Approximate Membership Queries

Ryan Van Voorhis

May 11, 2023

Source Code

This project was written in rust. It relies heavily on the libraries probabilistic-collections-rs[3], boomphf[5], and sucds[4] for their implementations of the bloom filter, BBHash minimal perfect hash function, and compact integer vectors respectively. Additionally the serde[2] library in combination with bincode[6] was used to estimate the in memory size of the datastructures. Finally the clap[1] framework was used to create a command line interface for running experiments against the studied data structures. All the source code for this project is available at https://github.com/rjvanvoorhis/approximate-rs.

1 Bloom Filter

1.1 Implementation

The bloom filter implementation was a wrapper around the library probabilistic-collections-rs' BloomFilter struct. It accepts a vector of keys and a target false positive rate and constructs and stores a bloom filter with those parameters. The wrapper only exposes two methods: contains and size_in_bytes. Contains returns true if the result of the contains query on the underlying bloom filter. The size_in_bytes method returns the size of the bloom filter when serialized with serde. This behavior was formalized in two traits, "MembershipSupport" and "KnowsSize", which require the "contains" and "size_in_bytes" methods to be implemented respectively. All the AMQ structs implemented for this project implement these two traits.

1.2 Challenges

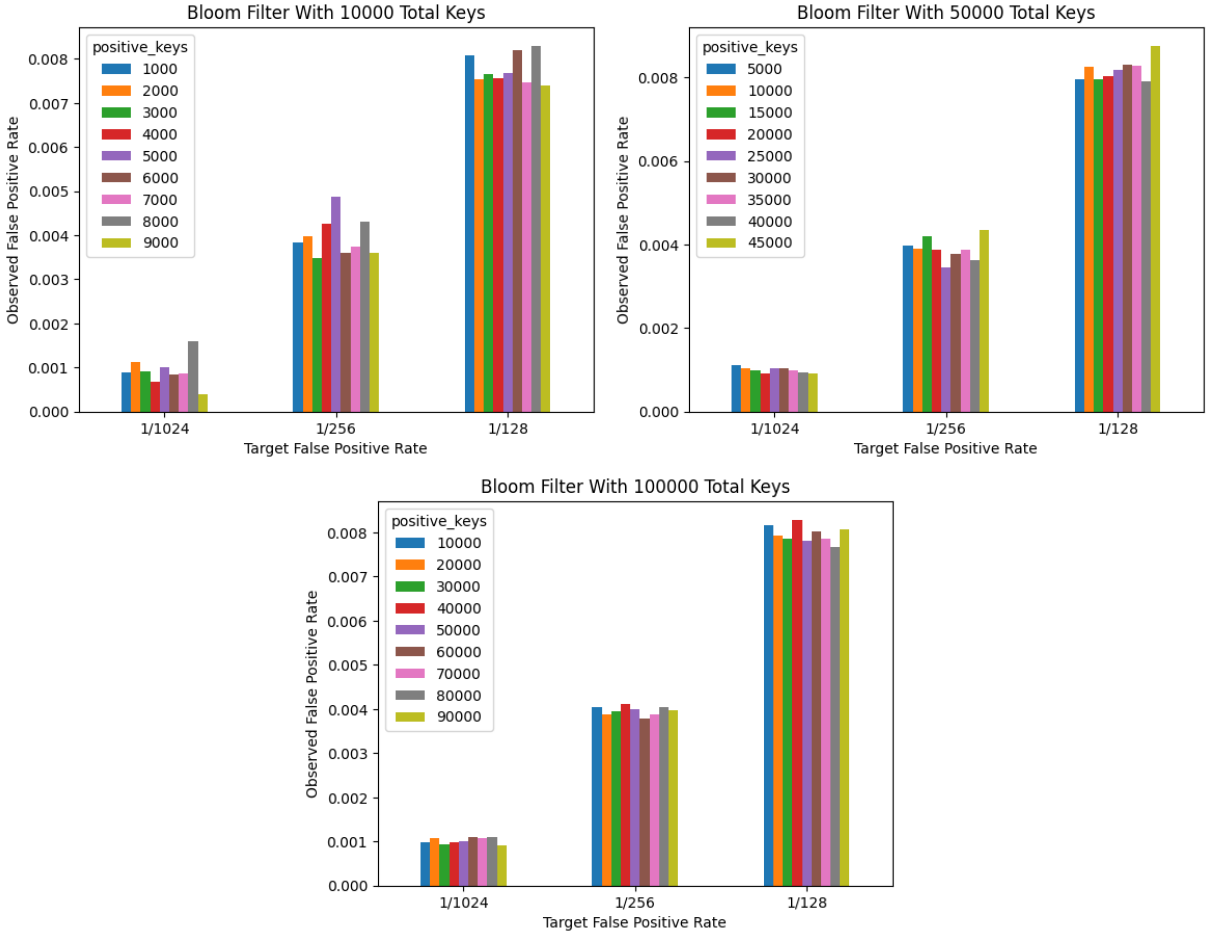
The most difficult part of this task was selecting an existing bloom filter implementation. On crates.io there were nearly 100 crates offering bloom filters, with various level of completeness and quality. I selected probabilistic-collection-rs as it's interfaced supported all the task's requirements. Furthermore the serde feature flag was helpful to complete the size analysis section of the assignment.

1.3 Results

For the experiment bloom filters were constructed with the suggested $1/2^7$, $1/2^8$, and $1/2^{10}$ false positive rates. The experiment used 3 sets of keys, K'. These datasets contained 10,000, 50,000, and 100,000 unique keys respectively. Each K was a string of 30 characters. For each

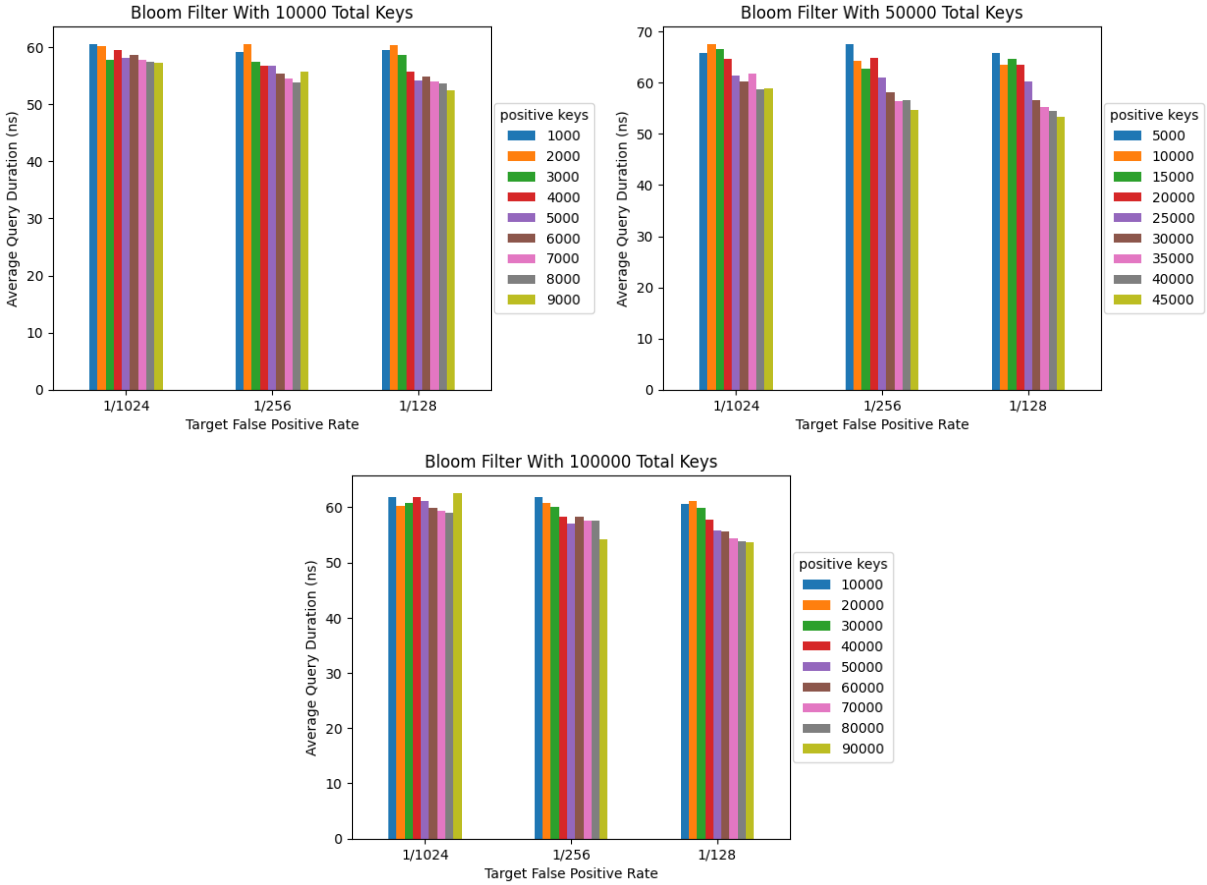
dataset a subset, K, containing 10% - 90% of the keys from K' were selected and used to construct the bloom filter. The observed false positive rate, average query time, and datastructure size were recorded for each combination of false positive rate, dataset, and positive key ratio.

Figure 1: Expected vs observed false positive rates for bloom filters with various sizes and mixtures of K and K' keys



As the total number of keys increased the effect of outliers diminished and the observed rates converged on the expected values.

Figure 2: Average query duration for bloom filters with various sizes and mixtures of K and K' keys



The query time for the bloom filter generally followed the theoretical limits. The queries approached constant time, taking approximately 55-60 ns per query. One aspect that differed from the expected behavior is the way query time was affected by the ratio of positive to negative keys in the query set. When executing a "contains" query the bloom filter can terminate early if if encounters an unset bit since it indicates that key must not exist. Because of this sets with a greater proportion of negative keys might be expected to query faster than one containing mostly positive examples. However, the experiments revealed the opposite, a slight preference for query sets containing mostly positive keys. Due to the simplicity of the wrapper this could be due to either an idiosyncrasy of probabilistic-datastructures implementation of the bloom filter or could be a caching issue in the experiment setup.

2 MPHF

2.1 Implementation

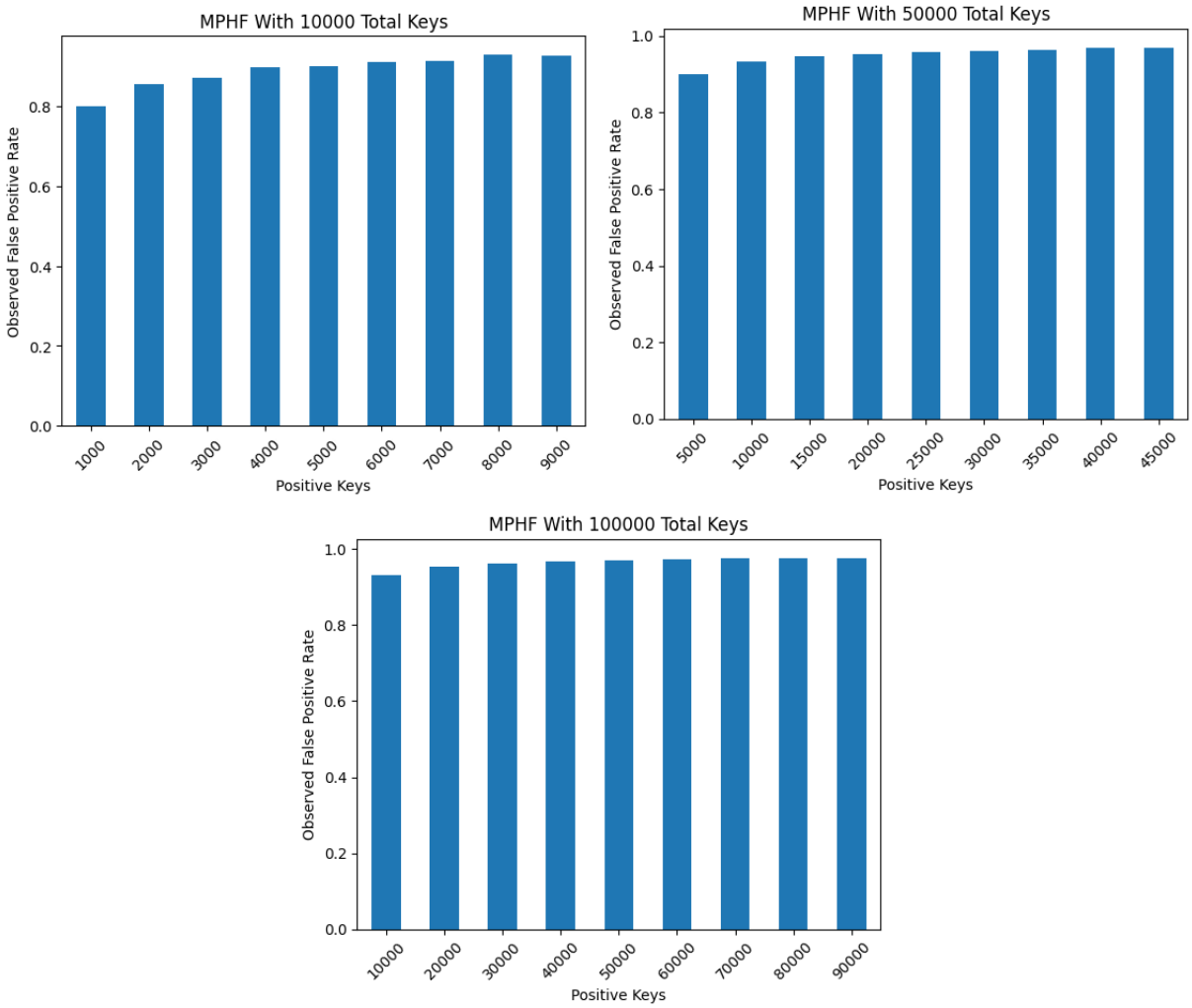
Like the bloom filter, the mphf implementation consisted of a thin wrapper around a third party library. In this case The BBHash Mphf from the boomphf library. The interface was similar to the bloom filter struct, accepting a vector of keys to construct the hash function. The "contains" method for this struct returns true if a given key successfully hashes to a value.

2.2 Challenges

The implementation was straightforward for this section since the construction and query logic for the mphf could be delegated to the underlying 3rd party libraries implementation. Unfortunately the library did not provide public access to the bitvector and other members that comprised the mphf. In order to record the size of the datastructrue the serialized size from serde had to be used to implement the "KnowsSize" trait.

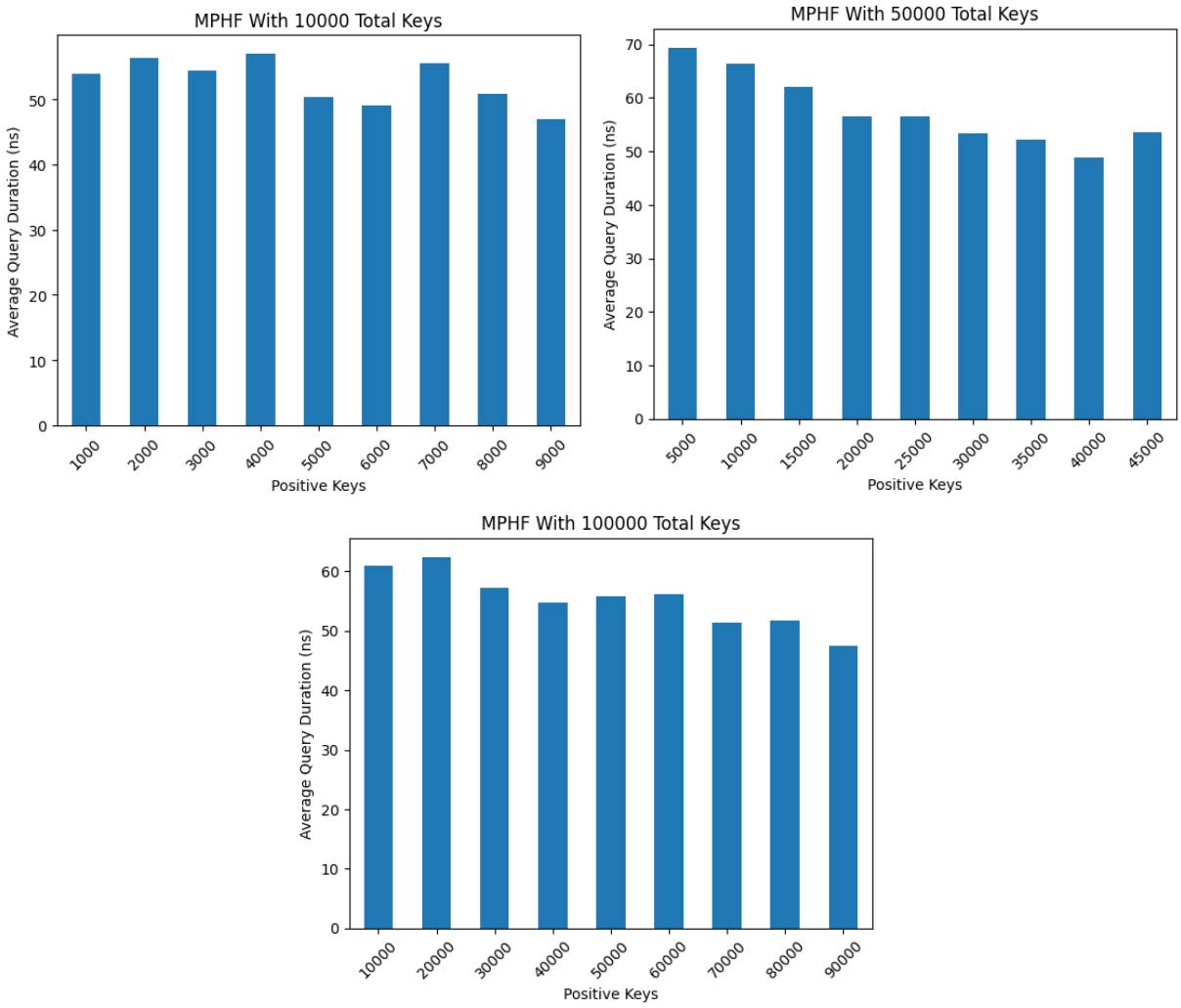
2.3 Results

Figure 3: Observed false positive rates for mphf with various sizes and mixtures of K and K' keys



Unlike the bloom filter the mphf makes no guarantees about the behavior of hashing keys from outside the construction set and has no way to tune the false positive rate. As a result this results in a roughly constant false positive rate. It was somewhat surprising how high the false positive rate is, nearly approaching 1. However with a gamma value near 1 the final bit vectors in the mphf should contain little to no unset bits. It is then most probable that a hash of a given key will collide with at least one set bit and produce a false positive.

Figure 4: Average query duration for mphf with various sizes and mixtures of K and K' keys



The query time for the mphf was approximately constant. As expected, the query time tended to average between 50 and 60 ns without respect to the number of keys used to construct the mphf. Especially in the dataset with 50,000 total keys there appeared to be a minor bias towards query sets containing a large proportion of positive keys. The underlying implementation includes an early termination path in the try_hash method only for positive keys. This could explain the resulting trend of shorter query times for mostly positive keys.

Figure 5: Total memory usage for mphf with various sizes and mixtures of K and K' keys

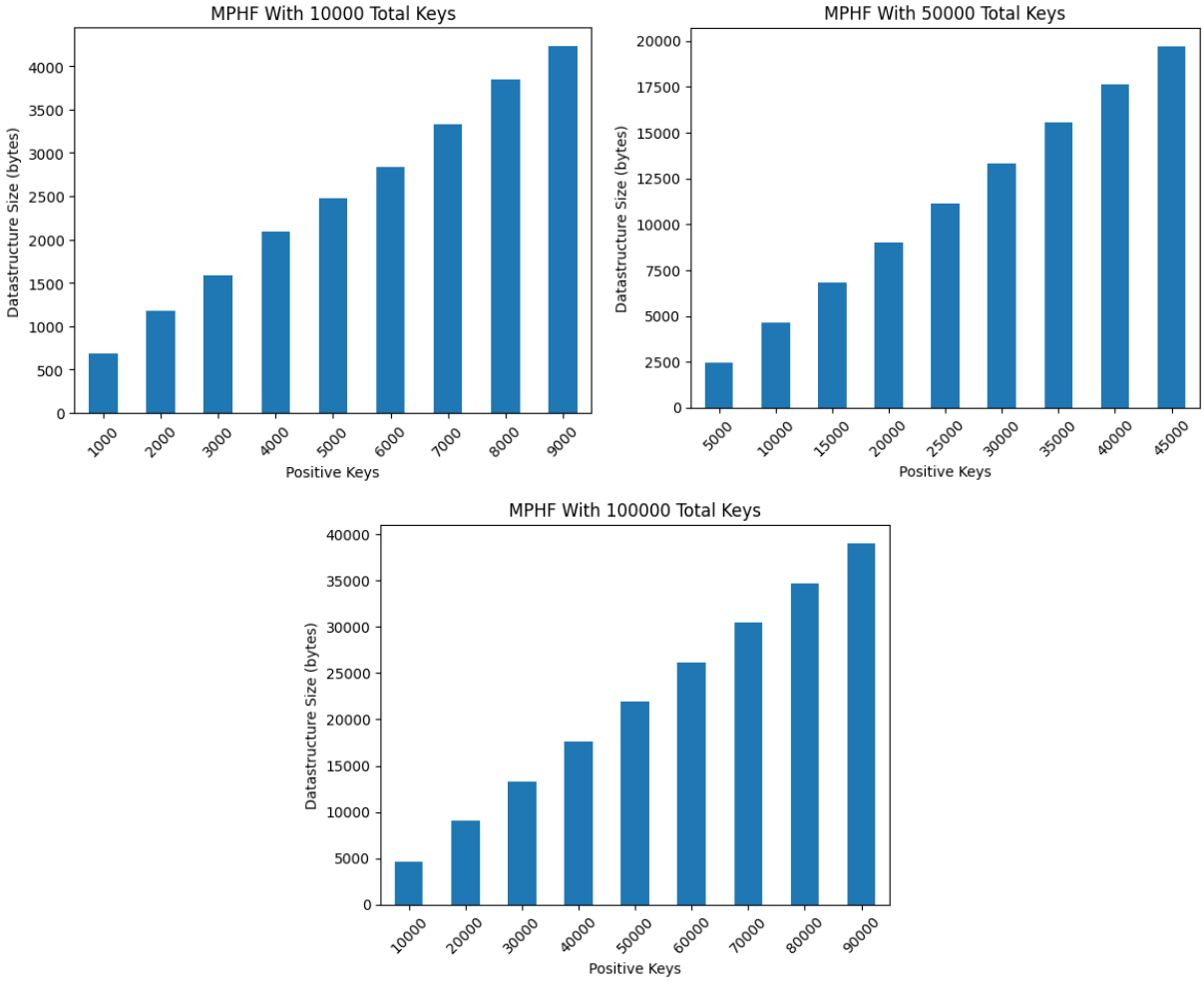
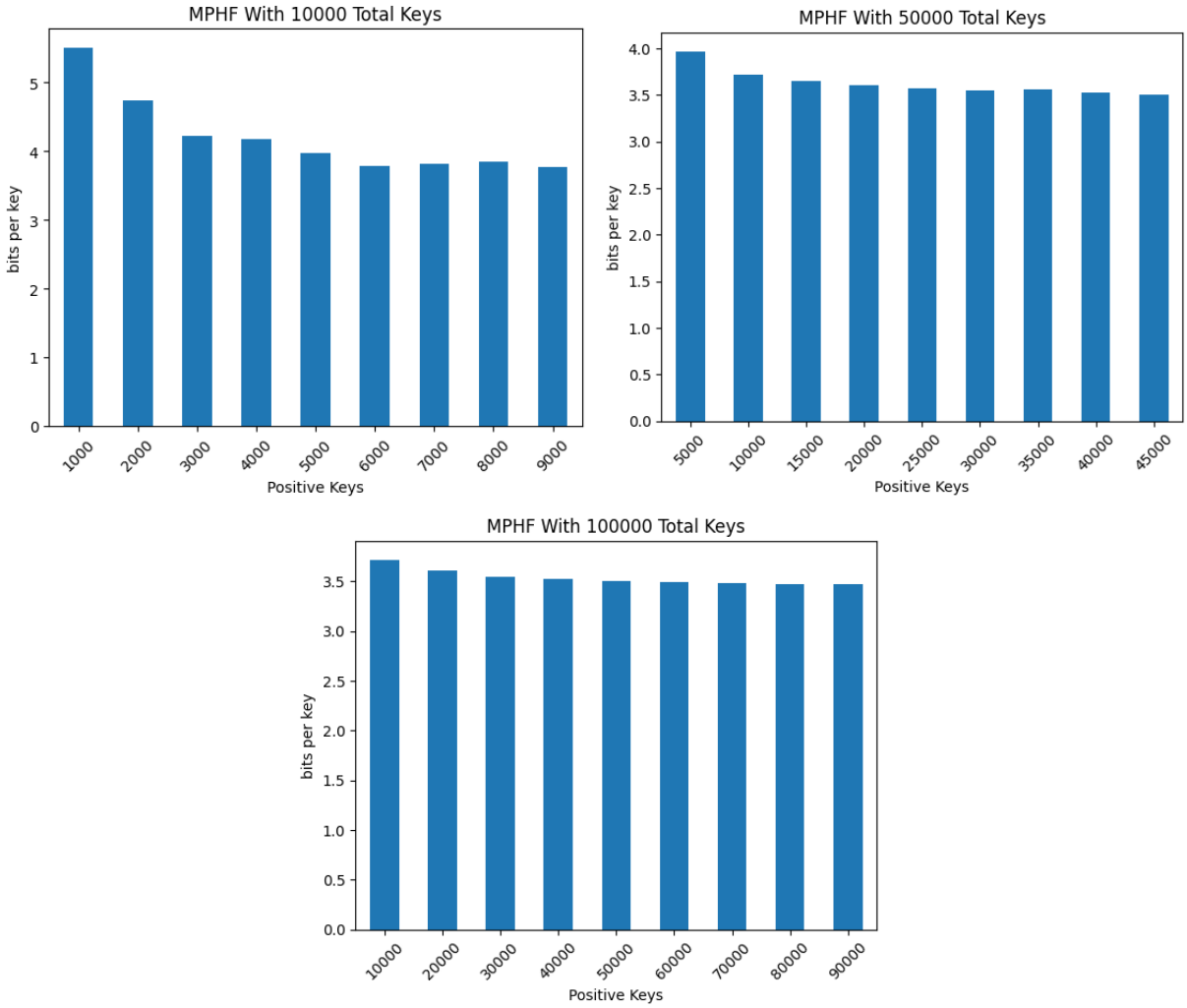


Figure 6: Bits per key for mphf with various sizes and mixtures of K and K' keys



The memory usage of the datastructure follows the theoretical expectations. Memory usage increases linearly with the number of keys used to construct the mphf. The overall bits per key was approximately 3.5 which follows the 3 expected bits per key of BBHash. Only in the degenerative case where the number of keys in the construction set is small (less than 2000 keys, for example) this implementation of the mphf degrades in performance and can exceed as much as 5.5 bits per key.

3 Fingerprint Array

3.1 Implementation

The basis for the fingerprint array was a wrapper around the bloomphf Mphf. Additionally the struct contained an integer vector that would store the fingerprint of each key. To construct a fingerprint the user submits a set of unique keys, K, and specifies the width, w, for each key's fingerprint. The mphf is constructed from the set K and the integer vector is initialized with length equal to |K| where each element is represented by w bits. For each element in K, rust's default hasher is used to calculate a secondary hash value for the key. The last w bits of the hash value are calculated by performing a bitwise AND with a bitmask. The result is

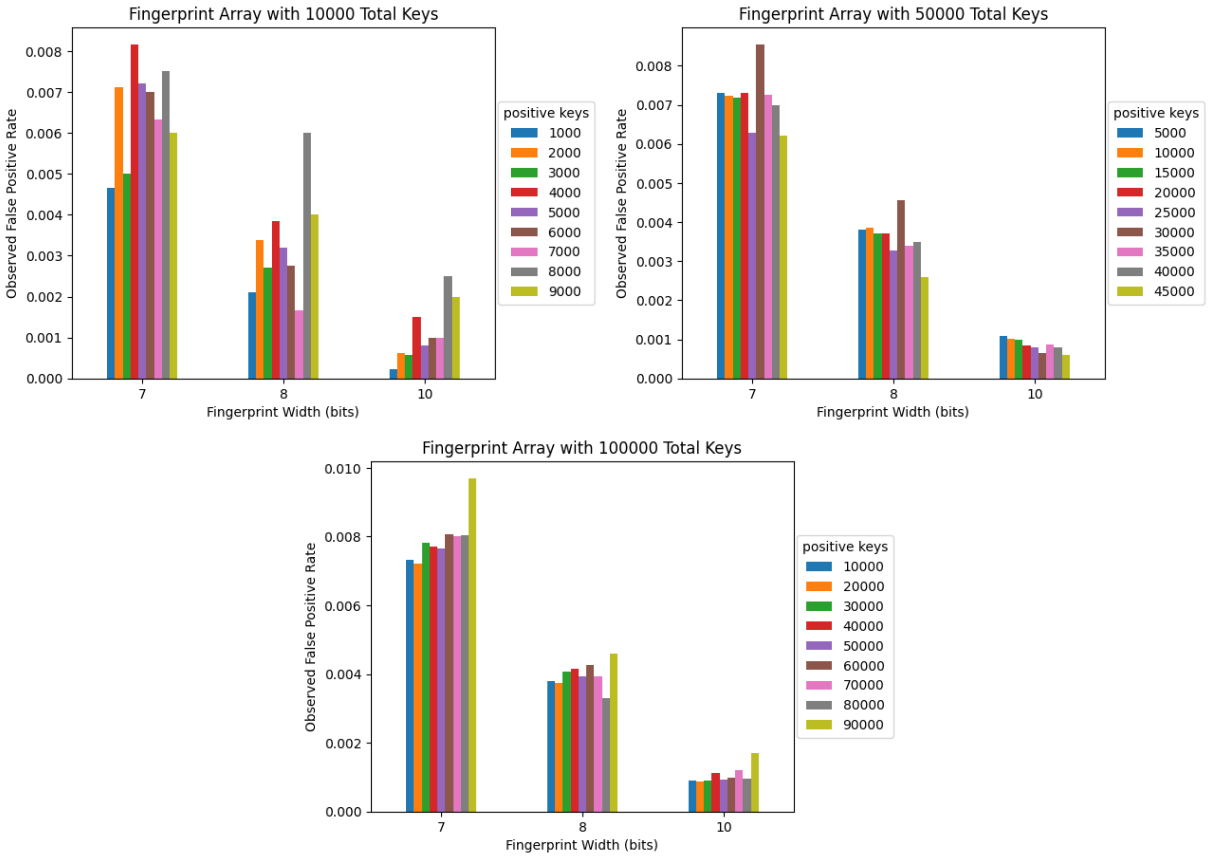
saved in the integer vector at the index equal to the key's hash value in the mphf. To query the datastructure the hash value in the mphf is first calculated. The fingerprint of the query key is then calculated using the same secondary hash function and bitmask method from array construction. The contains method returns true if the newly calculated fingerprint matches the fingerprint saved at the corresponding index in the integer vector.

3.2 Challenges

This task was the most involved of the three. The most difficult part was extracting the last bits of each keys hash which was required to construct the integer vector of fingerprint bits. I also considered which secondary hash function to use to produce the fingerprint keys. Rust's DefaultHasher seemed sufficiently random to approximately half the false positive rate with each additional fingerprint bit. However, a future enhancement I would like to make is to allow the fingerprint array constructor to accept an arbitrary hasher.

3.3 Results

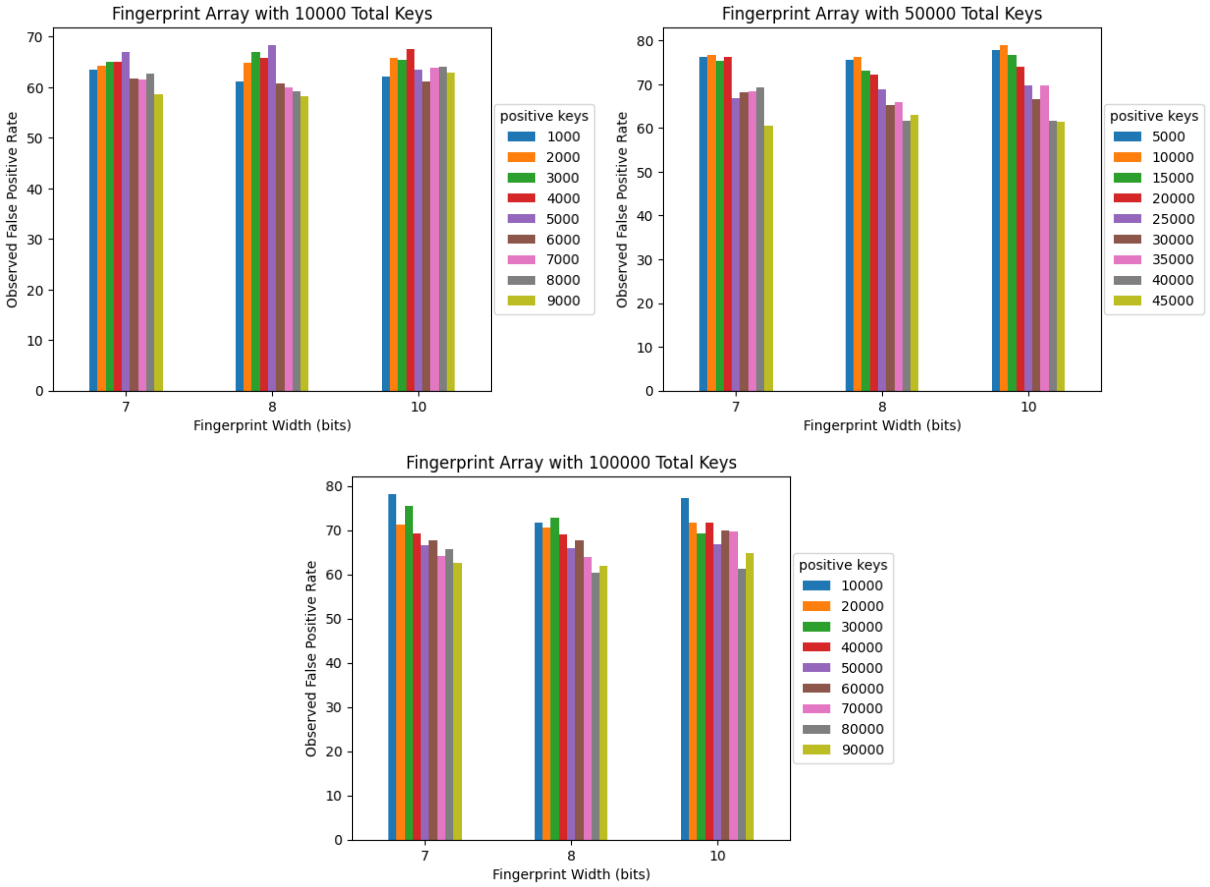
Figure 7: Observed false positive rates for fingerprint array with various sizes and mixtures of K and K' keys



The low absolute number of false positive rate in the smallest dataset produced results with high variance. However, in general the effect of altering the fingerprint width on the observed false positive rate followed theory. With each successive bit added to the fingerprint the false positive rate was approximately halved. Because the baseline false positive rate of the mphf

was nearly 1 the expected false positive rates for the fingerprint array approaches $\frac{1}{2^w}$ where w is the fingerprint width in bits.

Figure 8: Average query duration for fingerprint array with various sizes and mixtures of K and K' keys



The average query duration was similar to that of the of the bloom filter at around 60-70 ns per query. Because the datastructure was built on top of the bbhash mphf it exhibits the same trend in preferring query sets with mostly positive keys. These keys will have a greater chance of colliding with a set bit and terminating the try_hash loop early. Apart from this trend the query time was not affected by the total number of keys. Additionally adjusting the fingerprint width and therefore the false positive rate had no effect on query time. No matter the width the fingerprint is read from the int vector as a usize and the comparison between the query key and the saved fingerprint are done in constant time.

Figure 9: Fingerprint array total memory usage for various sizes and mixtures of K and K' keys

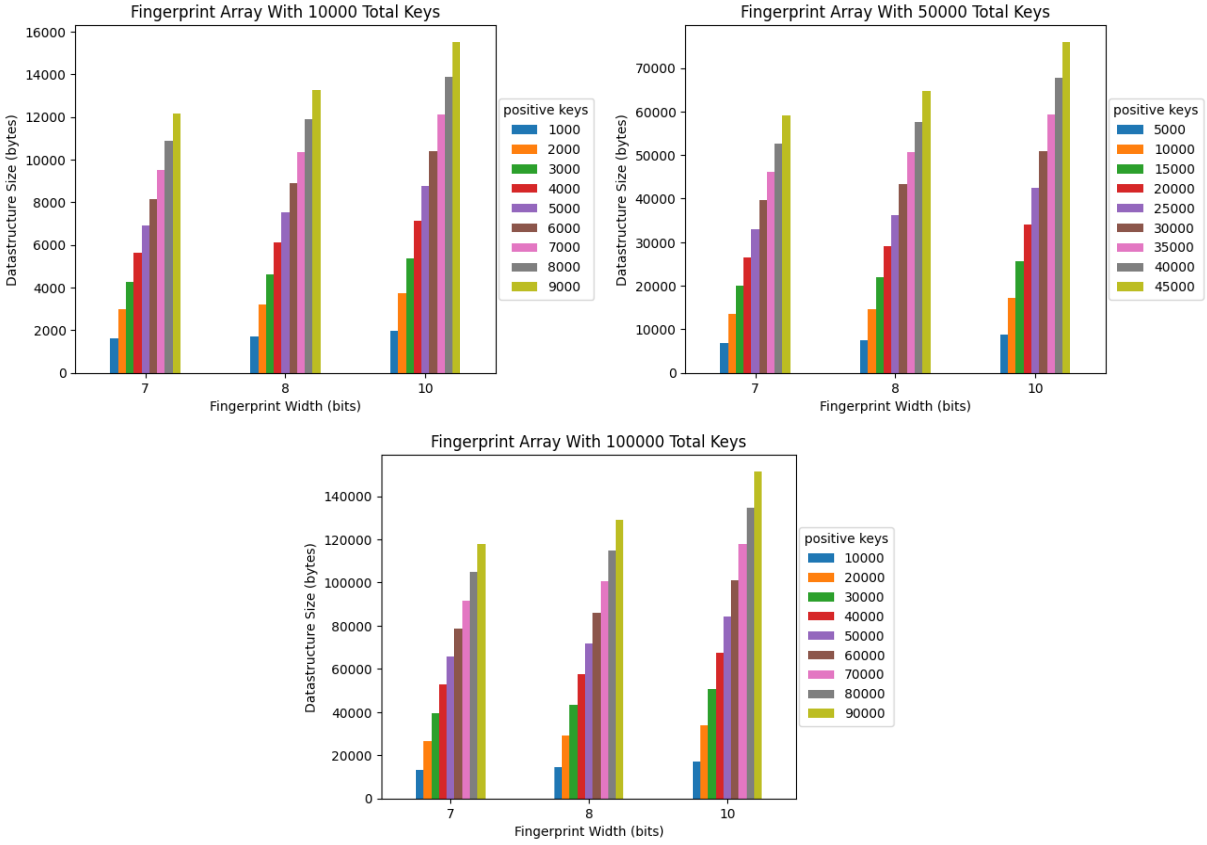
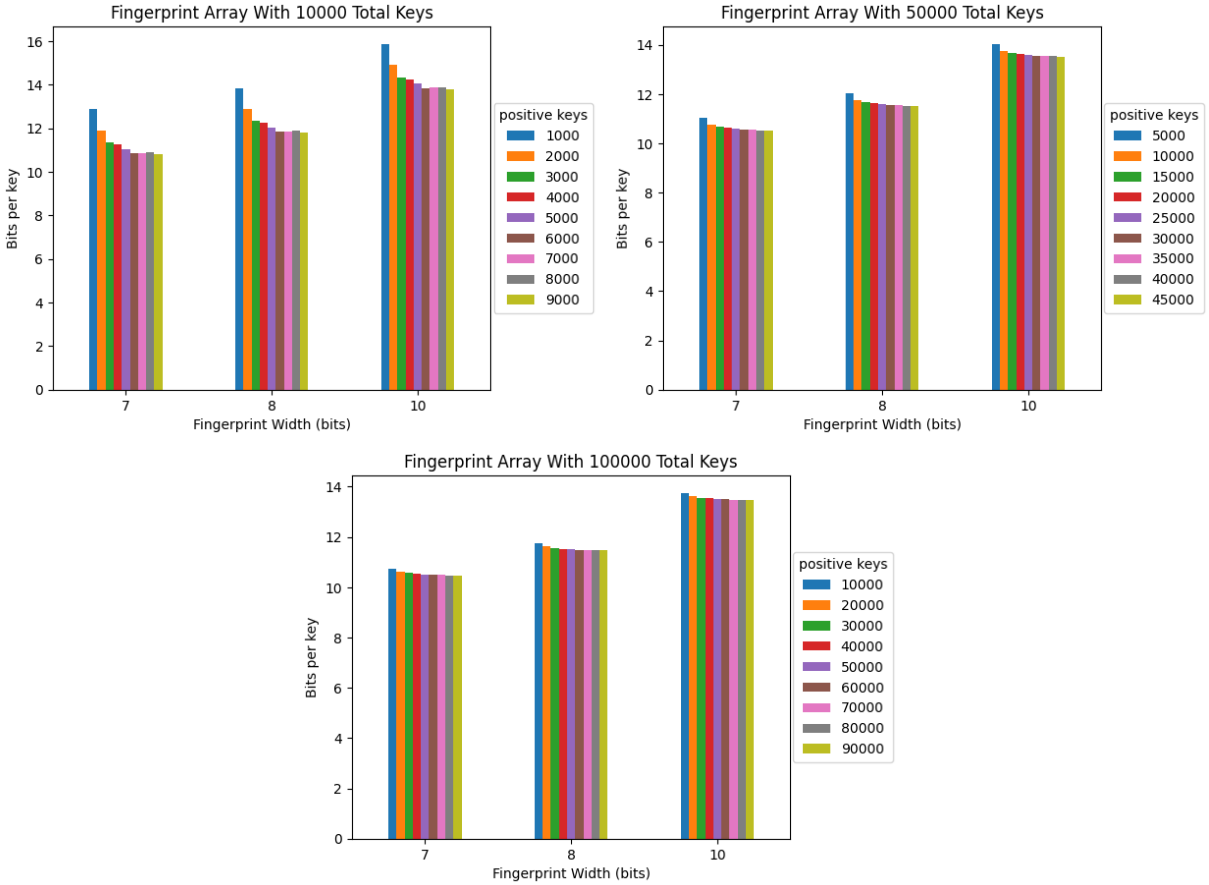


Figure 10: Fingerprint array bits per key for various sizes and mixtures of K and K' keys



The memory usage of the fingerprint array is in line with the theoretical limits. For the smallest dataset with 10,000 total keys the bits per keys are somewhat higher than expected. However this is due to the limitations of the underlying minimal perfect hash function. For each key added to the fingerprint array 3 bits are required for the mphf and w bits are required for integer vector where w is the width the fingerprint array was initialized with.

As identified in task 2, the mphf used as much as 5.5 bits per key for small key sets which skew the overall results.

References

[1] Clap. `https://github.com/clap-rs`, 2015.

[2] Tolnay David and Tryzelaar Erick. serde. `https://github.com/serde-rs/serde`, 2017.

[3] Tolnay David and Tryzelaar Erick. probabilistic-collections. `https://gitlab.com/jeffrey-xiao/probabilistic-collections-rs`, 2018.

[4] Shunsuke Kanda. sucds. `https://github.com/kampersanda/sucds`, 2018.

[5] Patrick Marks. boomphf. `https://github.com/10XGenomics/rust-boomphf`, 2021.

[6] Victor Koenders Ty Overby and Zoey Riordan. bincode. `https://github.com/bincode-org/bincode`, 2021.