Report

**PC Specs**

CPU: Intel® Core ™ i5-8250U CPU @ 1.60 GHz

OS: Windows

RAM: 8.0 GB at 2400 MHz speed

**Introduction**

For this project, I performed a scalability study for finding frequent pairs of elements by dividing the dataset into different chunks and measuring the time performance. On each of the different chunks, I implemented four algorithms: A-Priori, PCY, Multistage, and Multihash with different thresholds of 1%, 5%, and 10%. For this project, I used the Python programming language. To start off, I will briefly describe my implementation of each of the algorithms below:

A-Priori

For the A-Priori algorithm, I started off by computing the number of lines that will be gone through and the threshold that corresponds to the number of lines calculated. The next step is to find the frequent singletons. This is done through a function that starts off with a dictionary of all the singletons and an array of the frequent singletons. As we advance through the lines, the value that corresponds to key that equals to the current singleton is increased by one in dictionary, and if that incrementation by one results in the value equaling to the threshold, the number is appended to the list of the frequent singletons. Once all the frequent singletons are found, the candidate pairs are found by taking all possible pairs of the frequent singletons. After this, we go through each line again and see which pair exists within which line, and similar to the logic for finding the frequent singletons, the frequent pairs are found.

PCY

For the PCY algorithm, there are two passes. I started off by computing the number of lines that will be gone through and the threshold that corresponds to the number of lines calculated. The next step is to perform the first pass. In the first pass, the frequent singletons are found and every possible pair is hashed to the hash table, a dictionary. The process of finding the frequent singletons is similar to that of A-Priori. The difference in PCY is that when a singleton is being examined, I also take all possible pairs of that singleton with the rest of the items in that basket and hash each of those pairs to that hash table using the has function (i+j) % 100000. Each pair is hashed to the corresponding bucket, and if the value at that hash code is now equal to the threshold, the value at that hash code is changed to one in the bitvector, another variable declared at the beginning of the first pass. In the second pass, all the candidate pairs are found by taking all the possible pairs of the frequent singletons and checking which of those pairs hash to a value of one in the bitvector. Once the candidate pairs are found, I then go through the file again counting the frequency of each pair.

Multistage

For the Multistage algorithm, the implementation is similar to the PCY algorithm, but with one added pass in between. In the second pass, I similarly go through the data chunk and make pairs from all the singletons in each basket and pass it through a second hash function which hashes it to a second hash table. However, a pair is only hashed if both its items are frequent and the bit that it hashes to in the first bitvector is one. If these conditions are met, then it is hashed again, and likewise if the position it is hashed to ends up beng equal to the threshold, then the bit that corresponds to that same hash value is set to one in the second bitvector. Therefore, in the third pass, I then create all possible pairs from the frequent singletons, and if a pair hashes to bit of one in both the first and second bitvectors, it will be considered as a candidate pair. Once this is done, the frequencies of each of the pairs is counted in the files, and only those that reach the threshold are returned.

Multihash

For the Multihash algorithm, the implementation is also similar to PCY, but there are some changes. In the first pass, instead of passing each pair in the basket to one hash function, they are passed through two hash functions which correspond to two different bitvectors. Thus, once the frequent singletons are found, in the second pass, I create all possible pairs from these singletons and consider them as a candidate pair if they hash to a bit of one in both the first and second bitvector.

**Results**

Chart, line chart

Description automatically generatedChart, line chart

Description automatically generatedIn order to perform the study, I used the Python time library to track the time for each algorithm to execute and the matplotlib library to create a line graph to visualize the results. I used a loop to go through each data chunk and appended the run times to four different arrays that were used to construct the lines corresponding to each algorithm. The results are as follows:

Chart, line chart

Description automatically generated

The two glaring observations that can be made based on these results is that the A-Priori algorithm performed the most optimally in terms of time, and the PCY algorithm came a close second place. For the 1% and 5% thresholds, the PCY was second place, and for the 10% threshold, it was second place for the most part. However, when we look at the algorithms, the PCY algorithm is definitely much better in terms of memory usage since it makes use of all the memory whereas with the A-Priori algorithm a lot of the memory is idle. When it comes to the time performance, the PCY algorithm has one more pass to it, and that pass results in more computations and thus more time. Even though, based on my testing, the PCY algorithm removes 2/3 of the possible pairs for the 80%, 90%, and 100% data chunks, it still takes more time to execute than the A-Priori algorithm. The discrepancy lies with the Multistage and Multihash algorithms. In general, we find the pattern that the Multihash function executes slower than the Multistage algorithms, and the reason I believe this to be the case is the size of the buckets and the hash functions. With the Multihash algorithm, all the hash tables in the first pass take up as much space as just one hash table in the Multistage algorithm, and thus the pairs will not be as spread out, and thus there are more false positives generated. With there being more false positives, there is much more computation that takes place then in comparison to Multistage. At the end of the day, it all boils down to the hash function that is used. If the most optimal hash function is used, then the algorithms would run much faster.