings are different. For a pair to be candidate, it needs to map to two frequent buckets from each hash function.

# Sampling:

This algorithm reads a sample of the file, I used %30, then ran the Apriori algorithm to count the items and pairs, then compared the number of frequent pairs with the actual number that I got from the other algorithms and quantified the number of false positives and false negatives.

# SON:

I divided the files into chunks, and ran Apriori's first pass on each chunk separately, then combined the local frequent items, went back, and counted only those items to get the true positives. A true frequent item must be frequent in at least one chunk.

This saves space because we must count less items at once, saving space in memory and preventing the I/O operations with the Hard Disk

Then for the second pass, I did the same thing, process one chunk at a time, combined local frequent pairs and then checked which of those are truly frequent

**Distributed version:** I also implemented the distributed version of SON, by splitting the file into 8 chunks, I ran the Apriori first pass on each chunk in parallel in threads, and then combined the count from each thread and got the frequent items. This algorithm was the fastest to process the Netflix dataset since all cores of the CPU are being utilised, and each chunk is independent.

The second pass works the same, count pairs in different threads then combine the count and check for frequent pairs that pass the threshold.