For this implementation of a Hash Table, I expanded on the source code to allow for generic type keys in addition to generic values (limited to float, int, and string). I tested the hash table by fully loading it with random data, with URLs from ted-talks, and alphanumeric unique ids from a fake news data set, I'll include the results of each hash compared to the ideal at the end of this analysis*.

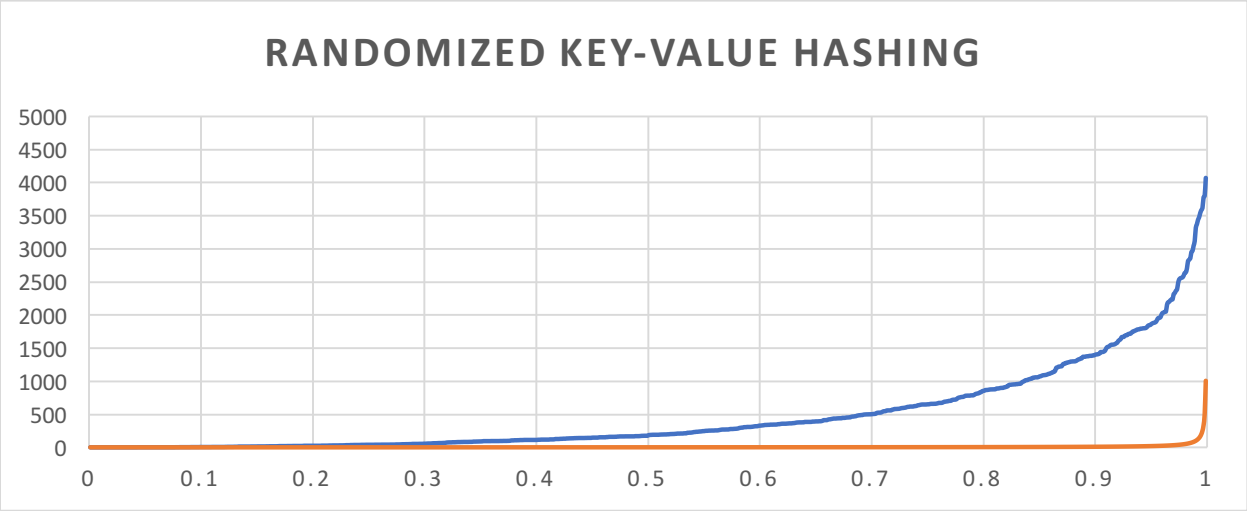
As for the implementation of the hash table, I needed a hashing algorithm that worked on the byte level in order to work with strings and to achieve a decent avalanche. I settled on a version of the Jenkins hash for my primary hashing algorithm because of its praise, avalanche capability, and ability to work with char arrays. Additionally, I used linear probing as my collision resolution method where the probe distance is determined by a second hash "the One-At-a-Time Hash" which is also made by Bob Jenkins and is considerably simpler than the primary hash I used. I found that using a varied probe distance reduced my overall collisions by an estimated 15%.

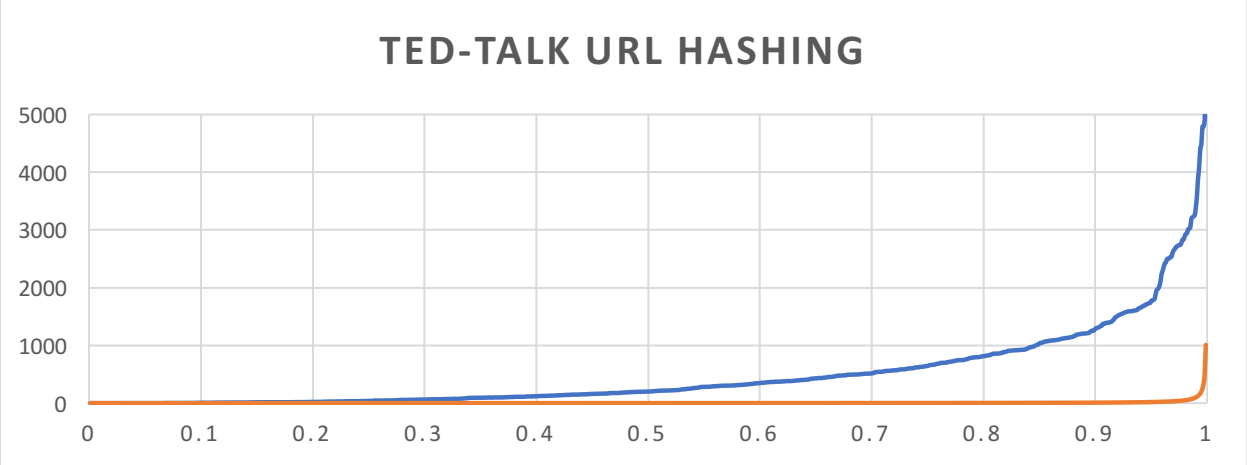
Interestingly, the hash performed very consistently across the three data sets (random, URL, and unique ID), clustering can be seen impacting the URL the most around an alpha of .96, and not dramatically at .50 in any data set. However, the table varies considerably from the ideal, despite the (subjectively) low level of clustering. It's hard to say whether the clustering that did happen was the result of primary or secondary clustering as I did not track the difference, nor the keys distance from home. Rather, I scanned the printed table to look for clusters and found, on average, a fairly well distributed hash table. For 504 inserts, I did find a cluster of 12 once, but in further tests, I found typical clusters maxed at 4. In a more advanced implementation of this focusing on finds, and especially if disk reads were costly, I would implement a robin hood hash would takes advantage of a clever trick to reduce probe length variance and make better use of the cache. Since none of these things were desired here, I chose to implement the simpler hash table.

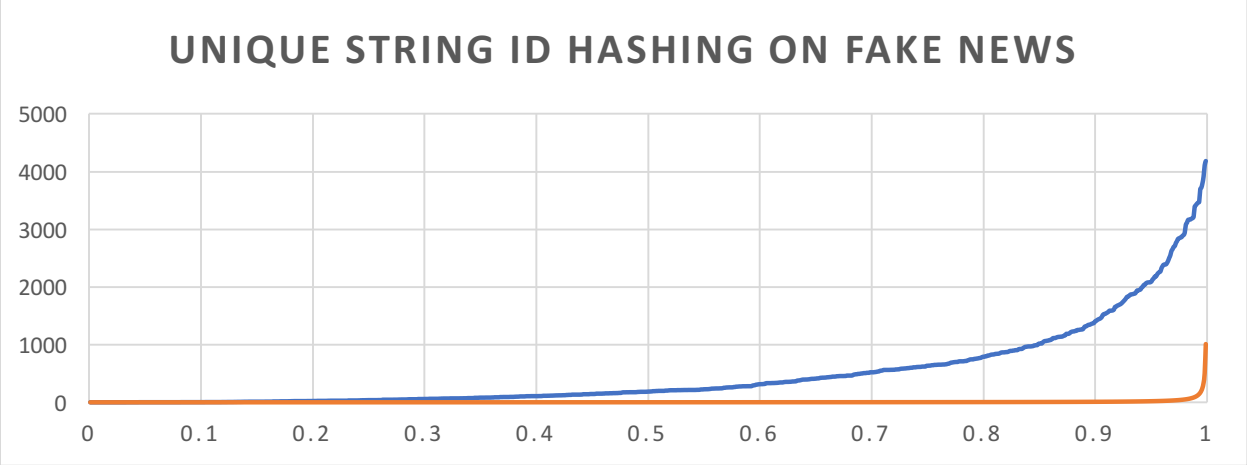
Some notable issues I ran across: the MAXHASH value was not prime, and put forward a risk of looping forever, I made it prime. The probe distance would occasionally be a multiple of the MAXHASH value and would cause a loop, I solved this by reducing the probe length potential. To achieve a general process to hash different types, I utilized the std::hash module that calculates a hash value- for primitive values it returns the same value, but for strings it calculates a unique id, which was very helpful.

* On further reflection, it is possible that I portrayed the graphs incorrectly ☺. These represent the total number of collisions as a function of alpha, rather than the average collisions. I gathered this data by outputting `cout<<table.alpha()<<","<<collisions<<endl;` on every insert from 1 to 1009 (the max) into a .csv file, which I then opened in Excel to graph. I considered dividing the collisions at each point of alpha, by the current size at the time, but that resulted in a value lower than the ideal, which seemed less likely than my grossly worse result. In response, I'll leave it as is.

** Note: if I considerably messed up one of the extensions, please feel free to disregard it. I can likely get you a version that does not implement it since I version controlled with git.


RANDOMIZED KEY-VALUE HASHING
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


TED-TALK URL HASHING
5000
4000
3000
2000
1000
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1


UNIQUE STRING ID HASHING ON FAKE NEWS
5000
4000
3000
2000
1000
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1