**CMSC701 Homework 3: Evaluating two different AMQs**

**Student: Jessica Zhu**

**Date: 10MAY2023**

**Github: https://github.com/jhzsquared/CMSC701\_hw3**

**Task 1 — Empirical evaluation of the bloom filter**

To create K we generate a vector of 31 character length random keys. We then generate K’ by taking a certain percentage (defined by a mix percent value) of keys from the original key set and then generating additional new keys.

Using Rust’s BloomFilter crate, we observe a higher false positive rate than the defined false positive rate of the given bloom filter. It is a bit odd that the false positive rate of 1/(2^8) has a higher on average observed false positive rate, but this may due to be a couple of outliers. We do observe a decrease in observed false positive rate as expected as we increase the parameter. This aligns with the theoretical of false positive rates (Figure 1). It takes on average .1 microseconds to query a key regardless of false positive rate parameter (Figure 2). The size of the bloom filter is consistently 184 bytes.

The most difficult part of implementing this was choosing how to present the results.

A picture containing diagram, screenshot, plot, line

Description automatically generated

Figure 1: Graph of defined false positive rate (1/2^x) vs observed false positive rate for the bloom filter

A picture containing text, screenshot, rectangle, diagram

Description automatically generated

Figure 2: Average time per query for Bloom filter given various defined false positive rate

**Task 2 — Empirical evaluation of a minimal perfect hash**

We used the Boomphf crate’s minimal perfect hash function in this implementation. To check if something was a false positive, we tried to hash it and if it hashed successfully to a value less than the length of the size of the original keys (n), this was a false positive. The observed false positive rate (% of false positive in total vector) is on average only 5-10% less than the number of new K’ values (Figure 3). In other words, only 5-10% of the new K’ values were categorized correctly. As expected, we did not experience any false negatives. This is a significantly higher false positive rate than the bloom filter. However, the MPHF is only allocated 32 bytes, 1/5 of the size of the Bloom filter. I did not expect such poor performance from the minimal perfect hash. Clearly it is minimally perfect.

The most difficult part of this implementation was remembering to check for the hash to be less than the number of items in K as a final check for false positives.

A picture containing screenshot, text, rectangle, diagram

Description automatically generated

Figure 3: Observed false positives given various mix percentages of true keys

**Task 3 — Augment the MPHF with a “fingerprint array”**

We implement the fingerprint array using the ahash crate. We find the MPHF with fingerprint array to be more accurate than the bloom filter by at least .2% --or 25% of the bloom filter FPR across various pos/neg mixes, number of last bits saved, and K’ sizes (Figure 4). This is as expected, as the addition of a good hash provides enough of an identifier to avoid most collisions. It also only required 56 bytes to be allocated (1/3 of the size of the bloom filter) for the MPHF and array combined. However, it takes almost 2x as long as the bloom filter to complete a query. (Figure 5) We do not observe a distinct trend with regard to false positive rate and look up time (Figure 6). Average time is relatively consistent across false positive rates.

The most difficult part of this implementation was finding a new hash function and figuring out how to implement it.

A picture containing screenshot, diagram, text, plot

Description automatically generated

Figure 4: Observed false positive rate by the number last bits (b) saved across various pos/neg mixes.

A picture containing diagram, screenshot, text, rectangle

Description automatically generated

Figure 5: Average time to query one key across methods

A picture containing text, screenshot, diagram

Description automatically generated

Figure 6: Observed false positive rate vs average time to query a key for the MPHF with Fingerprint

**References**

<https://docs.rs/bloom/latest/bloom/>

<https://docs.rs/ahash/0.8.3/ahash/>

<https://stackoverflow.com/questions/54275459/how-do-i-create-a-random-string-by-sampling-from-alphanumeric-characters>

<https://10xgenomics.github.io/rust-boomphf/master/boomphf/index.html#modules>