Contents

Hashing Analysis

Discussion of Plots

After getting the results object, I organized the data into respective x, y coordinate pairs and utilized the JavaScript library, Plotly, to generate interactive graphs. This way I could generate graphs more efficiently for different experiments by avoiding exporting data to .csv and fitting them to graphs in excel.

There are four plots, one for each experiment. Each plot has three different lines that correspond to a trial based on table size. Their values and color can be found in the legend at the top right of each plot.

The x-axis represents load factor and ranges from 0 to 1, following increments of 0.2. The y-axis represents numbers of collisions at any load factor. The range of y-values changes in some of the experiments, but it is generally around 0-30 for table sizes of 10, 25, and 50.

Every trace has data points incremented by $x = \frac{1}{table\ size}$, therefore the bigger the table size, the more results you there are to work with. Each point is connected by a line of the same color.

In trials with smaller sized tables (Page 3), the lines in the graph seem to be increasing linearly. However, when table sizes of 10000, 25000, and 50000 are used in the experiments (page 5), every trace clearly displays positive exponential growth with collisions reaching up to 30000.

Effects of collision resolution scheme

The effects of collision resolution screen are clearly displayed in the experiment's results. In both pairs of experiments, the trials that utilized the separate chaining scheme finished with less collisions than the open Addressing scheme. In the experiment with the largest hash table size (50000), separate chaining reached a load factor of 1 with around 15000 collisions while the trial with open addressing had reached up to 25k collisions

This makes sense and displays what I believe is the main advantage of using separate chaining over open addressing. Separate chaining, at any point during the experiment left more spots in the array open, as collisions wouldn't fill up spaces that didn't *technically* belong to the key value in its current iteration. Because of this, there would be the same number of open indices in the hash table after a collision handled by separate changing. In the open addressing trials, open indices are filling up no matter what, therefore after every single collision, the chance of another collision occurring increase slightly.

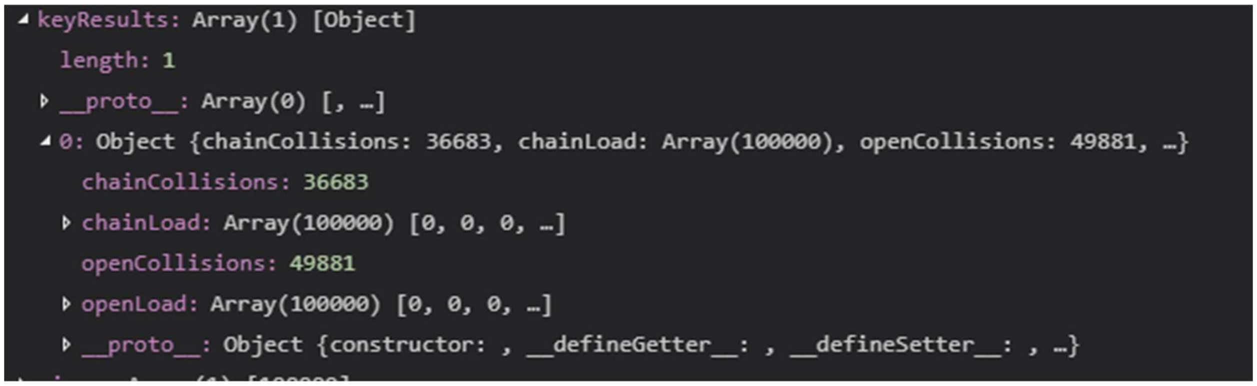
Effect of different hash functions

Different hash function also played a role in collisions vs. load factor. In my trials, the Mid Square results for both collision schemes showed more collisions than the those utilizing the Key Mod Table Size function.

To get a closer look at this effect I lowered my iterations of the experiment to only one trial of table size 100,000. I was curious to see if the Mid Square function would perform better at higher results. I assumed this because with higher numbers, the mid square function would result in a much larger range of unique keys.

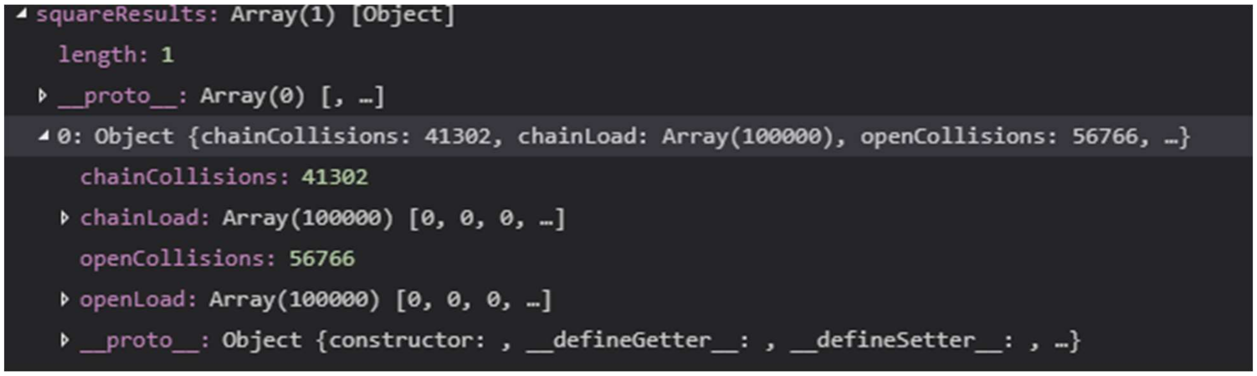
However, my hypothesis was incorrect. Even with a table size of 100,000 the mid square function finished with more collisions than the key mod table size function. Graphing these results would be a heavy load on the machine, so I have instead included some screen caps displaying these results. These were taken off VS Code's debugger after setting a breakpoint on the init() function's return statement.

Figure 1 Key Mod Table Size Results: 36,683 collisions for separate chaining. 49,881 collisions for open addressing.



```
◢ keyResults: Array(1) [Object]
   length: 1
  ▸ __proto__: Array(0) [, …]
  ◢ 0: Object {chainCollisions: 36683, chainLoad: Array(100000), openCollisions: 49881, …}
     chainCollisions: 36683
    ▸ chainLoad: Array(100000) [0, 0, 0, …]
     openCollisions: 49881
    ▸ openLoad: Array(100000) [0, 0, 0, …]
    ▸ __proto__: Object {constructor: , __defineGetter__: , __defineSetter__: , …}
```

Figure 2 Mid Square Results: 41,302 collisions for separate chaining, 56,766 collisions for open addressing

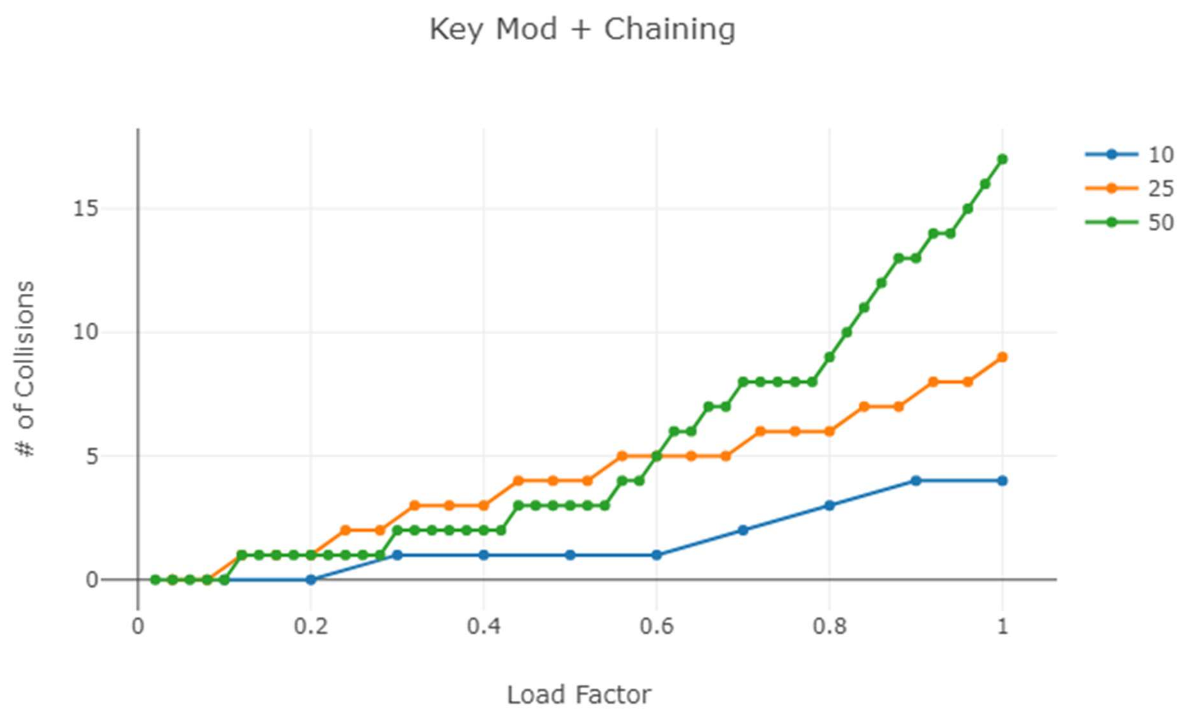


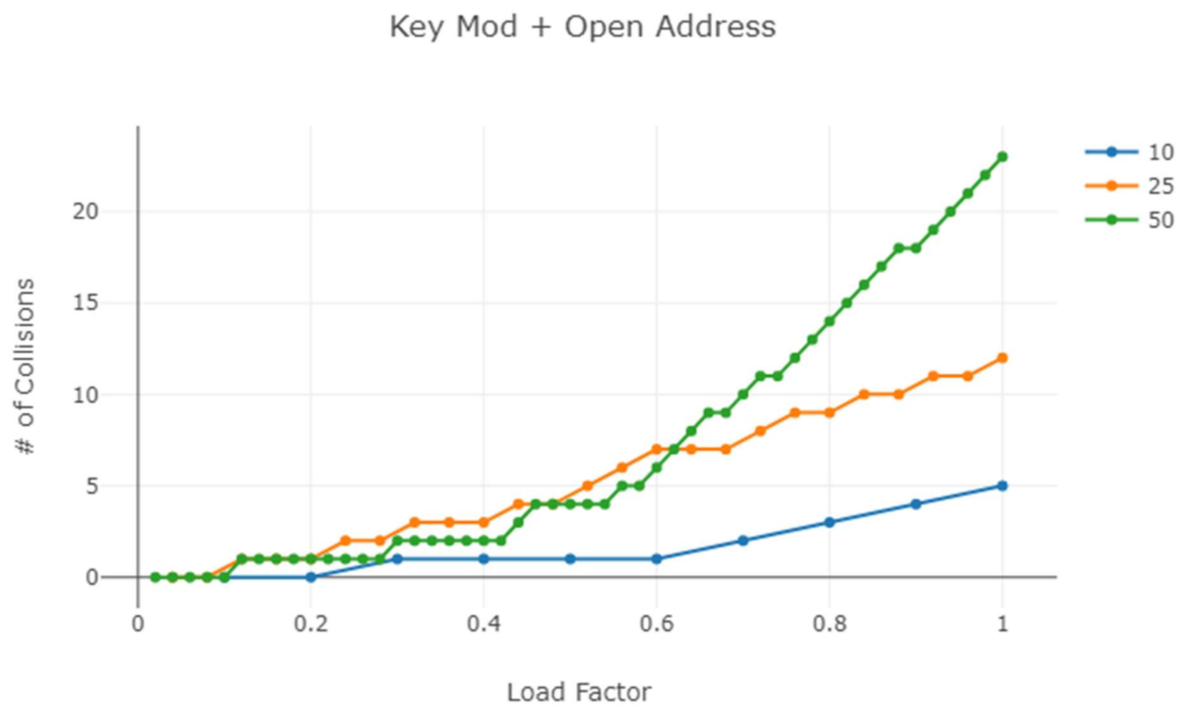
```
◢ squareResults: Array(1) [Object]
   length: 1
  ▸ __proto__: Array(0) [, …]
  ◢ 0: Object {chainCollisions: 41302, chainLoad: Array(100000), openCollisions: 56766, …}
     chainCollisions: 41302
    ▸ chainLoad: Array(100000) [0, 0, 0, …]
     openCollisions: 56766
    ▸ openLoad: Array(100000) [0, 0, 0, …]
    ▸ __proto__: Object {constructor: , __defineGetter__: , __defineSetter__: , …}
```

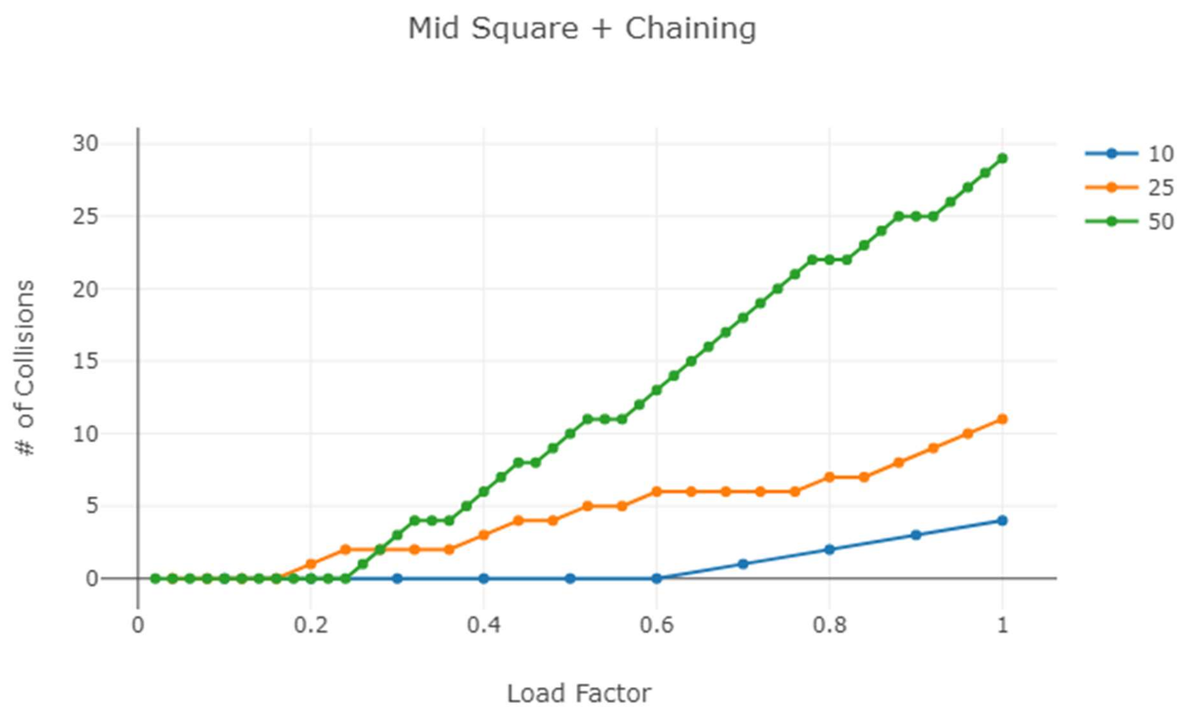
Collectively, my results show that the Key Mod Table Size hashing function utilizing the Separate Chaining collision resolution scheme produces the most efficient hash table. However, this conclusion is bound to the restraints on the project: integer key values with hash table sizes of less than or equal to 100,000.

Plots

Table Sizes: 10,25, and 50






Mid Square + Chaining
# of Collisions
Load Factor
10
25
50
0
5
10
15
20
25
30
0
0.2
0.4
0.6
0.8
1

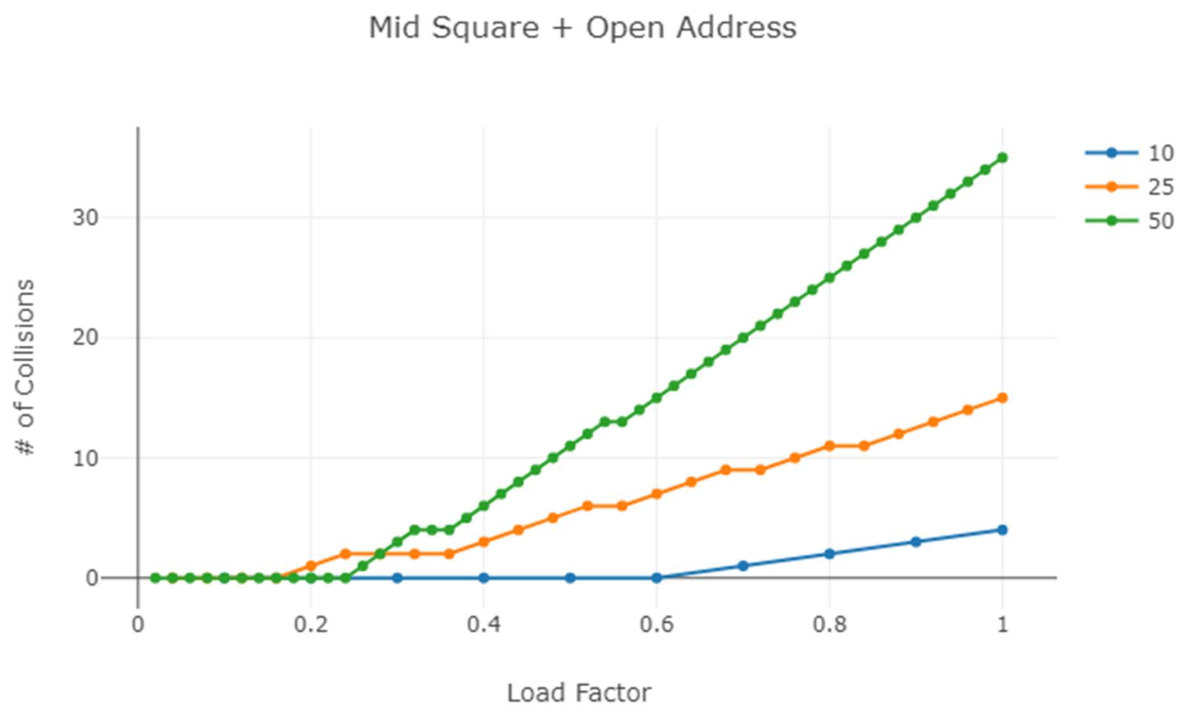
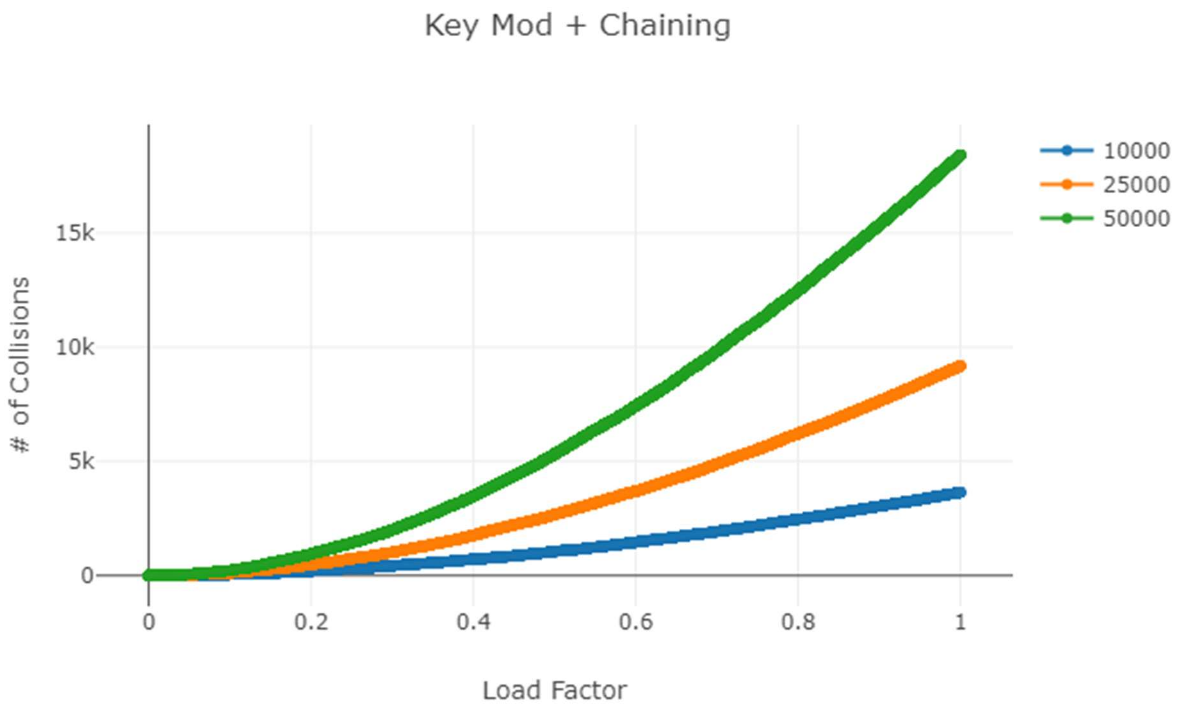
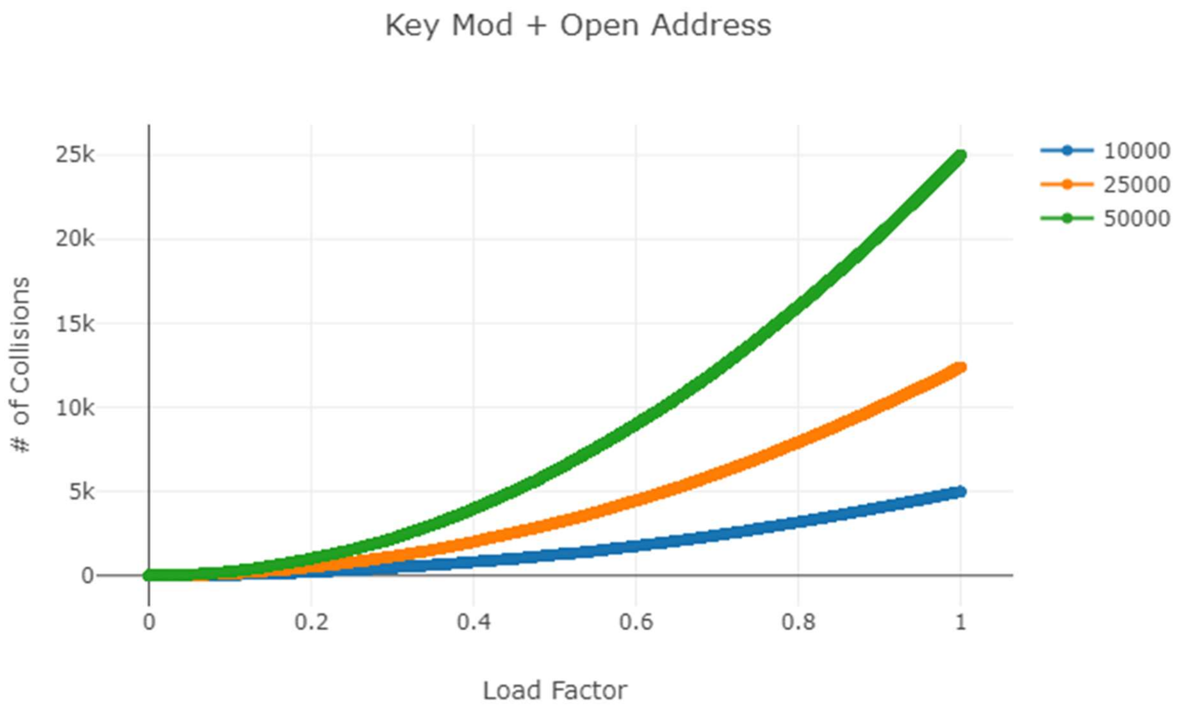
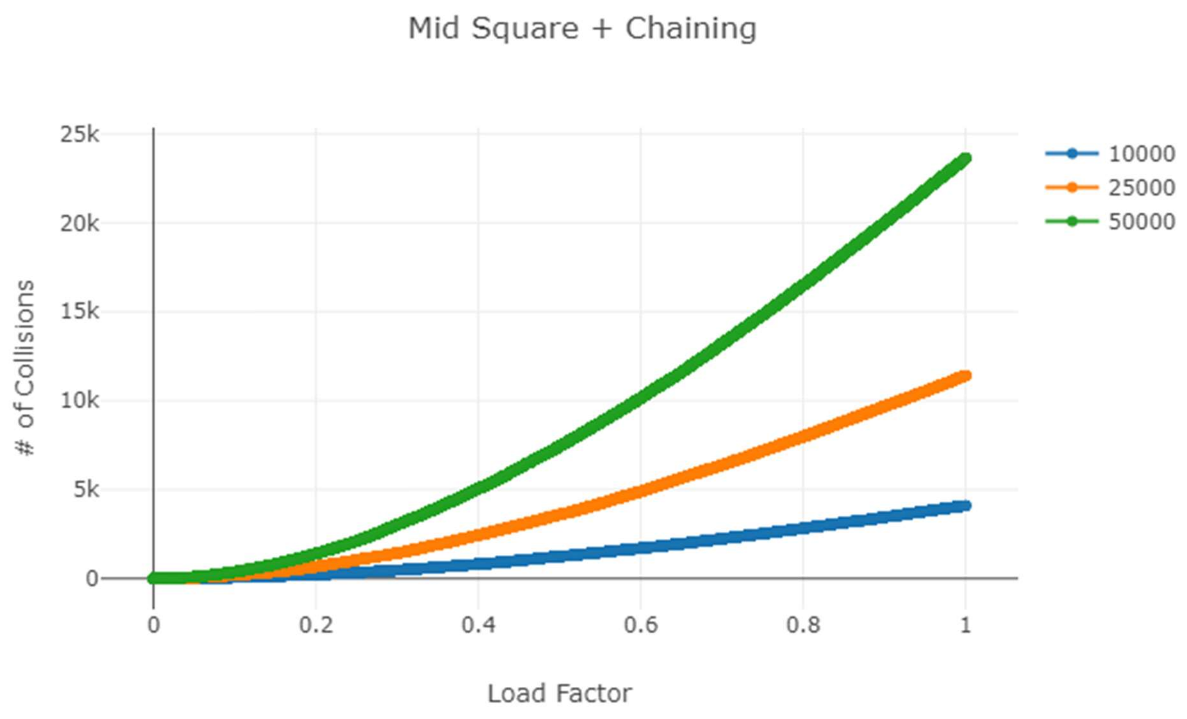
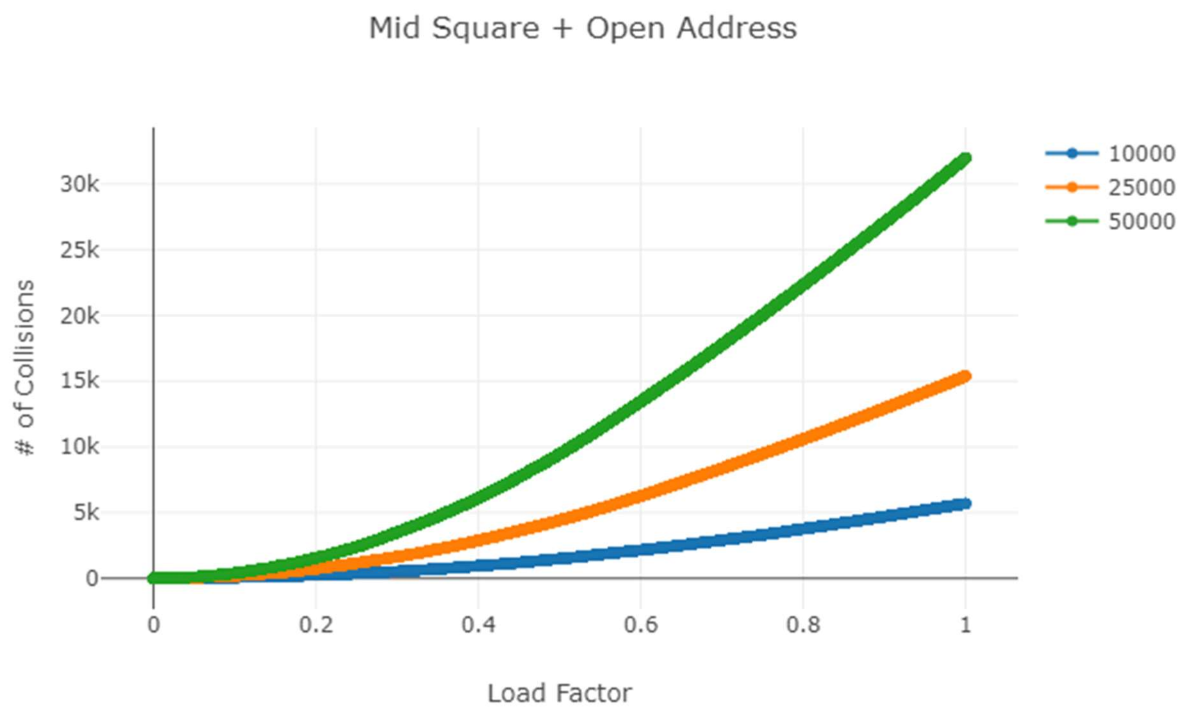

Mid Square + Open Address
# of Collisions
Load Factor
10
25
50
0
10
20
30
0
0.2
0.4
0.6
0.8
1

Table Sizes: 10K, 25K, and 50K






Mid Square + Chaining
# of Collisions
Load Factor
10000
25000
50000
0
5k
10k
15k
20k
25k
0
0.2
0.4
0.6
0.8
1


Mid Square + Open Address
# of Collisions
Load Factor
10000
25000
50000
0
5k
10k
15k
20k
25k
30k
0
0.2
0.4
0.6
0.8
1