University of Maryland

COMPUTER SCIENCE AND ENGINEERING

Algorithms, Data Structures and Inference for High-Throughput Genomics

CMSC 858D

Homework 2

Caring for your cache with AMQs

November 28, 2019

STUDENT ID :
Kazi Tasnim Zinat(116760904)

Introduction

A pproximate membership query data structures are used widely where it is important to have perfect specificity or recall so the true positive rate must be 1, but the precision can be compromised to some extent, so we can accommodate some false positive values. Bloom Filter is one example of such kind of data structure. For this assignment, we had to implement two variants of the Bloom Filters- The basic bloom filter and the blocked bloom filter. This write up contains the implementation procedure of each of the implementation along with the challenges faced, and some evaluations performed on generated dataset.

Languages and Libraries used: Both the tasks are implemented C++, the key files are generated using PYTHON, and executables were developed via SHELL-SCRIPTING. The hash function used is the 32-bit variant of MurmurHash3, implemented in $C++$ available here

Dataset Generation: The generated corpus is all k-mers of length 12. From there, I sampled $n + 1000$ number of elements. Of them the first n was used to build the Bloom FIlter, and 1000 was sampled from that to generate $1^{st} set$ of queries where everything in the query file is present in the BloomFilter. The thousand before the last 500 was used t generate the 2^{nd} set of queries where 50% of the items are present in the Bloom Filter. Finally, the last thousand was not present n the key set and was used to produce the 3^{rd} dataset, where nothing is present in the original set.

Task 1 — Basic Bloom Filter

Implementation

This data structure has the following member functions.

- n (size of universe)
- fpr(Expected False Positive Rate)
- m (optimal size of the bloom-filter)
- k (number of bits associated with each input key)
- bit-vector (The main structure containing bit values)

The data structure has two constructors- the first one takes n and fpr as input, and computes m and k as given below. A $vector < bool >$ of size m representing the bit-vector is created, and is initialized with all $false$ values.

$$\text{m} = \left\lceil -\frac{nlnfpr}{(ln2)^2} \right\rceil$$
$$\text{k} = \left\lceil \frac{m}{n} ln2 \right\rceil$$

The second constructor reads an input fie and assigns values to all the variables and populates the bit vector from the file.

Insertion: For each key in the key_file MurMurHash3 is called k times to generate the hash values which is mapped to an index of the bit-vector by modulus operation with m. The derived index is set to *true*

Query: For each key in the key_file MurMurHash3 is called k times to generate the index of the bit-vector by modulus operation with m. If all bits are *true* then the key is assumed to be present in the set.

Most difficult part

The trickiest part of Basic Bloom Filter was determining which Hash function to use and how to integrate in the code.. One problem which I spent a significant amount of time on was figuring how to use string as the `MurMurHash3` key. I had to convert the string object to a `c_string`. Also, instead of passing the string size, I mistakenly passed the size of the pointer to the string as second parameter. That took some time to debug.

Plots

Figure 1 plots the execution time for Bloom Filter and Figure 2 plots the empirical false positive rate.

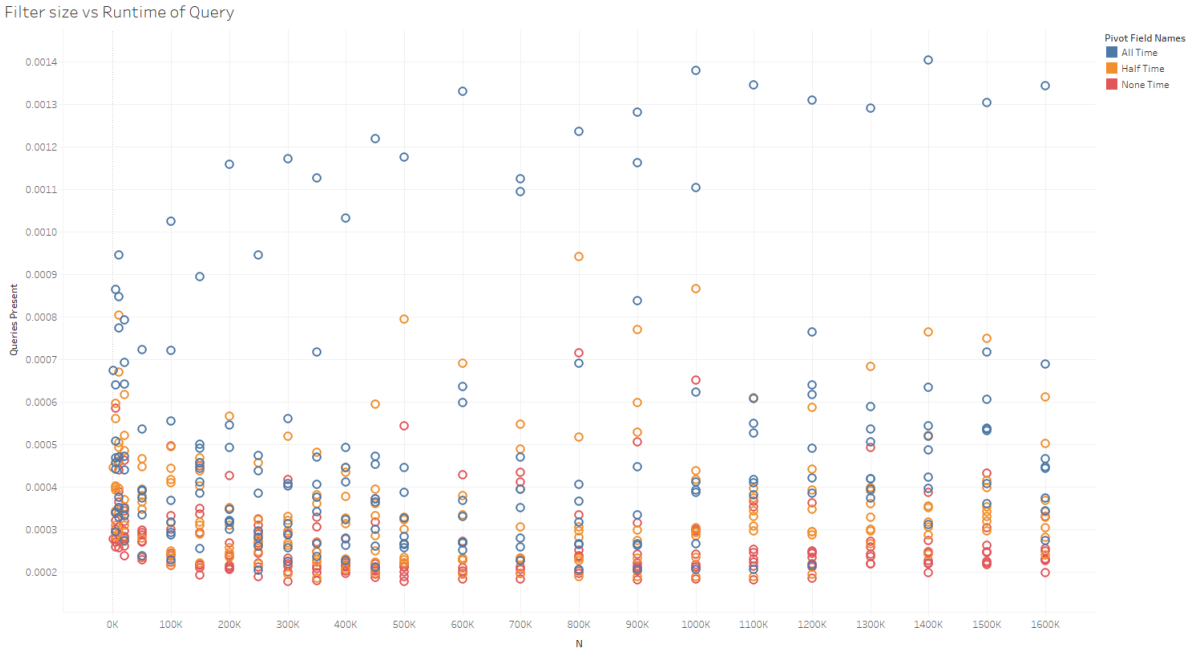

Figure 1: For Basic Bloom Filter, this plot shows the average run time for the three set of queries for different input sizes and False positive rate. Color Blue represents the set where all queries are present in the original set, Orange and Red represents sets where half and none of the queries are present in the actual dataset. We can see that as N increases the queries take longer to execute. Also in every case half and none cases take less time than all- which is rational because once a bit corresponding to a query is false we immediately know this is not present in the dataset- providing us the opportunity to early exit, and saving CPU cycle

Figure 2: Here Empirical False Positive rate for various N is plotted against Expected False positive rate.The Red gradient represents query set where nothing is present in the original dataset. The Blue gradient represents query set where 50% is present in the original dataset. Darker shades represent higher values of N. We can observe that with some variation, empirical false rates remain close to the original value, and for any fpr, usually the test cases with high N have higher empirical false positive rate. A subject of speculation might be the increase of variance in empirical fpr for greater values of expected False positive rate.

Task 2 — Blocked Bloom Filter

In this variant the bloom filter is partitioned into several blocks and each input key will map to any one of the blocks.

Implementation

This data structure has the same member functions as the basic Bloom Filter, with two additional ones.

- s (Number of Blocks)
- cache_line_size (Size of each block)

Like Basic Bloom filter, the data structure also has two constructors- in addition to computing m and k as referred in Task 1, the first constructor acquires cache line size from the system with the command `_SC_LEVEL1_DCACHE_LINESIZE` which gives the L1 cache sizes in bytes. Then number of blocks is calculated by taking the ceiling value of m divided by $cache\_line\_size$.

The second constructor reads an input fie and assigns values to all the variables and populates the bit vector from the file.

Insertion: The difference with basic Bloom FIlter is that the first hash value returned by MurMurHash3 determines which block this key will map to after modulus operation by s. The next $k-1$ values from MurMurHash3 maps to an index of the bit-vector within that block number with modulo m . The derived index is set to *true*

Query: For each key in the key_file MurMurHash3 is called k times. The first call $mod \quad s$ gives block number. The subsequent calls generate in-block index by modulus operation with m. If all bits are *true* then the key is assumed to be present in the set.

Most difficult part

m might not always be perfectly divisible by $cachelinesize$. In that case the last block will have fewer number of bits than the other blocks. I overlooked this corner case during initial phase of implementation. This resulted in `Core Dumped Segmentation Fault` and required rigorous debugging. To solve the problem, I adopted a simple but less optimized method- altering the bloom filter size to $s * cachelinesize$

Plots

Figure 3 represents the the execution time and Figure 4 plots the empirical false positive rate for Blocked Bloom Filter.

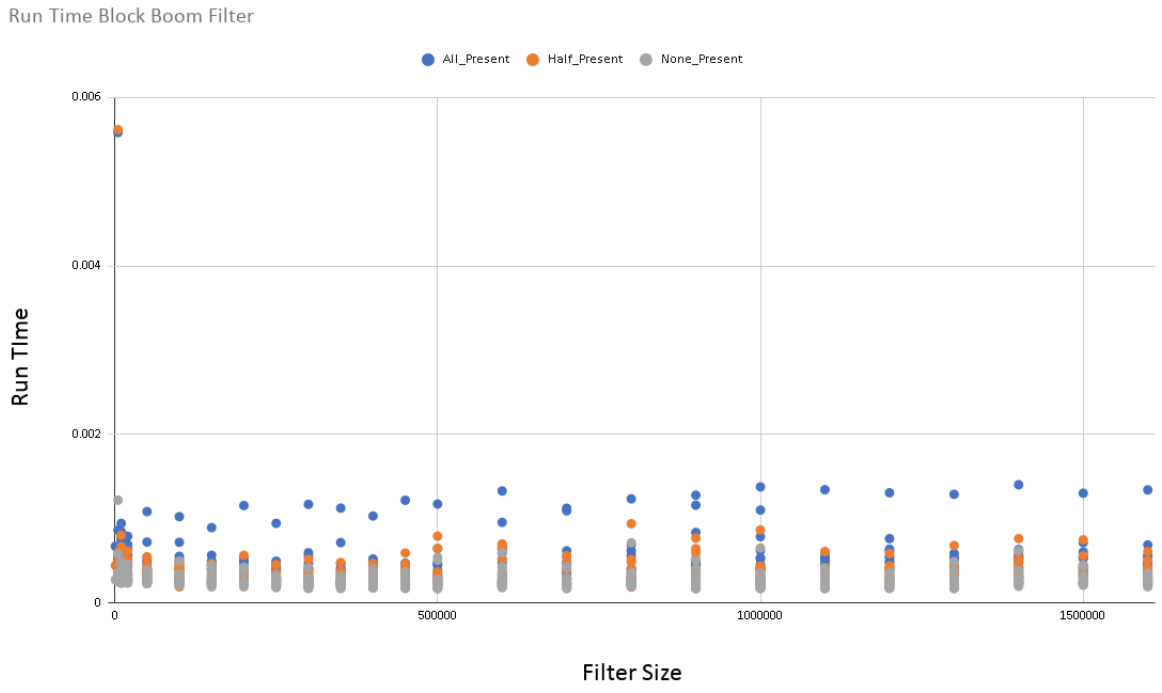

Figure 3: For Block Bloom Filter, this plot shows the average run time for the three set of queries for different input sizes and False positive rate. Color Blue represents the set where all queries are present in the original set, Orange and gray represents sets where half and none of the queries are present in the actual dataset in that order. The most interesting point of this figure is the two orange and blue dots at top left corner, showing high execution time for a small N. This can be interpreted as the first query, with cache misses, and after that as the input set is cache aligned we observe decreased execution time due to cache hit. The average execution time for the all_present set is highest in all cases as there is no early exit, but unlike basic Bloom Filter, even for large universe size the average execution time does not change much.

Figure 4: Here Empirical False Positive rate for various N is plotted against Expected False positive rate for Blocked Bloom Filter.The Red gradient represents query set where nothing is present in the original dataset. The Blue gradient represents query set where 50% is present in the original dataset. Darker shades represent higher values of N. We can observe that with some variation, empirical false rates remain close to the original value, and for any fpr, usually the test cases with high N have higher empirical false positive rate.

Comparing The Results

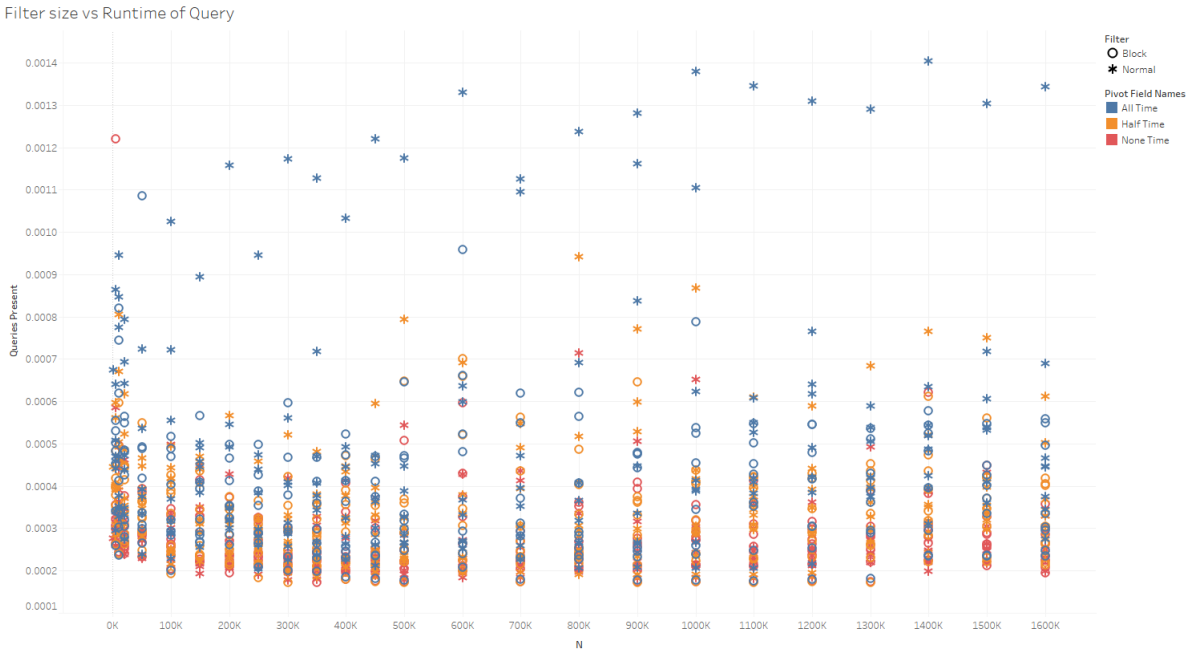

Figure 5: This plot shows the average run time for the three set of queries for different input sizes and False positive rate for both basic(Normal) and Blocked Bloom Filter. Here, marks from Block Bloom Filters are of circle shape, marks from normal bloom filters are star. Color Blue represents the set where all queries are present in the original set, Orange and red represents sets where half and none of the queries are present in the actual dataset in that order. In all cases queries from All_present set of basic Bloom Filter has maximum execution time

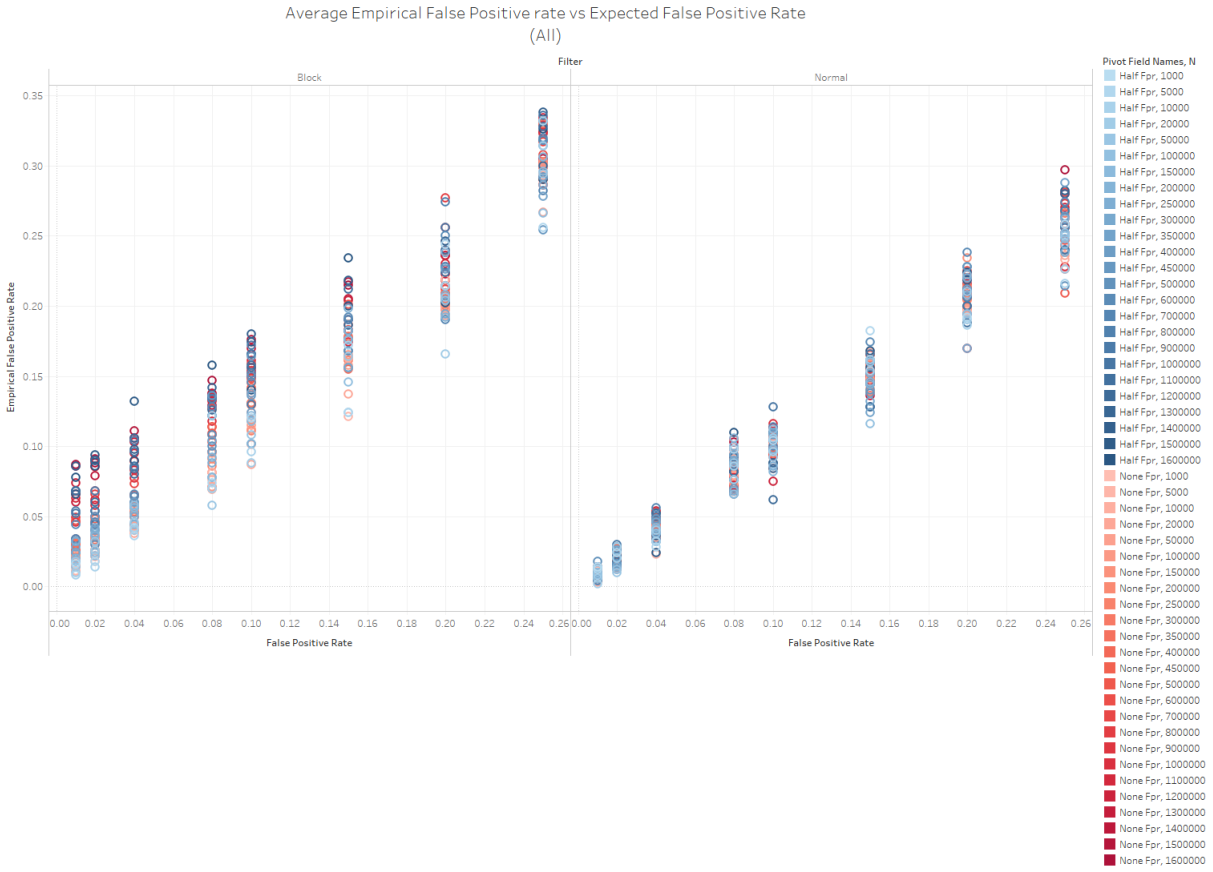

Figure 6: Here Empirical False Positive rate for various N is plotted against Expected False positive rate . The left subplot contains results from Block and the right subplot contains results from Basic (Normal) Bloom filter. The Red gradient represents query set where nothing is present in the original dataset. The Blue gradient represents query set where 50% is present in the original dataset. Darker shades represent higher values of N. We can observe that Block Bloom Filter has higher empirical fpr than the normal one

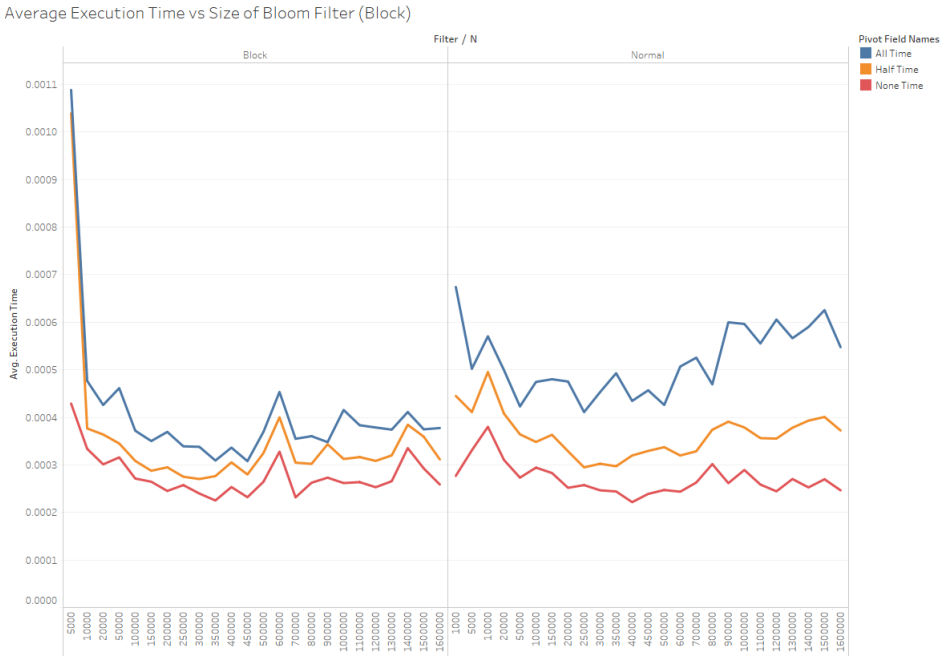

Figure 7: Here average run time for the three set of queries is plotted against various N. The left subplot contains results from Block and the right subplot contains results from Basic (Normal) Bloom filter. The Red gradient represents query set where nothing is present in the original dataset. TColor Blue represents the set where all queries are present in the original set, Orange and red represents sets where half and none of the queries are present in the actual dataset in that order. We can observe that the blue line is on top for both variant, and for larger N, execution time of Basic block bloom filter query increases.

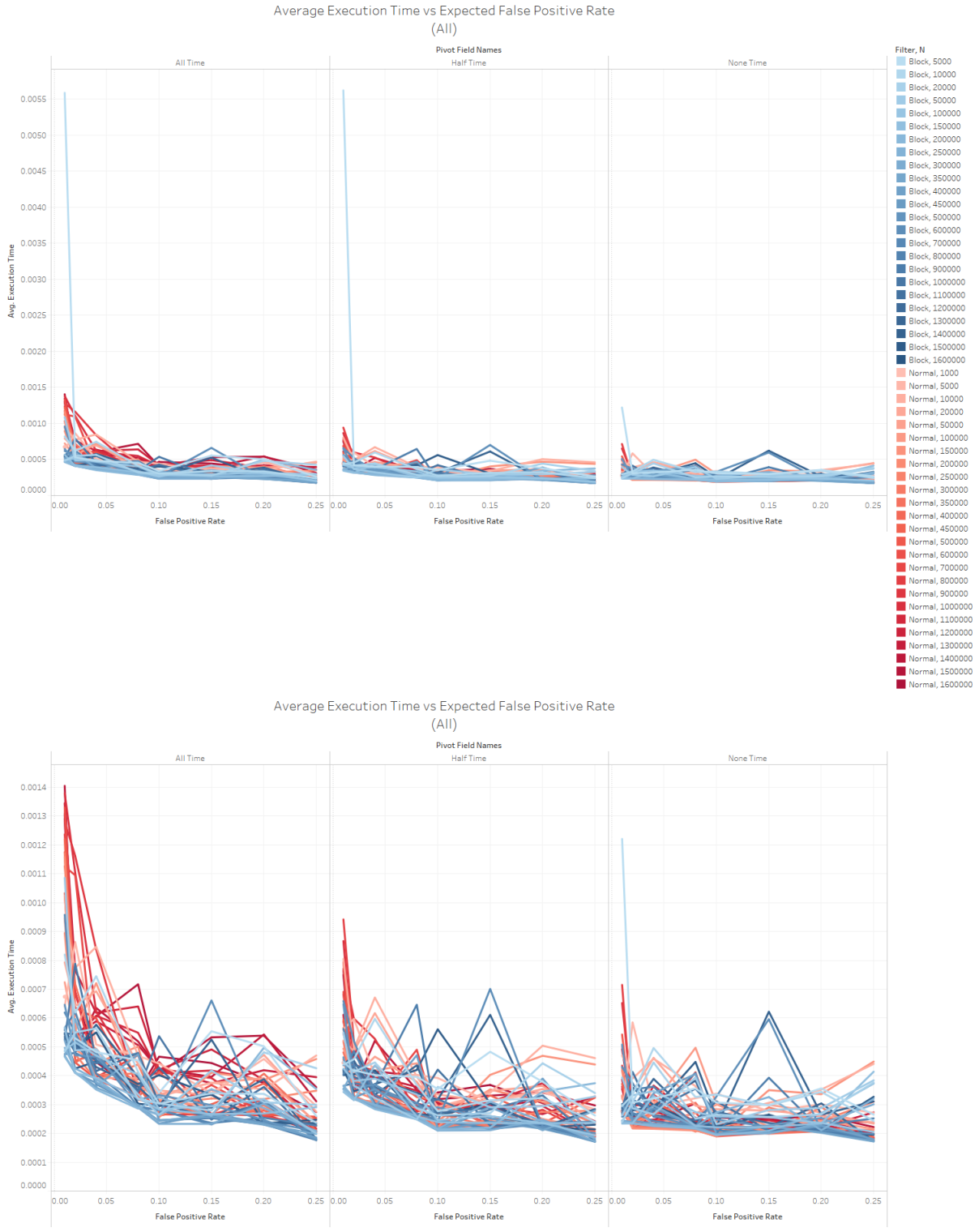

Figure 8: Here average run time for the three set of queries is plotted against Expected False Positive Rate. The left subplot contains results from All_present set, the middle one contains 50% set and the right figure presents None_present set. Red gradient represents basic Bloom filter and the Blue gradient presents the Block Boom Filter. We can observe that for higher False Positive rate, execution time decreases in all cases. This can be even clearer by omitting the highest execution times from All_present and Half_present dataset. This can be Clearer from the second figure.

Take Away

Choosing between basic and Block Bloom Filters is basically a trade-off between execution time an False Positive rate. Similarly, the choice of False Positive rate is also a trade of between precision and execution time (And also size of the filter). But this data structure with early exit feature is immensely important for larger datasets, like Reddit username and K-mer collections. Basic Bloom Filters