SCHOOL OF COMPUTER SCIENCE

Enhancing Efficiency and Accuracy in Approximate Membership Queries

A Comprehensive Study

Jayden Roh

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Monday 13th May, 2024

Abstract

We explore the Approximate Membership Query, a data structure using much less space compared to conventional sets by compromising small possibility of false positives. The primary focus is on the theoretical analysis and practical implementation of various AMQ filters including Bloom, Blocked Bloom, Cuckoo and XOR filters. These probabilistic data structures are designed to offer a balance between accuracy and efficiency, reducing memory usages while maintaining acceptable false positive rates.

A notable achievement is the detailed evaluation of the space complexity and false positive rates of these filters, which bridges the gap between theoretical and practical implementations. The research is extended to the empirical testing of these filters, identifying their performance and limitations of each filter.

Below, we present the key contributions and achievements:

- I wrote a total of 4000 lines of source code, implementing selected AMQ filters to access their efficiency and accuracy.
- I provided empirical evidence supporting the theoretical models, along with comprehensive performance comparisons.
- I have identified and addressed specific practical challenges in the implementation of AMQ filters, particularly the Cuckoo filter, due to its dependency on the underlying hash table mechanics.

Dedication and Acknowledgements

I would like to express my special thanks to my supervisor, Dr.Raphael Clifford, for suggesting the topic of the thesis, and his kind support.

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Taught Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, this work is my own work. Work done in collaboration with, or with the assistance of others including AI methods, is indicated as such. I have identified all material in this dissertation which is not my own work through appropriate referencing and acknowledgement. Where I have quoted or otherwise incorporated material which is the work of others, I have included the source in the references. Any views expressed in the dissertation, other than referenced material, are those of the author.

Jayden Roh, Monday 13th May, 2024

Contents

List of Figures

List of Tables

Ethics Statement

- This project did not require ethical review, as determined by my supervisor, Dr. Raphael Clifford;

Supporting Technologies

- I used `C++ 17` and its standard library to implement the data structures of Approximate Membership query.
- I used `Python Matplotlib` library to provide visuals for this thesis.

Notation and Acronyms

AMQ	:	Approximate Membership Query
T	:	the filter array
ϵ	:	target false positive rate
r	:	number of hash function used
C	:	bits per item
m	:	total bits of a filter ($m = n/C$)
n	:	number of items
α	:	load factor ($0 \leq \alpha \leq 1$)
f	:	fingerprint length in bits
b	:	size of a bucket (number of entries in a single bucket)
t	:	number of buckets
B	:	size of a block
l	:	number of blocks
c	:	edge capacity of a hypergraph

Chapter 1

Introduction

In modern technologies, the scale of data that we handle is much larger than in the past. In handling this high throughput of data, it is crucial to manage and process this data effectively. In this context, there has always been a need for a data structure that stores data and tells the user if the searched element is present in the structure. The traditional data structure, a set, has inserted every item to the structure without any modification of the data, which ensured 100% accuracy in terms of searching, but the problem arises that the structure may be inefficient in handling large number of data, in terms of space and time. To handle this issue, in compromising between less accuracy and better performance, the idea of Approximate Membership Query (AMQ) was introduced.

AMQ filters are probabilistic data structures capable of storing a large number of keys with significantly reduced memory usage. Despite allowing for a controlled inaccuracy, the benefits of reduced time in inserting and searching elements with only a small memory requirement often outweighs the drawbacks. The trivial inaccuracy in the filters occur in the form of false negatives, which is where the filter says the element is included in the structure, although it was not inserted. However, they ensure that there are no false negatives, meaning that every element which was inserted can be searched with no errors.

The benefits provided by the filters was enough to be implemented into actual use cases, such as deep packet inspection [5], network monitoring [10], web caching [30], malware detection [31], transactions in cryptocurrency [1, 3] and many other use cases [12].

The most conventional AMQ filter is the Bloom filter, designed by B.H.Bloom [8]. After the Bloom filter was introduced, there has been continuous research into AMQ systems, and designing an alternative to a Bloom filter, such as Cuckoo filter [18], Blocked Bloom filter [29], XOR filter [20], Quotient filter [19] and Ribbon filter [17]. Through the usage, there was a need for enhancing these filters to better align with practical demands, like reduced error rates and more efficient resource utilization.

Currently, as the design of these filters are very complex, some of the filters have chosen their intrinsic parameters, and its space complexity only by empirical testing, without any theoretical analysis. This research, therefore, reviews the theoretical analysis on selected AMQ filters, and based on the analysis, we will implement the filters and observe the gap between theoretical and practical outputs. Then, we will seek for possible improvements in optimising the filters in terms of space and time.

In Chapter 2 we provide a comprehensive background of the AMQ filters are observing; Bloom filter, Blocked Bloom filter, Cuckoo filter and Xor filter. The chapter than introduces basic concepts for hardware that are closely related to the implementation. Following the background, Chapter 3 offers a theoretical analysis on those selected filters, with optimisation ideas. To validate the theoretical analysis, Chapter 4 is used to setup the experiments and testings, including the implementations of the AMQ filters. In Chapter 5, we offer the results and further discussions about the results. Finally, in Chapter 6, we conclude our findings and discuss the future research and application based on the results.

The following summarises the objectives and goals for our project:

1. Analyse the theoretical performance of selected AMQ filters in terms of space-efficiency and false positive rates
2. Implement the filters which optimises the benefits of our own testing environment
3. Justify the theoretical performance by comparing it to the output of the implementation
4. Empirically test their performance on intrinsic variables that determine the performance of each filter
5. Compare the performances by the filters, and identify the strengths and weaknesses of each filters

Chapter 2

Background

2.1 Overview

This chapter starts by offering the fundamental concepts about AMQ filters and its key performance metrics, along with the critical role of hash functions employed in the AMQ filters. Then, four AMQ filters are introduced, with their implementation strategies for essential operations. Finally, this chapter will provide considerations in order to improve the performance of these AMQ filters.

2.2 Key Operations in Approximate Membership Queries

Approximate Membership Query (AMQ) filters are probabilistic data structures designed to store a set of elements S in space-efficient manner. They support dictionary operations such as Insert, Lookup, and sometimes Delete [13]. These operations are essential for managing data in environments where memory efficiency is critical.

2.2.1 Insert

Every AMQ filter features an $insert(k)$ operation where inserts the key k from the universe U into the set S. This process involves compressing the keys in a manner specific to the type of filter. As AMQs are designed to ensure there are no false negatives, the $insert(k)$ operation must function without errors. False negatives are situations where a search for an element that is in the set returns false, indicating that the element is not present when it actually is.

2.2.2 Lookup

The $lookup(k)$ operation returns true if $k \in S$ or false otherwise. While this operation is designed to be highly efficient, it may occasionally produce false positives. False positives occur when a search for an element that is not in the set returns true, saying that the element is present.

2.2.3 Delete

The $delete(k)$ operation removes a key k from S when the key is included in the set. The ability to delete elements from an AMQ without compromising the accuracy of other operations is a feature of AMQs. However, not all AMQs support this operation, thus it is out of scope in this project.

2.3 Key Features of Approximate Membership Query Systems

Like other data structures, every AMQ filter can be evaluated in terms of space and time complexity. However, attributes such as false positive rate, bits per item, construction time and query time are particularly crucial for AMQs. Each of these features are considered determining the overall effectiveness and applicability of an AMQ system.

2.3.1 False Positive Rate

The false positive rate(ϵ) is a critical metric for AMQ filters, reflecting the probability that a $lookup(k)$ operation incorrectly indicates whether the element is present. Minimizing ϵ is essential for maintaining the credibility and usability of the filter, especially in applications where accuracy is important. Hash functions play a critical role in the false positive rate, since the collision between hash functions would directly affect the false positive rate. The design of an AMQ aims to achieve a balance between a low false positive rate and other optimal utilization.

2.3.2 Bits per Item

The efficiency of an AMQ filter in terms of space is measured by 'bits per item'. The theoretical lower bound for a data structure for storing data without loss is $\log_2(1/\epsilon)$ bits per item [26]. For instance, a conventional data structure might allocate 64 bits per item to store n items, each consisting of 64 bit, resulting in $64n$ bits in total. In contrast, an AMQ designed with a false positive rate of $\epsilon = 0.01$ would only require 6.64 bits per item as a lower bound, since $\log_2(1/\epsilon) \approx 6.64$. Most AMQs operate within a factor of two of this lower bound. So, striving to refine the structure closer to the lower bound, whilst maintaining low false positive rates can be defined as the main aim of the AMQ system research.

2.3.3 Computational Performance

The performance of the code, including both construction and query time, is also regarded as an essential consideration in AMQ design. The construction time refers to the duration required to build the AMQ structure. In other words, it is the time to initialise the data structure itself and iterate $insert(k)$ operation on k in set S. This is highly dependent on the design of the AMQ and the number of hash functions used. Similarly, the query time, is the attribute that shows the time elapsed for operating $lookup(x)$. The query time can be categorised as positive query time and negative query time, which is checking the elements that have been added to the filter, and have not been added to the filter, respectively [12].

Given their role in determining the efficiency and accuracy of AMQ filters, it is essential to understand how hash functions are integrated within these data structures. The following Section 2.4 will explore the various roles and implementations of hash functions in AMQ filters.

2.4 Hash functions in Approximate Membership Queries

Hash functions are essential to the functionality of AMQs. This section outlines how hash functions are employed in various AMQ filters to optimise performance and minimize the space overhead required. A hash function is a function that maps an element to a fixed-length size output.

$$h : U \to [0, 1, ..., m-1] \tag{2.1}$$

According to Eq.(2.1), given a key k from universe U, it outputs a fixed-size m, which is $h(k)$ [13].

Hash functions in AMQs serve primarily two purposes:

- **Position Mapping** : Each of the hash functions maps items to a specific position within the AMQ structure, facilitating efficient item insertion and lookup. As the number of space allocated is much smaller than the number of items inserted (n), hash function contributes to align the item by assigning their index. However, as $U \gg m$, some keys are assigned to have same output of the hash which is called hash collision. Variants of AMQs have different strategies to solve this collision and ensure that there are no false negatives, and allowed as small false positives as possible.
- **Uniqueness - fingerprints** : In some AMQs, hash functions help ensure that each item is uniquely identified within the structure, reducing the likelihood of false positives during lookup operations. In comparison to conventional sets storing the key itself, they select a fingerprint function; which is a hash function [20]. Here, the membership query of a key x is done by comparing the fingerprint value of a key $f(x)$ equals to one of the stored value inside the set.

Ivanchykhin et al.[24] insists that effective hash function distribute hash values in a manner that appears uniformly random across the output space. This is to minimize the probability of collisions, which in

our case, to prevent the arise of false positive rate. Also, the hash function needs to be practical and fast enough to implement in the code. It has been shown that strong hash algorithms, such as MD5, SHA-1 and RIPEMD-320, require significant computational resources [2]. To maintain performance of our AMQs, MurmurHash finalizer is chosen for non-cyrptographic, simple hash function. Adding the selected random seed and the key added being x, the finalizer mixes x with following sequence [6]:

$$\begin{aligned} x &= x \oplus (x >> 33); \\ x &= x \times 0xff51afd7ed558ccd; \\ x &= x \oplus (x >> 33); \\ x &= x \times 0xc4ceb9fe1a85ec53; \\ x &= x \oplus (x >> 33); \end{aligned} \tag{2.2}$$

2.5 Specific Data Structures

This section, we will introduce four AMQ filters, Bloom filter, Blocked Bloom filter, Cuckoo filter and XOR filter. The design of the filters and its key features will be exhaustively introduced.

2.5.1 Bloom Filter

A Bloom filter [8] is a conventional probabilistic filter, which has huge advantage on its compact design. Setting $h_1, h_2, ..., h_r$ as a set of hash functions, and T as the array of a data structure, in insertion of an item x, we calculate $h_1(x), h_2(x), ..., h_r(x)$ respectively, and set the corresponding bits in T to 1. This can be restated of executing an OR operation to every position $T[h_1(x)], T[h_2(x)], ..., T[h_r(x)]$. The lookup process of an item x is even simpler, as follows [12]:

$$lookup(x) = T[h_1(x)] \wedge T[h_2(x)] \wedge ... \wedge T[h_r(x)] \tag{2.3}$$

The output in Eq.(2.3), denoted as $lookup(x)$, is obtained by performing a logical AND operation across the bits at the positions in the filter indicated by these hash values of x.

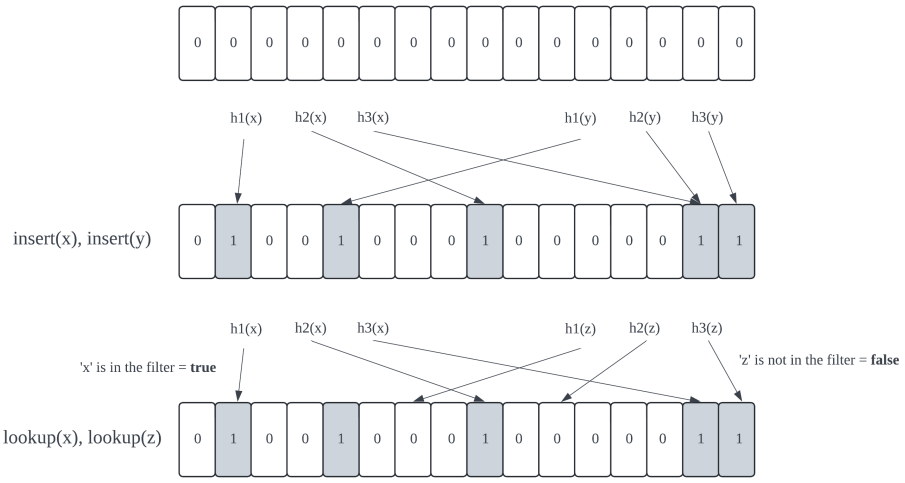

Figure 2.1: **Visualisation of a Bloom filter**. The Bloom filter is initialized, followed by insertion of items x and y. Afterwards, the lookup has been done with items x and z, returning true and false respectively.

The Bloom filter is known to use $1.44 \log_2(1/\epsilon)$ bits per item, when setting $r = \ln(2) \cdot C$ hash functions per query [8], where C indicate bits per item. The space complexity only depends on the false positive

rate ϵ.

Even though the simple nature of the construction, there are some deficiencies of the filter. The element can not be removed, since it is not guaranteed that every bit is determined by solely by a single element. Also, the performance is relatively low when ϵ is small [21].

2.5.2 Blocked Bloom Filter

The Blocked Bloom filter [29] is proposed to address the cache inefficiency issues of traditional Bloom filters. In a Bloom filter, a $lookup(x)$ operation can result in r cache misses, especially if the filter's size exceeds the cache line size of the architecture. To mitigate this issue, the Blocked Bloom filter consists of b smaller Bloom filters, each sized to fit within a single CPU cache line. A cache line is the unit of data fetched from the main memory by the CPU [23]. This will be discussed in more detail in Section 2.6.1. As data is fetched in chunks (cache lines), if all the required data resides within one unit, there is no need to access additional memory outside of this chunk. Each item is then assigned to one of these smaller filters, determined by a single hash function. The remaining hash functions are used to assign bits within a single cache line, similarly to how bits are assigned in a standard Bloom filter.

The filter gains huge advantage in terms of optimised lookup and computation. Also, due to the nature of the parallel design, it exploits from the single instruction multiple data (SIMD) instructions on modern processors. However, the bits per item rises to 30% of the one compared to the original Bloom filter in order to match the target false positive rate, due to the uneven distribution of load across the array of smaller Bloom filters.

2.5.3 Cuckoo Filter

A Cuckoo filter [18] is a efficient variant of a cuckoo hash table that stores only the fingerprints of items, rather than key-value pairs. As the fingerprint is much smaller than the key itself, this simplification not only facilitates easier implementation but also allows for item deletion by solely removing the item's fingerprint. Unlike traditional cuckoo hashing that requires moving the actual keys around within the table [32], the Cuckoo filter operates on fingerprints, thus eliminating the need to store or move actual keys. Another remarkable feature of the Cuckoo Filter is its ability to accommodate multiple elements in a single hash location, as they can be distinguished by the fingerprint. Fan *et al.* [18] defines an "entry" as a slot where a single fingerprint fits, and a "bucket" as a collection of entries that that share the same hash position.

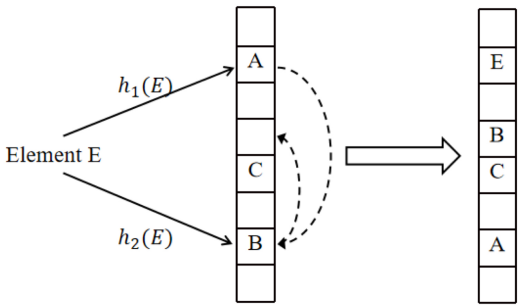

Figure 2.2: **Insert(E) operation on Cuckoo hash**. Figure taken from [32].

Figure 2.2 illustrates the insertion of cuckoo hashing table, of 8 buckets and each bucket having a single entry. This cuckoo hash table would be extended to a cuckoo filter if there are multiple entries in a single bucket.

The standard insertion procedure in Cuckoo filters involves two hash functions and a finger print function, with a method known as partial-key cuckoo hashing [18]. This method enhances traditional cuckoo

hashing by only requiring fingerprints, thus avoiding direct manipulation of keys. Assuming that $f :: \mathbb{N} \to \mathbb{N}$ is a fingerprint function, for an item x, the hashing process is described by:

$$\begin{aligned} h_1(x) &= hash(x), \\ h_2(x) &= h_1(x) \oplus hash(f(x)). \end{aligned} \tag{2.4}$$

Here, the xor operation in Eq.(2.4) ensures that each of the hash locations, $h_1(x)$ or $h_2(x)$ is dependent on the other through the fingerprint of x. This relationship allows the filter to create two hashes, only using a single hash and the fingerprint. Also, as $n \gg f$, the output of the second hash is at most f bits far away from $h_1(x)$, which increases the locality of two buckets. The benefit of being close to each other benefits in terms of lessening the likelihood of cache misses.

The insertion process of a Cuckoo filter can be described as below:

1. Compute $h_1(x)$, $h_2(x)$ and $f(x)$.
2. If the designated bucket for x (either $h_1(x)$ or $h_2(x)$) is occupied, use the another hash to determine its alternate position.
   (a) If the alternate location has empty entry, insert $f(x)$ into the position.
   (b) If the alternate location is also full:
      i. Randomly select one of the entry $f(y)$ and insert $f(x)$ in the position.
      ii. Find the alternate space for y by using Eq.(2.4).
      iii. Try inserting $f(y)$.
      iv. Repeat the process until no fingerprint is evicted.

During the insertion process, it is important to note that the repetition can potentially end in an infinite loop if too many items share the same hash values. In such cases, rehashing is necessary, which involves abandoning the current process and starting over with a new hash. Although the possibility is very low, further details will be discussed in Section 3.4.2.

Even if the process does not end in an infinite loop, it is still necessary to avoid circumstances where key evictions are frequent, as this can lead to significant computational overhead.

A naive solution for this is to set the threshold to ensure that repetitions do not exceed a designated value. The optimal threshold can be empirically determined by analyzing the runtime of insertions. However, this still leaves the problem of needing to rehash when the number of repetition is beyond the threshold.

To address this issue, Fan *et al.* suggests a slight increase in space to stabilize the insertion process by leaving some empty space for a potential unequal hash distribution. This approach involves setting the load factor ($0 \leq \alpha \leq 1$) of the filter to less than 1. The load factor indicates how many elements are loaded in the filter compared to its maximum space allocation by proportion. Empirical testing has determined that setting $\alpha = 0.94$, meaning that the filter is 94% full, is safe enough to not fall into the insertion failure, or excessively long operations [18].

Compared to complex insertion operation, the *lookup*(k) procedure is straightforward as it simply returns true if there is any entry matching the fingerprint in buckets of $h_1(k)$ or $h_2(k)$. This guarantees that there will be no false negatives, as the whole insertion phase holds a loop invariant that all of the items' fingerprint remains in either $h_1(k)$ or $h_2(k)$, and the bucket never overflows.

The fingerprint size is only determined by the false positive rate, ϵ. They are known to use $(\log_2(1/\epsilon) + \log_2(2b))/\alpha$ bits per item per key. When setting $\alpha = 0.96$ and $b = 4$, the value is $1.04\log_2(1/\epsilon) + 3.15$ bits per key [18]. Compared to the Bloom filter using $1.44\log_2(1/\epsilon)$ bits per key, Cuckoo filter uses less asymptotic bits, but it remains a fixed constant depending on the size of a bucket b.

2.5.4 XOR Filter

The XOR filter, referenced from [20], is a fingerprint-based filter that is distinguished by each slot containing a single element. The principal concept behind this filter is to assign a unique value to each slot for every item.

The insertion process begins by determining which slot each item will own. Initially, the filter calculates all r hashes per item and increments a counter to track how many items could potentially be assigned to each slot. It then iterates through the counter to identify slots that contain a single element. These elements are marked for a definite position and recorded in a stack, and decrementing their remaining $r-1$ hash positions in the counter. This iterative removal process in known as the peeling process [25]. Following the order established in the stack - reversed to utilize the First In, First Out (FIFO) property, the filter begins inserting values into the slots. Assuming f as a fingerprint function, T is a filter array and $h_p(x)$ was chosen for the item x, the insertion is executed using the following XOR calculation [16]:

$$T[h_p(x)] = \left(\bigoplus_{i=1}^{r} T[h_i(x)] \right) \oplus f(x) \tag{2.5}$$

An essential question becomes apparent regarding what happens if the values of the other $r-1$ slots gets altered after the XOR operation. The stack's implementation ensures that any slot positions determined during the entire insertion sequence. Thus, the last item to be peeled is guaranteed that its associated positions will not be modified afterward. If they were, the item would have been peeled earlier.

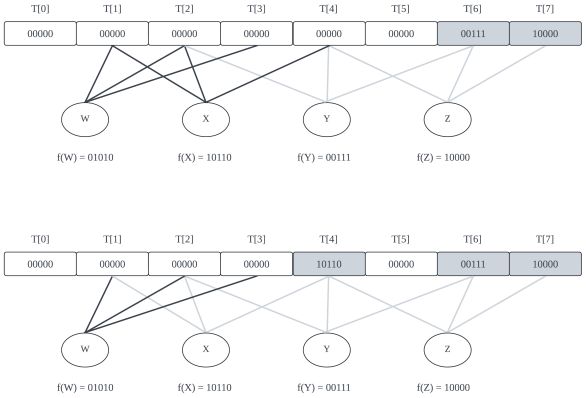

Figure 2.3: **Visualisation of Final Two Insertions in a XOR filter**. The diagram illustrates how hash functions map four elements W, X, Y and Z to their respective slots in the filter. Lines represent the hash functions, with the used hash functions highlighted in gray. Slots that have been assigned for insertions are also marked in gray.

The $lookup(x)$ operation simply compares the calculated value with the insertion method:

$$lookup(x) = \begin{cases} 1, & \text{if } f(x) = \bigoplus_{i=1}^{r} T[h_i(x)] \\ 0, & \text{otherwise} \end{cases} \tag{2.6}$$

This guarantees that there are no false negatives, as the filter uses XOR calculation internally within the lookup operation [20]. The properties of the XOR operation that facilitate this mechanism are:

$$\begin{aligned} X \oplus X &= 0 \\ X \oplus 0 &= X \\ 0 \oplus 0 &= 0 \\ A \oplus (B \oplus C) &= (A \oplus B) \oplus C \end{aligned} \tag{2.7}$$

Using the equation above, the filter ensures the integrity of the filter's operation, particularly during the lookup process in Eq.(2.6). Here's a detailed explanation:

$$\begin{aligned} \bigoplus_{i=1}^{r} T[h_i(x)] &= T[h_1(x)] \oplus ... \oplus T[h_p(x)] \oplus ... \oplus T[h_r(x)] \\ &= T[h_1(x)] \oplus ... \oplus \left(\bigoplus_{i=1}^{r} T[h_i(x)] \oplus f(x) \right) \oplus ... \oplus T[h_r(x)] \\ &= (T[h_1(x)] \oplus T[h_1(x)]) \oplus ... \oplus (T[h_r(x)] \oplus T[h_r(x)]) \oplus f(x) \\ &= 0 \oplus ... \oplus 0 \oplus f(x) \\ &= f(x) \end{aligned} \tag{2.8}$$

This proves that if an item have been inserted earlier, the lookup will always return true, confirming that there are no false positives.

In choosing the number of r, T.M Graf and D.Lemire [20] have shown that using 3 hash function for the calculation, $r = 3$, is enough for the performance, demonstrated from empirical testing.

The XOR filter is noted for its superior speed in handling membership queries compared to the Bloom filter and requires less space. Specifically, setting $r = 3$, it is uses just $1.23 \log_2(1/\epsilon)$ bits per item [20, 21]. However, constructing the filter, particularly implementing the peeling algorithm, is challenging and time-consuming. If any items remain unassigned, rehashing is necessary, similar to the process in Cuckoo filter. In addition, due to interconnected nature of values within the XOR filter, where each slot's value is dependent on the XOR of values at other slots, the entire insertion process must be completed in a single operation. Instead, all insertions must be performed simultaneously as a bulk operation.

2.6 Optimisation Opportunities

2.6.1 Memory Latency

Since AMQs are designed to handle large number of items quickly, the implementation must consider the architecture of the hardware, in terms of storing and fetching memory, since data is accessed from different locations during the program execution. The central processing unit (CPU), which is responsible for general computations, continuously reads and writes to the memory [23]. Here, the time delay between the processor issuing a read or write command and the action being performed is known as memory latency. Memory latency depends on the location where the actual data is located. Intuitively, if the data is physically closer to the processor, it takes less time to access the data, and longer time if it is further. To optimise operations, modern architectures are designed to store frequently accessed or recently referenced data closer to the CPU. This is achieved through a hierarchy of memory types including caches, random access memory(RAM), and hard drives. [23].

Operation	Latency Time	Note
L1 cache reference	0.5 ns	
Branch misprediction	5 ns	
L2 cache reference	7 ns	14x L1 cache
Mutex lock/unlock	25 ns	
Main memory reference	100 ns	20x L2 cache, 200x L1 cache
Send 1K bytes over 1Gbps network	10,000 ns	
Read 4K randomly from SSD*	150,000 ns	1GB/s SSD
Disk seek	10,000,000 ns	
Read 1 MB sequentially from disk	20,000,000 ns	

Table 2.1: **Memory latency in nanoseconds**. 1 ns = 10^{-9} seconds. Data retrieved from [9].

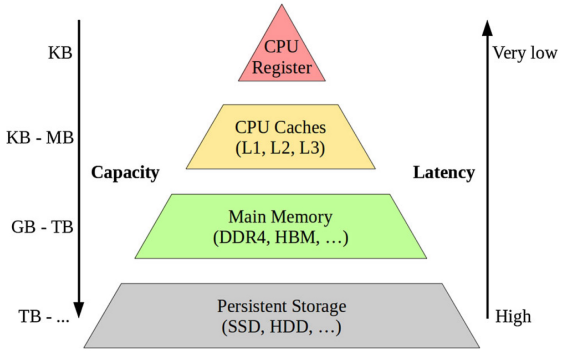

Figure 2.4: **Visualisation of Memory Hierarchy**. Figure taken from [27].

2.6.2 Cache Miss

Even though cache memories are very fast in retrieving memory, they are small; they can only store a small amount of data. Therefore, every event new data gets stored in cache, the processor has to provide space for the data by evicting one of the data that was already in the cache memory. Thus, cache memory is kept updated by adding and evicting the data by handling instructions. Referring to table 2.1, if the required data is already in the cache, which is cache hit, the program benefits from speed derived from less latency. However, if cache miss occurs, which is an event where the requested data is not found in the cache, as it was never placed, or evicted, the data has to be found in further places, obtaining penalties for high latency costs. Therefore, abundant cache misses should be avoided as it would decrease time performance.

2.6.3 Simple Instruction, Multiple Data

Simple Instruction, Multiple Data (SIMD) is an instruction set provided by the processor to handle multiple computations simultaneously [23]. SIMD has a feature of handling plural operations once, which exploits the property of data parallelism. The basic idea of SIMD derived from spotting an recursive basic operation in a loop. For example, adding two elements requires instructions of not only add, but also load, store, update and branch. In conventional methods, single instruction single data(SISD) would execute the instructions of load, add and update for every pair of operands. However, SIMD transforms these operands into vectors, converting the addition of numbers to be processed as vector additions. This method significantly reduces the number of calculations. This benefits in runtime, as time complexity is reduced.

Chapter 3

Theoretical Analysis

3.1 Overview

This chapter provides detailed analysis on the four AMQ filters explained in chapter 2, with its theoretical foundations of AMQ filters. The chapter then continues to explore methods to improve the performance in reducing false positive rate, space complexity and time complexity.

3.2 Bloom Filter

Bloom filter, from Section 2.5.1, is known that the calculation depends on number of hash functions r used. In this section, we will provide proofs about selecting the optimal number r, m and C. Furthermore, we would like to correct the approach of calculating false positive rate from Bose *et al.* [11].

3.2.1 Finding Optimal Intrinsic Parameters

We begin with theoretical analysis of the Bloom filter by determining the optimal number of hash functions, denoted as r. This analysis starts by defining the false positive rate ϵ in relation to m, n and r [12].

When performing the $lookup(x)$ operation, each r hash functions take the item x as an input and output the number in range $1, ..., m$, where m is equivalent to the size of the filter in bits. Here, we assume that hash functions are perfectly random, which means that the hash functions distribute values uniformly at random. A false positive occurs when all the hash bits corresponding to an item k, which was never inserted, are all set to 1. Initially, inserting a single item sets r bits to 1. After inserting n items, at most rn bits are set to 1. So, the possibility that a specific bit is chosen is $\frac{1}{m}$, so after a single insertion, the probability that a bit remains 0 is $(1 - \frac{1}{m})^r$. Thus, the probability p that a bit remains 0 after inserting n items is as follows [12, 29]:

$$p = \left(1 - \frac{1}{m}\right)^{rn} \approx e^{-\frac{rn}{m}} \tag{3.1}$$

Observe that the case rn bits set to 1 is only when all the hashes of n items doesn't have duplicated values. So, the actual probability p might be smaller than the Eq.(3.1), and the gap depends on the hash function used. This will be discussed later in this section, and we carry on proving with Eq.(3.1).

Let $p' = e^{-\frac{rn}{m}}$. Given the definition of the exponent, we can simplify p to p' for convenience:

$$\lim_{x \to \infty} \left(1 - \frac{1}{x}\right)^{x} \approx e^{-1} \tag{3.2}$$

The false positive rate ϵ could be derived from Eq.(3.1), as it is equivalent to the possibility of choosing r bits that have been set to 1:

$$\epsilon = (1 - p')^r = \left(1 - e^{-\frac{rn}{m}}\right)^r \tag{3.3}$$

To find the optimal r that minimises ϵ, we differentiate ϵ with respect to r. Letting $g = \ln \epsilon = r \ln\left(1 - e^{-\frac{rn}{m}}\right)$, the derivative can be shown as:

$$\frac{\partial g}{\partial r} = \ln\left(1 - e^{-\frac{rn}{m}}\right) + \frac{rn}{m}\left(\frac{e^{-\frac{rn}{m}}}{1 - e^{-\frac{rn}{m}}}\right) \tag{3.4}$$

Mitzenmacher, M. and Broder, A. [18] states that substituting Eq.(3.5) to the derivative will make it 0 and thus makes it global minimum:

$$r = \ln 2 \cdot \frac{m}{n} \tag{3.5}$$

Alternatively, Mitzenmacher suggests that using Eq.(3.1), g can be derived to:

$$g = -\frac{m}{n} \ln(p') \ln(1 - p') \tag{3.6}$$

By the symmetry of the natural logs, the minimum is on $p' = 1/2$, where again $p' = e^{-\frac{rn}{m}}$, leading to Eq.(3.5). Since $\frac{m}{n} = C$, the optimal number of hash function used in order to minimise ϵ can be restated as $\ln 2$ times of bits per item in the filter. Also, plugging Eq.(3.5) into Eq.(3.3) to obtain C, we can derive:

$$\begin{aligned} \epsilon &= \left(1 - e^{-(\ln 2 \cdot \frac{m}{n})\frac{n}{m}}\right)^{\ln 2 \cdot \frac{m}{n}} \\ &= \left(1 - e^{-\ln 2}\right)^{\ln 2 \cdot C} \\ &= \left(\frac{1}{2}\right)^{\ln 2 \cdot C} \end{aligned} \tag{3.7}$$

Rearranging Eq.(3.7), we can also obtain the theoretical bits per item C of the bloom filter.

$$C = \frac{\log_2(1/\epsilon)}{\ln 2} \approx 1.44 \log_2(1/\epsilon) \tag{3.8}$$

Obtaining theoretical bits per item, we could express how many bits needs to be allocated when n item is inserted from Eq.(3.8):

$$m = n \cdot C \approx 1.44 n \log_2(1/\epsilon) \tag{3.9}$$

3.2.2 Adjusted False Positive Rate

From the calculation of possibility of false positive, we have assumed that rn bits are set to 1 when inserting n items. However, the bits determined by $h_1(x), h_2(x), ..., h_r(x)$ may overlap, making p larger, and ϵ smaller in practical circumstances. This is because that we have assumed that the probability of each bit set to 1 is independent while we are adding rn bits to the array. Once a single bit is set from 0 to 1, the next time, the probability of a single bit being altered from 0 to 1 will decrease since there are now 1 less bits of 0.

Bose *et al.* [11] has pointed out this exact problem and suggested to bring an adjustment to this false positive probability, by taking the *Stirling number of the second kind* into account. Stirling number of the second kind, denoted as $\left\{ {n \atop k} \right\}$, represents the count of distinct ways to divide of n objects into k non-empty subsets [22]:

$$\left\{ {n \atop k} \right\} = \frac{1}{k!} \sum_{i=0}^{k} (-1)^i \binom{k}{i} i^n \tag{3.10}$$

To calculate accurate ϵ, First let A the event that the actual false positive occurs. The goal here is to evaluate $P(A)$, which will be ϵ. Then let E_I the event where in the filter T, $I \subseteq \{0, ..., m-1\}, |I| = i$ is the set of positions where the bits were set to 1 after attempting to set rn bits. $P(A)$ can be expressed with a conditional probability, which is adding the A happening in all possible array cases:

$$P(A) = \sum_{I \subseteq \{0,...,m-1\}} P(A|E_I) P(E_I) \tag{3.11}$$

$P(A|E_I)$ can be defined as:

$$P(A|E_I) = \left(\frac{|I|}{m} \right)^r \tag{3.12}$$

The event E_I occurs when each of the rn bits are assigned to set I of size i, which can be rephrased as partitioning rn bits into i number of sets without any empty sets. Ensuring that there is no empty set is crucial here, since if there is a empty set, it means that the certain bit hasn't been attempted to set to 1, which will remain 0. Thus, we could use the stirling number of the second kind, in Eq.(3.10), making $i! \cdot \left\{ {rn \atop i} \right\}$. The denominator will be rn bits each assigned to one of m slots, which will be simply m^{rn}:

$$P(E_I) = |I|! \cdot \frac{\left\{ {rn \atop |I|} \right\}}{m^{rn}} \quad = |I|! \left(\frac{1}{|I|!} \cdot \sum_{j=0}^{|I|} (-1)^j \binom{|I|}{j} j^{rn} \right) \tag{3.13}$$

Adding everything together, we obtain [11]:

$$\begin{aligned} P(A) &= \sum_{I \subseteq \{0,...,m-1\}} \left(\frac{|I|}{m} \right)^r \cdot |I|! \cdot \frac{\left\{ {rn \atop |I|} \right\}}{m^{rn}} \\ &= \frac{1}{m^{r(n+1)}} \sum_{i=1}^{m} i^r i! \binom{m}{i} \left\{ {rn \atop i} \right\} \end{aligned} \tag{3.14}$$

After evaluating Eq.(3.14), Bose *et al.* concluded that the actual false positive rate ϵ' denoting previous result in Eq.(3.7) as ϵ is [11]:

$$\epsilon < \epsilon' \leq \epsilon \cdot \left(1 + O\left(\frac{r}{\epsilon} \sqrt{\frac{\ln m - r \ln \epsilon}{m}} \right) \right) \tag{3.15}$$

This implies that the actual false positive rate might be slightly bigger than target false positive rate.

3.3 Blocked Bloom Filter

Blocked Bloom filter is designed [29] in order to mitigate the number of cache misses generated in Bloom filter. At most r cache misses might occur in standard Bloom filter, as the size of the filter does not fit into the memory when n is sufficiently large. Also, the involvement of hash function, in order to spread out the keys by adapting achieve avalanche effect, spreads out a single key's hash locations, which would make the CPU to randomly access the position and traverse around the whole filter, making high possibility that a single query or an insertion to slow down [24].

The Blocked Bloom filer takes the advantage of the fast and compact design of the Bloom filter, but splits the filters into blocks, where each of the block is an independent Bloom filter, to make the operations even faster. The size of the block B might depend on the architecture of the system, but the basic idea is to split the blocks so that it would fit in a single CPU cache line. The Blocked Bloom filter ensures that only a single block handles the insertion of an item. Therefore, there is a single hash assigned address the block for an item, and the rest of the hash functions contributes setting the bit to 1 inside this block. Hence, Blocked Bloom filter only needs a single cache miss for a single operation, which is the phase of finding the block determined by the first hash [29].

To determine the false positive rate of the Blocked Bloom filter, as a naive approach, we could think of equal possibility for a single element to get assigned to a certain block, $\frac{1}{l}$, where l is the number of blocks. As we have assumed that each block acts independent, the false positive rate of the Blocked Bloom filter might be same as the conventional Bloom filter, since letting the i'th block's false positive rate ϵ_i the expected value of ϵ would be $\epsilon' = \mathbb{E}(\epsilon) = \sum_{i=1}^{l} \frac{1}{l} \cdot \epsilon_i = \sum_{i=1}^{l} \frac{1}{l} \cdot \epsilon = \epsilon$. However, the number of elements assigned to a single block, denoting as n', will be different, where some will be over the expected value $\lambda = \mathbb{E}(n') = \frac{n}{l}$, and some will be under the value, which will affect each of the false positive rate due to varying number of elements n' from Eq.(3.3). In fact, n' is a random variable that follows the binomial distribution $B(n, \frac{1}{l})$, since the number of elements inside a single block is independent due to the assumption of uniform hashing [28]. The Binomial distribution then can be approximated by the Poisson distribution, as we got large n, and small $\frac{1}{l}$. The resulting false positive rate of Blocked Bloom filter can be described as the converging sum of each local false positive rate $\epsilon_{local}(B, i, r)$ multiplied with the possibilities obtained from the Poisson distribution [29]:

$$\epsilon = \sum_{i=0}^{\infty} Poisson(i) \cdot \epsilon_{local}(B, i, r) \tag{3.16}$$

Using the equation from Eq.(3.3), we can substitute $\epsilon_{local}(B, i, r)$ from the above equation:

$$\begin{aligned} \sum_{i=0}^{\infty} Poisson(i) \cdot \epsilon_{local}(B, i, r) &= \sum_{i=0}^{\infty} \frac{\lambda^i e^{-\lambda}}{i!} \cdot \epsilon_{local}(B, i, r) \\ &\approx \sum_{i=0}^{\infty} \frac{\lambda^i e^{-\lambda}}{i!} \cdot \left(1 - e^{-\frac{ri}{B}}\right) \\ &= \sum_{i=0}^{\infty} \frac{(n/l)^i e^{-(n/l)}}{i!} \cdot \left(1 - e^{-\frac{ri}{B}}\right) \end{aligned} \tag{3.17}$$

For the above equation, we have used Eq.(3.3) in calculating $\epsilon_{local}(B, i, r)$ rather than the corrected false positive rate Eq.(3.14), for easier calculation.

3.4 Cuckoo Filter

Extending from Section 2.5.3, we analyse the false positive probability derived from calculations, using the result to determine the optimal size of fingerprint f. Then, we will move on observing the likelihood of the failure of the construction.

3.4.1 Optimal Parameters

Cuckoo filter has different structural type compared to Bloom filter; on a single location, multiple items can be inserted. Each location, referred to bucket, can hold multiple items, defined as an entry. This section starts by finding the optimal fingerprint size f, with defining the false positive rate ϵ.

Referring to Eq.(2.4), the false positive of an uninserted key k occurs when 1) the fingerprint value of k, $f(k)$ matches one of the added key, and 2) the hash value of k, $h(k)$ matches either $h_1(x)$ or $h_2(x)$ [18]. One might wonder why only a single hash needs to be collided, not all h_1 and h_2 in order to achieve a collision. The reason is that as soon as one hash matches, the other hash matches if the output of the fingerprint is equivalent. The specific cases of both two hashes matching respectively are shown below:

Proof. Only one of the hash function and the fingerprint function collision is enough to make the collision in whole bucket in partial-key cuckoo hashing:

Consider two cases of an item k matching an arbitrary inserted key x where $x \in S$.

- Case 1: $h_1(k) = h_1(x)$ & $f(k) = f(x)$

  Here,

  $$\begin{aligned} h_2(k) &= h_1(k) \oplus h_1(f(k)) \\ &= h_1(x) \oplus h_1(f(x)) \\ &= h_2(x) \end{aligned}$$

  Since we have assumed that $h_1(k) = h_1(x)$ and we have shown that $h_2(k) = h_1(x)$, this case is valid.

- Case 2: $h_1(k) = h_2(x)$ & $f(k) = f(x)$

  In this case,

  $$\begin{aligned} h_2(k) &= h_1(k) \oplus h_1(f(k)) \\ &= h_2(x) \oplus h_1(f(x)) \\ &= (h_1(x) \oplus h_1(f(x))) \oplus h_1(f(x)) \\ &= h_1(x) \oplus (h_1(f(x)) \oplus (h_1(f(x)))) \\ &= h_1(x) \oplus 0 \\ &= h_1(x) \end{aligned}$$

  As $h_1(k) = h_2(x)$ and $h_2(k) = h_1(x)$, this case also holds.

□

We have shown that in two items, their two hash functions can collide to each other if only a single hash and a fingerprint function overlaps. The possibility of a key k matching an arbitrary key x is $\frac{1}{2^f}$. Therefore, the possibility of k not matching in a single entry is $1 - \frac{1}{2^f}$. Since there are $2b$ entries that should not collide, so the possibility p of at least one collision occurrence can be derived as follows:

$$\begin{aligned} p &= 1 - \left(1 - \frac{1}{2^f}\right)^{2b} \\ &\approx 1 - \left(1 - \frac{2b}{2^f}\right) = \frac{2b}{2^f} \end{aligned} \tag{3.18}$$

The approximation in Eq.(3.18) can be easily derived from Taylor series expansion. In order to meet the target false positive rate ϵ, $p = \frac{2b}{2^f} \leq \epsilon$ is required. Therefore, the lower bound of f is [18]:

$$f \geq \lceil \log_2(2b/\epsilon) \rceil = \lceil \log_2(1/\epsilon) + \log_2(2b) \rceil \tag{3.19}$$

3.4.2 Construction Failure

Unlike Bloom filters, Cuckoo filter have a certain probability of encountering failure during construction process. For this reason, Cuckoo filter has added a constant α ($0 \leq \alpha \leq 1$), which is a load factor, meaning that the filter only allows use of only α times of the whole space available for insertion. This could be restated to the fact that the filter prepares n/α slots to accommodate n keys. From now on, we will observe the probability of construction failure in terms of α.

Construction failure is very costly in terms of construction time, as the ongoing process will be abandoned and we have to start over inserting items, with rehashing. This part will examine the case failure occurrence and analyse its probability, and provide guidelines to set the intrinsic parameters of the filter in order to minimise failure. In the *lookup*(k) operation, if evicted key's alternative operation is full, the filter fails to insert the item. So, in order for successful construction of the filter, it has to be ensured that no more than $2b$ keys are mapped to the same two buckets [18]. This can be restated, that if there are more than $2b + 1$ keys having hash collisions in both buckets, the filter fails to construct. According to Eq.(3.18) and Eq.(3.19), the probability of a single key having fingerprint collision with an arbitrary key x is $\frac{1}{2^f}$, and the possibility of the key having collision in one of the hashes of x is $\frac{2}{t}$. In order for a key to have collisions in both buckets, those two possibilities must occur simultaneously, which makes $\frac{2}{t \cdot 2^f}$, where t is number of buckets in a filter. As there are ${}_n\mathrm{C}_{2b+1}$ possible sets to make the collision, we could define the failure probability of the construction:

$$\begin{aligned}
\binom{n}{2b+1}\left(\frac{2}{t \cdot 2^f}\right)^{2b+1} &\leq \left(\frac{e \cdot n}{2b+1}\right)^{2b+1}\left(\frac{2}{2^f \cdot t}\right)^{2b+1} \\
&= \left(\frac{2en}{(2b+1) \cdot 2^f \cdot t}\right)^{2b+1} \\
&= \left(\frac{2e \cdot t \cdot b \cdot \alpha}{(2b+1) \cdot 2^f \cdot t}\right)^{2b+1} \\
&= \left(\frac{2e \cdot t \cdot b \cdot \alpha}{(2b+1) \cdot \left(\frac{2b}{\epsilon}\right) \cdot t}\right)^{2b+1} \\
&= \left(\frac{e \cdot \alpha \cdot \epsilon}{2b+1}\right)^{2b+1}
\end{aligned} \tag{3.20}$$

The inequality $\binom{n}{2b+1} \leq \left(\frac{e \cdot n}{2b+1}\right)^{2b+1}$ in Eq.(3.20) is proven in Cormen, T.H. and Leiserson, C. E.'s book [13]. After substituting Eq.(3.19) in terms of f, we can say that the lower bound of the failure possibility depends on the load factor α, target false positive rate ϵ, and size of a bucket b.

On Eq.(3.20), even though ϵ is included in the equation, since this is the targeted false positive rate and decided before the construction phase, so we regard this as a constant. Thus, it leaves to 2 factors: α and b. It is obvious that increased bucket size b, will contribute to lessen the failure probability, as b is included both in the denominator and the power. Similarly, we can observe higher load factors (α) increase the risk of construction failure. This equation shows that if we set at least moderate size of b, the construction failure is unlikely to happen since a small increase in b will decrease the probability rapidly. For example, setting $b = 4$, and for $\epsilon = 0.01, \alpha = 0.96$, the probability has a upper bound of $3.87 \cdot 10^{-12}$.

However, remaining other variables but setting $b = 2$, the probability would rise sharply up to $6.58^{\cdot}-7$, but still being a negligible value. From this, we can conclude b highly affects the failure probability, but setting b to at least moderate value would keep the filter safe from construction failure.

Obtaining the formula for f and acknowledging that having some marginal space α is critical for the Cuckoo filter, we can now express bits per item C in terms of f and α. The size of the Cuckoo filter in bits, m, can be defined as number of entries in the filter. But also, n could be restated to number of entries multiplied by the load factor [18]:

$$C = \frac{m}{n} = \frac{f \cdot \texttt{(number of entries)}}{\alpha \cdot \texttt{(number of entries)}} = \frac{f}{\alpha} = (\log_2(1/\epsilon) + \log_2(2b))/\alpha \tag{3.21}$$

3.5 XOR Filter

XOR filter has a complex design of construction, where a single slot's value is determined by a single element's fingerprint and its outputs of r hash functions. We first identify the probability of false positives, and analyse the construction failure rate with the help of a hypergraph. We conclude this section with stating the minimum size required to obtain the target false positive rate.

3.5.1 Optimal Parameters

We start with configuring the optimal size of the intrinsic parameters, here which is f, the size of a fingerprint function in bits. According to Eq.(2.5), setting $r = 3$, the false positive occurs when the queried item k, is not inserted, but $T[h_1(k)] \oplus T[h_2(k)] \oplus T[h_3(k)]$ equals to the fingerprint function $f(k)$. Regardless of the value of $T[h_i(k)]$ where $i \in \{1, ..., r\}$, the possibility p of false positive is stated below:

$$p = \frac{1}{2^f} \tag{3.22}$$

Since the obtained false positive p should be smaller than the target false positive rate ϵ, we get $p = \frac{1}{2^f} \leq \epsilon$. Rearranging this equation, we obtain the bound of f, which is $f \geq \log_2(1/\epsilon)$.

3.5.2 Construction Failure

The XOR filter also faces a risk of construction failure, which can result in substantial computational overhead. However, unlike the Cuckoo filter, whose failure probability is straightforward to calculate (as shown in Eq.(3.20)), the XOR filter's failure probability is more complex to determine due to its complex insertion process. Unlike other AMQs, where insertions can occur in stepwise, the XOR filter requires that all keys should be inserted simultaneously. If the filter needs to accommodate new keys, it must be reset and reconstructed from the beginning, including both the previously added and the new keys [16]. The failure can occur anytime during the simultaneous insertion, especially in the peeling process. During the construction phase, each key $k \in S$ is tentatively placed in slots determined by its hash values $h_1(k), h_2(k)$ and $h_3(k)$. We count the number of keys assigned to each slot and identify slots with a unique key, termed 'lone elements'. These lone elements are confirmed in their slots, and the peeling process begins by iteratively removing the confirmed keys and its hash values, therefore updating the slot assignments. During the process, if there are no more lone elements identified while unassigned keys remain, the construction fails, requiring rehashing.

The process can be conceptually modelled as operations within a hypergraph. A hypergraph [7] is a graph where an hyperedge E can join more than two vertices. Here are definitions about hypergraphs relevant to our discussion [25]:

Definition 3.5.1 (k-core hypergraph). The k-core of a hypergraph H is the maximum subgraph of H in which every vertex has degree of at least k.

Definition 3.5.2 (r-uniform hypergraph). The r-uniform hypergraph is a hypergraph which each edge consists of exactly r distinct vertices.

Definition 3.5.1 represents a stable state in the peeling process where each remaining vertex is linked by at least k keys. Applying these concepts to our Xor filter, the filter's slots can be viewed as m vertices, and each key from a set of n keys forms a hyperedge connecting three distinct vertices based on its hash values $h_1(k), h_2(k)$ and $h_3(k)$. Initially, the Xor filter, forms a 3-uniform hypergraph, making $r = 3$. Let $G^r_{m,cm}$ a hypergraph with m vertices, and cm hyperedges where each edge connects r different vertices. This is the r-uniform hypergraph, having c as the edge density of $G^r_{m,cm}$. In our case, $r = 3$, and since we have n edges, $cm = n$ and $m \approx 1.23n$ according to Graf and Lemire [20], yielding an edge density $c \approx 0.813$.

The goal of the peeling algorithm in this setup is to iteratively peel vertices, reducing the original 3-uniform hypergraph to its 2-core by repeatedly removing vertices with less then two connections. The complexity of this operation and the probability of successfully emptying the hypergraph (which indicates successful filter construction) depend significantly on the initial distribution of hyperedges across the vertices, and the edge density c. If the peeling process concludes with any vertices still having two or more edges, it indicates a construction failure.

As we assume the hash functions are uniformly random, we can also assume that the hyperedges are distributed fairly. The problem rises on the edge density. There was a previous research about the threshold values $c^*_{k,r}$ that, the resulting subgraph is empty with probability $1 - O(1)$ when $c < c^*_{k,r}$. However, if $c > c^*_{k,r}$, the k-core subgraph is nonempty with the possibility $1-O(1)$. Molly and Achlioptas [4] provides the formula for $c^*_{k,r}$:

$$c^*_{k,r} = \min_{x>0} \frac{x}{r\left(1 - e^{-x}\sum_{j=0}^{k-2}\frac{x^j}{j!}\right)^{r-1}} \tag{3.23}$$

According to Eq.(3.23), inserting $k = 2, r = 3$ leads to $c^*_{2,3} \approx 0.818$. To ensure that the peeling process terminating with high possibility, we have to set our filter's edge capacity c where $c < c^*_{2,3}$. As $c = \frac{n}{m} = \frac{1}{C}$, we have a bound of $c = \frac{1}{C} < c^*_{2,3} \approx 0.818$. Therefore, we get the approximation of $C > 1/0.818 \approx 1.222$. Since C stands for bits per word here, we obtain that XOR filter needs at least 1.23 words per item, where a single word has a bound of $f = \log_2(1/\epsilon)$ due to Section 3.5.1. Wrapping up, we conclude that a XOR filter requires at least $1.23\log_2(1/\epsilon)$ bits per item, which is the same result established from Graf and Lemire [20].

Chapter 4

Project Execution

4.1 Overview

In this chapter, we introduce the overall methodology of implementing four AMQ filters, Bloom, Cuckoo, Xor and Blocked Bloom filter, moving on to optimization strategies on those four. The chapter will follow on discussing the experimental setup of comparing the metrics of the benchmark result.

4.2 Implementation Details

In this section, we briefly explain how we have implemented the filters, starting from the development environment, moving towards the experiment setup, then identifying the objective and determining the variables of each experiment.

4.2.1 Language and Libraries Used

Language

Since the project is often exposed to high throughput, and a fast paced environment, there was a need to choose a language that is fast, and capable of sophisticated memory management. Due to these reasons, C++ 17 was chosen as the programming language due to its exceptional performance, and having efficient memory management. In addition, C++'s support for object-oriented programming allows for organized and scalable code structure, which was utilised by empirical testing.

Benchmark

The benchmark was done by using the C++'s standard library `std::chrono`. The class `std::chrono::high_resolution_clock` provides smallest tick period provided by the library. Instead of using third-party benchmark library, using the standard library is beneficial as it requires no additional dependencies, and providing direct access to system clocks and timing.

Graph Tool

Python's `Matplotlib` library, particularly its `pyplot` module, was chosen for generating plots and visualisations of the experimental data. `Matplotlib` offers a comprehensive range of plotting tools that are both powerful and flexible. Also, its concise design makes users easier to implement visuals.

4.2.2 Filter Design

In order to maintain consistency in running benchmark files, each filter was designed into a separate class. For convenience reasons, we have ensured that all the classes have same signature on initialisation, $insertion(k)$ and $lookup(k)$.

Initialisation

For the initialisation, every filter takes 3 arguments; n, ϵ and `construct`. Since the objects that we are creating are static, the value n is given to reserve the space for inserting n elements. Combined the value of n, the target false positive rate ϵ determines the size of the filter. After the initialisation of the data structure, randomly assigned keys are inserted to the filter depending on a boolean variable `construct`.

Insertion

The $insert(k)$ operation is implemented in every filter with the name $add(k)$ and $addAll(ks)$. $add(k)$ requires an unsigned 64 bit integer and adds the key in terms of the logic of the specific filter, and $addAll(ks)$ requires a vector of a key set, which simply iterates $add(k)$ for all $k \in ks$. Each filter is designed to throw an exception and exit the program when the insertion fails due to any unexpected circumstance

Lookup

The $lookup(k)$ operation, is also have shared signature throughout the whole filter, which takes an unsigned 64 bit integer for the input. This operation returns true if k is found in the set with its respective logic, and false otherwise.

4.3 Experiment Setup

In this section, we introduce the methodology of the experiments to benchmark our implemented filters. In order to obtain unbiased results throughout our experiments, we have ensured that these criteria were maintained consistent.

4.3.1 Generating Random Keys

In designing the AMQ filters, it was assumed that there were no insertions of a duplicated keys. Some filters, the insertion of duplicated items can lead to unwanted results as hash functions generate identical output for same keys. Although performing $lookup(k)$ operation before every insertion seems to be a straightforward solution, relying on the filters which are being tested could introduce biases in the experiment setup. Therefore, to obtain fair results, we ensured all keys generated are distinct. To achieve this, we have implemented a function that generates a 64-bit unsigned integer and adds it to a set, and adds to the key list only if the integer is not found in the set. Even though this requires linear time due to checking for set membership, it still guarantees that each key is unique, allowing us to evaluate the filters' performance under unbiased conditions. Also, these keys were precomputed prior to the benchmark, which does not affect the benchmark overall.

To ensure the uniform distribution of the keys, which is as crucial as the uniformity of the hash functions, we employed C++'s standard library `std::uniform_int_distribution` and `std::mt`. This function generates keys with a uniform probability distribution, where the probability of any integer i being generated within the range $[a, b]$ is defined by the equation [14]:

$$P(i|a, b) = \frac{1}{b - a + 1} \tag{4.1}$$

4.3.2 Positive and Negative Lookup

For benchmarking $lookup(k)$ operations in every filter, there was a need to separate positive and negative queries. Positive query is a taking a $lookup(k)$ operation where $k \in S$, which means querying the filter for the item that has been added. On the opposite, negative query is the operation for the item that has not been included. Those two should be differentiated apart since the AMQs have different runtimes on both queries originated from their design. It can be observed that no AMQ filter returns false on positive queries, since that simply implies the situation of a false negative, and for the main property of AMQ filters was having no false negative.

4.3.3 Measuring False Positive Rate

To compare the actual false positive rate with the target false positive rate ϵ, we have generated twice of the random keys, which is $2n$. Then, we insert n of the keys to the AMQ filters. Since we ensure that the key set has distinct keys, the $lookup(k)$ operation on the remaining n keys are regarded as negative lookup. Then, the false positive rate calculated by counting the number of times that the filter has outputted true divided by number of trials, n.

Algorithm 1 Measuring the False Positive Rate

Input: A set U of $2n$ unique keys
Input: AMQ filter object F
Output: Measured false positive rate $\epsilon_{observe}$

```
split U into two disjoint sets S and T each of size n
c ← 0
for all i ∈ S do
    F.insert(i)
end for

for all j ∈ T do
    if F.lookup(j) = True then
        c ← c + 1
    end if
end for
ε_observe ← c/n
return ε_observe
```

4.3.4 Unit Testing

No False Negatives

To achieve the fundamental identity of all AMQ filters, the unit test was included to check if there are any false negatives. After generating n keys, we insert all of them, and check if any of them returns false on a positive query. If the false negative is detected, we throw an exception implying the message of the detection and terminate the program, since the filter is unreliable.

Hash functions

In order to test hash functions, we test the avalanche effect of the hash function [24]; where small change of input should generate big difference in output. The effect is crucial as it prevents the hash output being biased when inserted sequential keys.

4.3.5 Benchmarking

Hardware

Our test platform is an Apple Mac featuring the M2 chip, which operates on an ARMv8 architecture. The architecture provides SIMD instruction set, where each variable could hold up to 128 bit. This system includes an Apple M2 processor with a base clock speed of 3.2GHz, equipped with 8 cores and 16GB of unified memory. The cache line size is 128Mb, and having 128 Kb of L1 data cache, 4Mb of L2 cache, and 8Mb of L3. The program was compiled with the `Clang 14.0.3`. For compiling the following flags were used:

- -O3 : Optimisation level provided by clang, used for higher performance improvements.
- -march=native : Ensures that the compiler to use best available instruction set, here, for Blocked Bloom filters, which require ARMv8 instruction set architecture.
- -fprefetch-loop-arrays : Used to prefetch memory for loops.
- -funroll-loops : Used for reducing number of iterations in a loop, for optimisation purposes.
- -DNDEBUG : Turned off the debugging for slightly performance improvement expectation.

Timing

To measure the time duration, we have wrapped the operation to the time function *now*() from `std::chrono` and subtracted the times to obtain the interval. However, even though the provided class has the smallest tick available, the ability to calculate exact time was limited due to the variable assignment and function callbacks, which took tens of nanoseconds on average. It might seem trivial, but as the project deals with a fast paced algorithm, to mitigate the inaccuracy of running the clock, we have expressed the times with the duration of a operation divide by number of keys. Also, to obtain consistent results, every benchmark has been run at least 10 times and the representative data was chosen to its average.

To solely calculate the operation time elapsed in terms of a fair manner, the operations such as key generation, or random seed generation for hash construction was precomputed to the operation. However, the time for each hash calculation is included in every operation since it is the fundamental of the operations. In addition, to avoid any external factors that could affect the timing in terms of operation, the benchmark was held in an environment where no programs were run except for the benchmark, and the network was blocked.

4.4 Testing Objectives

For each experiment, some features were determined before the experiment:

- Number of elements n
- Target false positive rate ϵ

For empirical testing for each filters, several additional features were precomputed or collected via results, depending on the objective of the experiment:

- Number of Hash functions used r
- Size of a Block B
- Load Factor α
- Construction Failure Probability

Using the determined values and the initialisation of the filters, the benchmarking observes the following features in general:

- Measured False Positive Rate
- Bits per Item C
- Construction Time per Key
- Positive Query Time per Key
- Negative Query Time per Key

4.4.1 Empirical Testing

Throughout the research, we have found that some of the intrinsic parameters the contribute to the construction of the filter have been only determined by their own empirical testing. As the testing environments vary, and some could be actually compared with the theoretical calculation, in this section, we will try our own version of empirical testing and compare with the computed values, where applicable.

Bloom Filter - number of hash functions used (r)

The Bloom Filter uses $r = \ln 2 \cdot C$ number of hash functions to minimise false positive rate, as proven in Section 3.2.1. However, it is obvious that if more hash functions are involved in the calculation, the construction time and query time would increase. We will observe how the number of hash functions affect in time, and also the measured false positive rate. The test will be split into three cases, with low, moderate and high target false positive rates, respectively:

	n	ϵ	C (2 d.p.)	optimal r	α
Test 1	1,000,000	0.001	14.35	10	1
Test 2	1,000,000	0.01	9.57	7	1
Test 3	1,000,000	0.1	4.78	4	1

Table 4.1: **Variables Used in Bloom Filter Test - Number of Hash Functions Used**.

The value of C is obtained from Eq.(3.8), and the optimal r is calculated via Eq.(3.5). The ceiling function is applied to the r value is used since r is an integer.

Bloom Filter - Measured False Positive Rate

As mentioned in Section 3.2.2, we will analyse the difference of target false positive rate and the measured false positive rate. Especially, we will focus on whether the measured false positive rate exceeds ϵ, as Bose *et al.* [11] proved:

	n	ϵ	α
Test 1	10,000	[0.001, 0.01, 0.1]	1
Test 2	100,000	[0.001, 0.01, 0.1]	1
Test 3	1,000,000	[0.001, 0.01, 0.1]	1
Test 4	10,000,000	[0.001, 0.01, 0.1]	1
Test 5	100,000,000	[0.001, 0.01, 0.1]	1

Table 4.2: **Variables Used in Bloom Filter Test - Observed False Positive Rate**.

Blocked Bloom Filter - Size of the filter

The bits per item in Blocked bloom filter has been determined empirically to ensure that the actual false positive rate is consistently less than the target false positive. It is known that Blocked Bloom filter uses 30% more space than Bloom filter, which would be approximately $1.872 \log_2(1/\epsilon)$. However, due to the false positive probability of Blocked Bloom filter introduced in Section 3.3, we could calculate the equation and obtain ϵ in a certain circumstance.

Let β a space overhead that a Blocked Bloom filter has over Bloom filter. Then, bits per item of a Blocked Bloom filter would be $\beta \cdot 1.44 \log_2(1/\epsilon)$. The goal is to find the minimum β which the calculated probability in Eq.(3.17) is under ϵ. We set the variables $B = 512, n = 1,000,000, \epsilon = 0.01, r = 7$.

Since $l = m/64 \cdot B$, as we are using 64-bit integer, and obtaining m from the following equation,

$$\begin{aligned} m &= \texttt{(Bits per Item of Blocked Bloom filter)} \cdot n \\ &= \texttt{(Bits per Item of Bloom filter)} \cdot \beta \cdot n \\ &= 1.44 \log_2(1/\epsilon) \cdot \beta \cdot n \end{aligned} \tag{4.2}$$

we obtain $l = \frac{1.44 \log_2(1/0.01) \cdot \beta \cdot 1,000,000}{512 \cdot 64} \approx 292 \cdot \beta$. With the $\lambda = n/l$, we plug in the values to Eq.(3.17) and calculate the results.

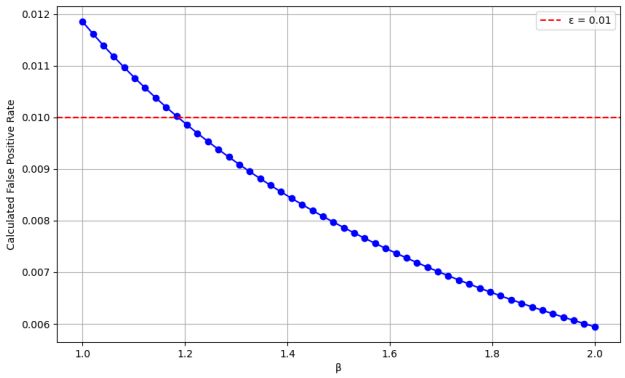

Figure 4.1: **Calculated False Positive Rate of Blocked Bloom Filter to Space Overhead** β. The red line indicates target false positive rate $\epsilon = 0.01$, and the blue line is β, the scale of Blocked Bloom filter's space usage compared to Bloom filter. The false positive rate was calculated via Eq.(3.17).

From Figure 4.1, we can see that the β intersects the false positive rate at $\beta = 1.19$. Therefore, we assume that blocked bloom filter uses 19% more space. To validate $\beta = 1.19$ is enough for Blocked Bloom filter to achieve under the target false positive rate, we measure false positive rate, and compare with the target value:

	n	ϵ	β	B
Test 1	1,000,000	0.01	1.19	512

Table 4.3: **Variables Used in Blocked Bloom Filter Test - Measuring False Positives**.

Cuckoo Filter - Finding Optimal Bucket Size

The number of entries within a bucket, which is the size of buckets, affects the load factor α of the Cuckoo filter, as larger b leads to higher allowance of hash collision, which reduces the need of key eviction. The goal of this experiment is to calculate how much space overhead is required compared to the theoretical lower bound in order to accommodate all keys that needed to be inserted. In this experiment, we have intentionally set the size of the filter identical to the lower bound of the AMQ filter, where the bits per item is $\log_2(1/\epsilon)$. The decision was held in order to illustrate more distinct result between number of b:

	b	n	ϵ	Bits per item Set
Test 1	1	[10,000, ..., 10,000,000]	0.01	6.64
Test 2	2	[10,000, ..., 10,000,000]	0.01	6.64
Test 3	4	[10,000, ..., 10,000,000]	0.01	6.64
Test 4	8	[10,000, ..., 10,000,000]	0.01	6.64

Table 4.4: **Variables For Cuckoo Filter Test - Size of Buckets to Load Factor.**

However, as b is highly related to the false positive rate according to Eq.(3.18), we also have to take account how b affects the false positive rate.

Cuckoo Filter - Validating Failure Probability

As failure of construction impacts huge on runtime, we have shown the probability of failure in Section 3.4.2. And using Eq.(3.20), we have shown that the failure probability is unlikely to happen if we set at least moderate size of b. To observe whether the suggested equation is actually bounded to the suggested probability, we test on high target false positive rate with a small size of b to check the frequency of construction failure happening, comparing with the theoretical value:

	b	n	ϵ	α	sample N	Calculated Construction Failure
Test 1	1	1,000,000	0.5	1	10,000,000	0.09299
Test 2	2	1,000,000	0.5	1	10,000,000	0.00148
Test 3	4	1,000,000	0.5	1	10,000,000	$4.08 \cdot 10^{-8}$

Table 4.5: **Variables For Cuckoo Filter Test - Measuring Failure Probability.**

As the theoretical probability obtained from cases with b larger than 1 is too small, we will create filter with elements full in terms of capacity for $N = 10,000,000$ times and count how many times the construction failure has occurred when attempting a single key. Then, we will compare the frequency with the calculated to the value, since the frequency that we will obtain from the experiment, the sample is large enough to approximate according to the law of large numbers [15].

Cuckoo Filter - Size of a fingerprint to False Positive Rate

Another critical factor in Cuckoo filter, which is the size of the fingerprint, also plays a crucial role in the filter's performance. Even though the increase of f directly impacts C, f has a lower bound of $\lceil(\log_2(2b/\epsilon)\rceil$ as shown in Eq.(3.19). However, we have applied a ceiling operation to f, as the feature should be an integer. Since the approximation of f could lead to slightly biased results, we have tested how the false positive rate is affected by f:

	b	n	ϵ	optimal f
Test 1	4	1,000,000	0.001	6.32
Test 2	4	1,000,000	0.01	9.64
Test 3	4	1,000,000	0.1	12.97

Table 4.6: **Variables For Cuckoo Filter Test - Size of a Fingerprint.**

XOR Filter - Locality of the Hash functions

One thing to note from the theory of XOR filter from Graf and Lemire [20], is that unlike other filters, each of the hash function in XOR filter has its own designated range, where the ranges are not overlapping each other. For example, when $r = 3$, and assuming that the size of the filter is c, the range of each hash function is as follows : $h_0 : S \to \{0, ..., c/3 - 1\}, h_1 : S \to \{c/3, ..., 2c/3 - 1\}, h_2 : S \to \{2c/3, ..., c - 1\}$. To find out whether this approach is more efficient compared to the hash functions having full range, we simulate the filter with both methods and observe the construction and query time, followed by the measurement of the false positive rate:

	Type of Hashing	n	ϵ
Test 1	Partitioned Range	1,000,000	0.01
Test 2	Full Range	1,000,000	0.01

Table 4.7: **Variables For XOR Filter Test - General Performance to Type of Hashing.**

4.5 Comparison of Filters

In this section, we setup the experiments of comparing the filters via benchmarked output.

4.5.1 Parameter Used in Filters

In order to obtain consistent results, we have fixed some parameters for each filter:

- Bloom filter : We use $r = \lceil \ln 2 \cdot \frac{m}{n} \rceil$ hash functions for each calculations.
- Blocked Bloom filter : The block size $B = 512$ is used. Even though the cache line size of testing environment can utilise up to $B = 1024$, as the SIMD instruction set could only support variables' sizes up to 128 bits, we decided to use $B = 512$ bits for optimisation reasons. Also, we set the bits per item 30% increased one from Bloom filter, which is $1.872 \log_2(1/\epsilon)$.

- Cuckoo filter : We set the load factor $\alpha = 0.96$, and fix the bucket size to $b = 4$. In addition, the size of the fingerprint is set to $f = \lceil \log_2(1/\epsilon) + \log_2(2b) \rceil$.
- XOR filter : We use $r = 3$ hash functions per operation. We fix the bits per item to $1.23 \log_2(1/\epsilon)$. We use partitioned range hashing for hash functions.

With those filters, we compare those features below with the following variable set.

4.5.2 Bits Per Item

	ϵ
Test 1	[0.0001, ..., 0.5]

Table 4.8: **Variables For Measuring Bits per Item.**

4.5.3 Measuring False Positive Rate

	n	Number of Queried Keys	ϵ
Test 1	1,000,000	1,000,000	0.01
Test 2	1,000,000	1,000,000	0.016

Table 4.9: **Variables for Measuring False Positive Rates by Filters.**

4.5.4 Construction Time

	ϵ	n
Test 1	4	1,000,000

Table 4.10: **Variables For Comparing Construction Time.**

4.5.5 Lookup Time

	Type	n	ϵ	Number of Lookups
Test 1	Positive Lookup	1,000,000	0.01	1,000
Test 2	50% Positive, 50% Negative Lookup	1,000,000	0.01	1,000
Test 3	Negative Lookup	1,000,000	0.01	1,000

Table 4.11: **Variables For Cuckoo Filter Test - Size of a Fingerprint.**

Chapter 5

Critical Evaluation

5.1 Empirical Testing

In this section, we represent the results obtained from Section 4.4.1 and have a discussion about the findings.

5.1.1 Bloom Filter

Finding Optimal Number of Hash Functions

ϵ	optimal r	r	Measured Fpr	Construction Time(ns)	S.D.	Query Time(ns)	S.D.
0.001	10	8	0.001122	32.36	14.42	18.27	0.37
0.001	10	9	0.001028	26.99	11.63	20.04	0.11
0.001	10	10	0.001003	21.61	2.71	21.94	0.23
0.001	10	11	0.001019	28.14	15.83	23.49	0.24
0.001	10	12	0.001032	27.58	7.82	24.37	0.18
0.01	7	5	0.011134	11.19	1.58	14.83	0.26
0.01	7	6	0.010186	12.13	0.19	17.16	0.11
0.01	7	7	0.010059	14.17	0.21	19.25	0.13
0.01	7	8	0.010556	16.00	0.18	21.76	2.24
0.01	7	9	0.010545	17.89	0.33	22.00	0.38
0.1	4	2	0.116249	4.43	0.18	9.79	0.12
0.1	4	3	0.100788	6.28	0.14	13.91	0.19
0.1	4	4	0.102390	8.15	0.21	16.18	0.37
0.1	4	5	0.114177	10.00	0.19	17.15	0.51
0.1	4	6	0.132511	12.00	0.19	18.20	0.49

Table 5.1: **Performance of Bloom Filter Varied by the Number of Hash Functions Used, to the False Positive Rate (ϵ).** The variables were set according to Table 4.1. Each case was executed 15 times, and the average and the standard deviation was presented. For construction time and query time, the values are divided by the number of keys, which were rounded at 2 decimal points.

Table 5.1 represents the performance of the Bloom filter varied by number of hash functions used. When measuring query time, $n = 1,000,000$ items were queried, where half of the items were inserted before *lookup*(k) operations, and the other was not inserted.

First, looking at the measured false positive rate, in every case, the result shows that the measured false positive rate has a tendency to have the smallest value near optimal r and the value increases as r gets further from the optimal value. The cases $\epsilon = 0.001$ and $\epsilon = 0.01$ has the minimum value when r is exactly the optimal, whilst in the case $\epsilon = 0.1$ shows the minimum where $r = 3$. This is derived from applying the ceiling function to obtain the optimal r, since the value obtained from Eq.(3.5) where $\epsilon = 0.1$ is 3.313,

which is closer to 3 than 4. So, in terms of the false positive rate, it can be interpreted that rounding the value of r rather than applying the ceiling function is more accurate. In addition, the tendency of r to be less than the optimal value is due to the increased likeliness for the *lookup*(k) to return true, as it requires less numbers of bit set to 1. On the other hand, in the cases of r being higher than the optimal, the possibility of hash collisions between keys is more likely to occur, thus leading to higher false positives.

Moving on to construction time, the insertion time outputs are increased as r increases, in moderate and high target false positives ($\epsilon = 0.01, 0.1$). This is due to more assigned hash functions per key to execute. However, in the low target false positive rate, $\epsilon = 0.001$, seems to have the least execution time when $r = 10$. This is an unexpected outcome, as cases of $r = 8, 9$ take longer time than $r = 10$. Regarding to the high standard deviation of these two outcomes, we believe that there are outliers that contribute to the high average of construction time.

The construction time in general is shown to be proportional to the log scale of $1/\epsilon$, as the bits per item only depends on ϵ as stated from Eq.(3.8).

It is observed that the query time per key increases as r increases, throughout all the tests. This is expected, as the time includes calculating hash functions, therefore increasing the number of it would definitely lead to increased computation. In terms of ϵ, although there is a decrease of an average query time as ϵ increases, the proportionality is not clear, compared to the construction time. This is due to the nature of the *lookup*(k) operation in Bloom filter, as in negative query, the operation returns false immediately when the filter encounters a hashed bit set to 0. For example, if $T[h_1(k)] = 0$ in *lookup*(k) operation, the processor returns false, as the boolean will return false regardless of the values of $T[h_2(k)], ..., T[h_r(k)]$.

In summary, in terms of selecting the number of hash functions, applying the floor operation rather than the average or ceiling operation to the calculated result seems to be the optimal approach, as using less hash function leads to improved performance in terms of execution time, and the actual false positive rate does not increase to a significant extent.

Analysis on False Positive Rate

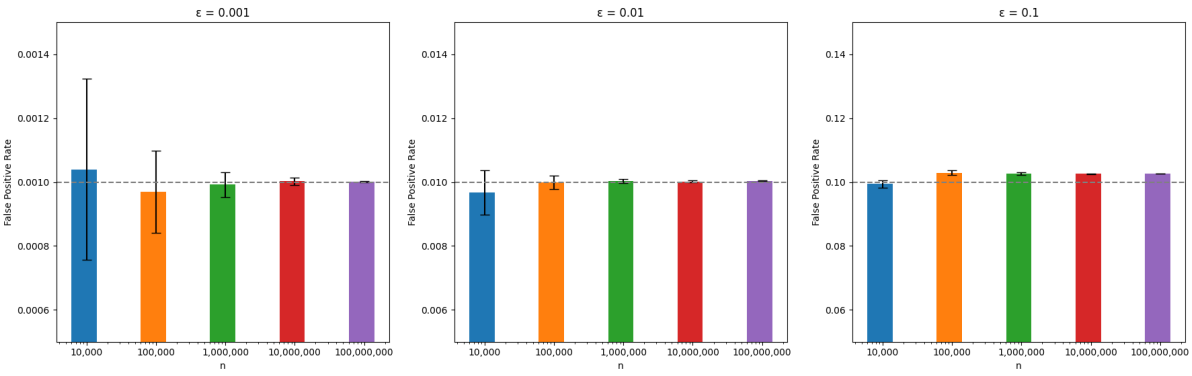

Figure 5.1: **Measured False Positive Rate to the Number of Keys Inserted (n), Varied by Target False Positive Rate (ϵ)**. The variables were set from Table 4.2. The dotted lines indicate the false positive rate ϵ. Each cases were done 10 times, and each of the average was presented. The error bars, indicating the range of the samples, are presented in a black line.

Figure 5.1 illustrates the difference of measured false positive rate in terms of ϵ and n. In all of the cases when n is small, there is high deviation, leading to unstable results, but as n increases, the measured false positive rate seems to converge, which is higher than the target false positive rate. This also occurs in Table 5.1. This indicates that the modified false positive rate proven by Bose *et al.* [11] is more accurate.

From the empirical testing of Bloom filter, we conclude that the target false positive rate ϵ as a minor error, as we have to take into account the fact that the possibility that a certain bit being set to 1 is

dependent on prior results. Also, in the case of finding the optimal number of hash function r, we could disregard the decimal points from the calculations, as we have observed minimal difference in terms of actual false positive rate, but we obtain better runtime as one less hash function is involved.

5.1.2 Blocked Bloom Filter

Bits Per Item

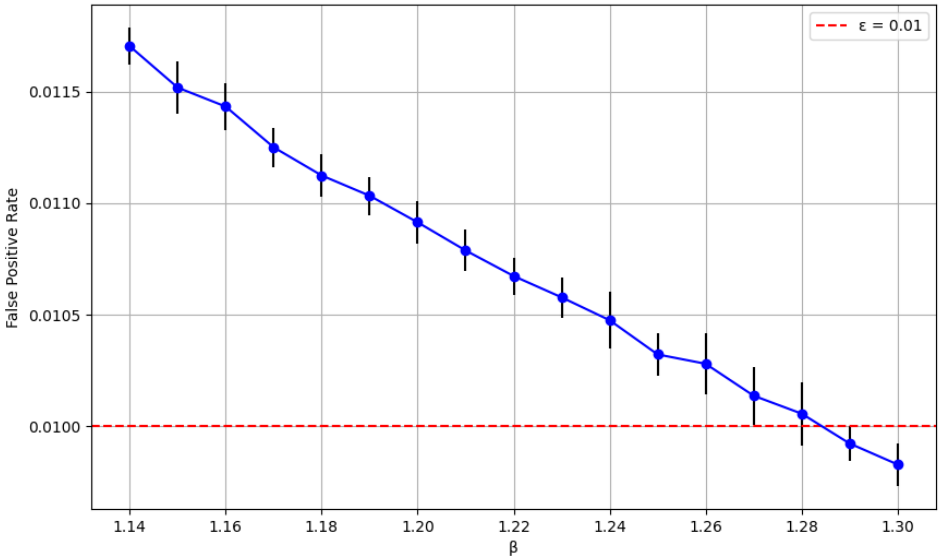

Figure 5.2: **Measured False Positive Rate of Blocked Bloom Filter on Space Overhead (β).** β represents the space overhead of Blocked Bloom filter compared to Bloom filter. The variables were set from Table 4.3. The dotted lines indicate the false positive rate ϵ. Each cases were executed 15 times, and each of the average was represented. The error bars, indicating the range of the samples, are presented in a black vertical line. The lines were connected for readability.

Figure 5.2 represents the measured false positive rate of a Blocked Bloom filter with β. Although we have set $\beta = 1.19$ for the minimum size in Table 4.3, the point where $\beta = 1.19$ showed higher false positive than the target, with the average value 0.011033. This was somewhat expected, as we used the uncorrected value of each block's false positive probability in Eq.(3.17) rather than the corrected probability. Since the corrected false positive rate of a Bloom filter was slightly higher, the minimum β for the calculated false positive which was under the target, was biased, in the result being lower. From the figure, we observe that β where the average probability including the error bar being underneath the target is first seen at $\beta = 1.29$, meaning that the Blocked Bloom filter uses 29% more space than the Bloom filter to obtain the same false positive rate, which supports the empirical testing obtained from Putze *et al.* [29].

5.1.3 Cuckoo Filter

Finding Optimal Number of Bucket Size

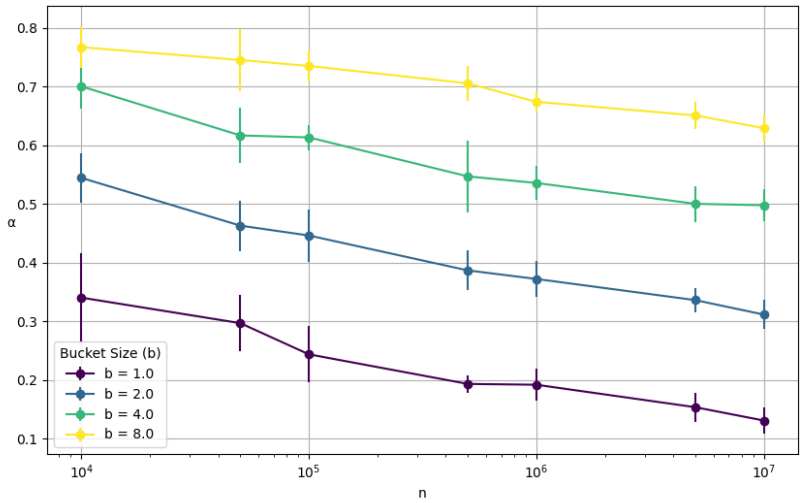

Figure 5.3: **Maximum Load Factor (α) for a Cuckoo Filter to Number of Keys Inserted (n) based on Size of a Bucket (b)**. The variables were set from Table 4.4. Each cases were executed 10 times, and each of the average was represented. The error bars, indicating the range of the samples, are presented in a black vertical line.. The lines were connected for readability.

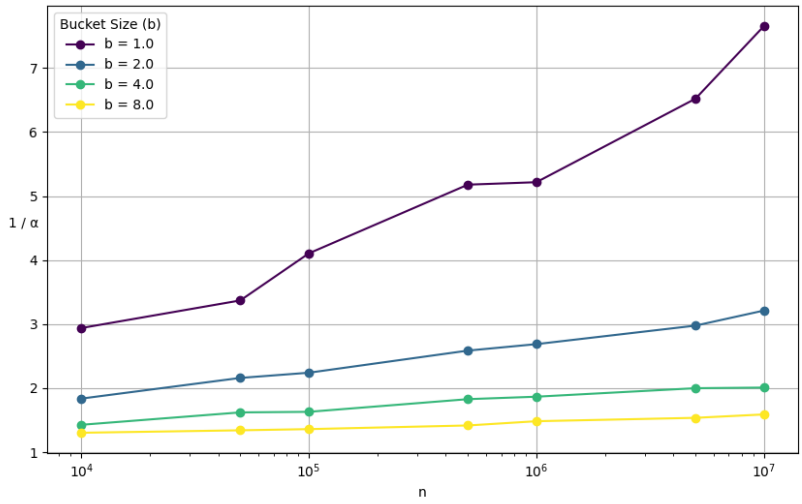

Figure 5.4: **Space Overhead ($1/\alpha$) of Cuckoo Filter to Number of Inserted Keys (n) in terms of Size of a Bucket (b)**. The y axis represent the inverse of α, which is the space overhead of Cuckoo filter compared to the theoretical lower bound. The average value taken from Figure 5.3.

Figure 5.3 represents the maximum load factor α, when attempting to insert n keys to the Cuckoo filter with a size of $\log_2(1/\epsilon)$. As expected, the yellow lines, where a single bucket held 8 entries, was capable of accommodating the most keys. If we take the inverse of the value of α, we can estimate how much space overhead is required to accommodate all the intended keys. For example, if $\alpha = 0.25$, it means the filter could only insert 25% of the keys, so in order to insert every key, we need $1/0.25 = 4$ times of space compared to the theoretical value. Figure 5.4 illustrates the following result. From the figure,

we first examine that all of the cases in terms of b shows increased value as n increases. This is due to the increased possibility of the hash collision in terms of both hash function and fingerprint function. Since the size of the hash function and fingerprint function is irrelevant to the number of keys inserted, we would have to allocate more space per key as n increases. Examining each cases, $b = 1$ showed high overhead of space even if we insert relatively less number of keys, for example, we have to allocate 2.93 times of theoretical space even in the case where adding $n = 10,000$ keys. In contrast, $b = 4$ and $b = 8$ seemed stable, as they have shown only tentative increase when n increased. In case of $b = 4$, only 2 times of theoretical space had to be prepared and 1.59 in the case of $b = 8$, when inserting $n = 10,000,000$ keys, respectively. At first it seems intuitive to use lower b value for the optimisation of space as low b decreases the size of a fingerprint, thus leading to lower bits per item according to Eq.(3.21). However, doing so in fact results in increased space, as the load factor decreases dramatically, which is also a factor determining space complexity. We is bad.

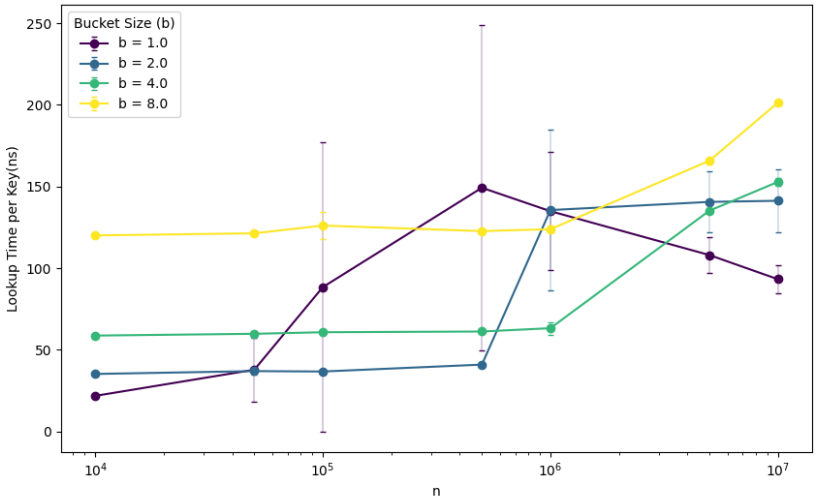

Figure 5.5: **Average Lookup Time per Key of Cuckoo Filter to Number of Keys Inserted (n) in terms of Size of a Bucket (b).** The lookup operations were done in a combination of half of the added keys, and the other half which was not added. The variables were set according to Table 4.4. Each cases were executed 10 times, and each of the average was presented. The error bars, indicating the range of the samples, are presented in a gray vertical line.

To check how b affects Cuckoo filter in a time wise manner, we checked the average lookup time per key. Figure 5.5 demonstrates the result. According to the figure, when we use a single entry in a bucket, the lookup time fluctuates on different number of keys. Also, the query time is unstable even in the same setup, illustrating high deviations. This is due to an increased number of buckets, as they can put only a single item in a bucket. Even comparing with $b = 2$, the number of buckets is approximately doubled, resulting in longer delays due to the memory's random access. If we have a bucket size which is too big, for example, $b = 8$, , in most of the cases, the time is asymptotically larger than others due to the linear search inside a single bucket. In contrast, bucket sizes being moderate, in $b = 2$ and $b = 4$, shows stable time throughout the number of keys inserted.

b	n	ϵ	Calculated Failure	Measured Failure Rate	S.D.
1	1,000,000	0.5	0.09299	0.060931	0.001844
2	1,000,000	0.5	0.00148	0.00148	0.000009
4	1,000,000	0.5	$4.08 \cdot 10^{-8}$	0	0

Table 5.2: **Construction Failure Rate in Cuckoo Filter to Size of a Bucket (b).** Variables were set according to Table 4.5. The measured rate was the number of failures divided by $N = 10,000,000$ trials. Each case was executed 10 times, and the average rate and the standard deviation was displayed.

Table 5.2 compares the calculated construction failure to actual measured failure rate. The result in $b = 4$, shows that the construction failure never happened through out $n = 1,000,000$ insertion in $N = 100,000,000$ trials, so we can conclude that $b = 4$ is enough to prevent the occurrence of failure. As the false positive rate involves the size of bucket in Cuckoo filter, there is no need to use more than $b = 4$, as using more entries in a single bucket increases computational overhead, leading to increased operation time. On smaller bucket sizes, $b = 1$ and $b = 2$, we observe the measured rate on average is still bound by the calculated value, whilst the case in $b = 2$ has exactly the same output. Considering that the calculation from Eq.(3.20) is also an approximation, we have expected a value which is smaller than the offered bound, but considering that the hash function applied cannot be uniform in practice, it is still an acceptable result.

Observing the experiments so far, low b results in bigger space usage, and introduces a potential of constriction failure. On the other hand, if the value of b gets too high, the time complexity rises, due to expanded search area within a bucket. Therefore, it is ideal to use $b = 4$ as a optimal bucket size, as it shows good performance in terms of time and space in general.

Finding the optimal number of Size of a Fingerprint

ϵ	optimal f	f	Calculated False Positive Rate	Measured False Positive Rate	S.D.
0.001	12.97	11	0.003906	0.003775	0.000059
0.001	12.97	12	0.001953	0.001886	0.000044
0.001	12.97	13	0.000977	0.000951	0.000028
0.001	12.97	14	0.000488	0.000466	0.000020
0.001	12.97	15	0.000244	0.000238	0.000019
0.01	9.64	8	0.031250	0.029738	0.000152
0.01	9.64	9	0.015625	0.014915	0.000127
0.01	9.64	10	0.007813	0.007574	0.000063
0.01	9.64	11	0.003906	0.003796	0.000054
0.01	9.64	12	0.001953	0.001897	0.000024
0.1	6.32	5	0.250000	0.214968	0.000516
0.1	6.32	6	0.125000	0.114058	0.000316
0.1	6.32	7	0.062500	0.058817	0.000164
0.1	6.32	8	0.031250	0.029776	0.000193
0.1	6.32	9	0.015625	0.015011	0.000174

Table 5.3: **Measured False Positive Rate vs. Calculated False Positive Rate in terms of Fingerprint Size (f) in Cuckoo Filter.** The variables were set in accordance of Table 4.6. Each cases were tested 10 times, and each of the average was presented. The false positive rate calculation was done with Eq.(3.18).

From Section 3.4, we have observed that the size of the fingerprint, f, is involved in false positive rate and bits per item, which was introduced in Eq.(3.18). Table 5.3 illustrates the comparison of the measured false positive rate and calculated false positive rate in terms of f. Theoretically, the false positive possibility becomes halved when we increase f by 1. The measured false positive rate seems to follow this tendancy in all cases, also maintaining lower values than its calculated false positive rate, respectively. Also, it is important to note that if we apply rounding or floor function to the optimal f in each cases, the value is not bound to the target false positive anymore. For example, in the case of $\epsilon = 0.1$, the optimal f is 6.32, and if we discard the decimal points, the corresponding f would be 6. The measured false positive rate of $\epsilon = 0.1, f = 6$ is 0.114058, which is above our target rate 0.1.

Thus, we conclude that the calculation of false positive rate in Eq.(3.18) fits in practice, and using larger bits in fingerprint helps lower the false positive rate. Also, the consideration of f is restricted to the values which are strictly over the optimal f, as failing to do so would not meet the target false positive rate.

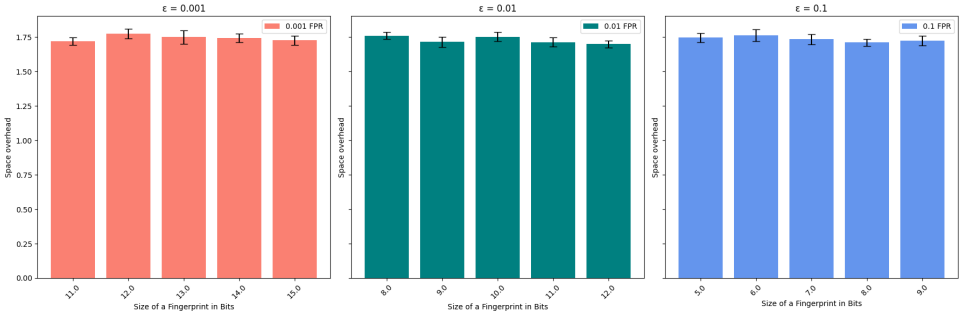

Figure 5.6: **Space Overhead Compared to Theoretical Lower Bound in Cuckoo Filter in terms of Size of a Fingerprint(f)**. The variables were set from Table 4.6. Each case was executed 10 times, and each of the averages were represented. The error bars, indicating the range of the samples, are presented in a black vertical line.

However, it is apparent that if we use more fingerprint bits, the size of the filter would be directly impacted since bits per item of a Cuckoo filter is equivalent to $C = f/\alpha$, according to Eq.(3.21). However, we are unsure that whether load factor α is dependent on the size of a fingerprint. Figure 5.6 demonstrates the space overhead compared to the theoretical lower bound, which is an inverse of the load factor α. As shown, in all cases of different target false positive rate ϵ, we can see there are no significant difference in the space overhead in terms of f. Therefore, we can state that the indicator of space complexity, C, decreases as the size of the fingerprint decreases.

Therefore, in determining the size of a fingerprint in Cuckoo filter, we get lower false positive rates as f increases, but doing so would also increase the size of a filter. Therefore, applying the ceiling operation to the theoretical f seems to be optimal.

5.1.4 XOR Filter

Locality of the Hash functions

Type of Hashing	Construction Time per Key (ns)	S.D.	Query Time per Key (ns)	S.D.
Partitioned Range	517.272	5.896	8.104	2.127
Full Range	518.914	8.225	7.669	2.393

Table 5.4: **Construction Time and Query Time per Key by Type of Hashing in XOR Filter.** The lookup operation was done on half of the keys that were inserted in advance, and half of the keys that were never inserted. Each test was done 15 times, and the mean and the standard deviation is represented. The variables were set according to Table 4.7.

As shown in Table 5.4, the construction times between partitioned and full range hashing strategies show no significant differences. This indicates that the concern of the construction failures derived from hash collision in fact does not affect the actual construction time. However, there is a notable difference in the standard deviation where hash functions used in full range has much higher value, 8.225. This can be interpreted as full range hash functions are less efficient in terms of cache handling due to poorer locality. Moving to query time, the slightly faster query time in the full range may indicate that once the hash table is successfully constructed, accessing only r locations with possibility of the data location being anywhere outperforms the case where each hashes are ensured to be separated by some margin.

Type of Hashing	ϵ	Measured False Positive Rate	S.D.
Partitioned Range	0.01	0.007799	0.000059
Full Range	0.01	0.007829	0.000116

Table 5.5: **Measured False Positive Rate by Type of Hashing in XOR Filter.** Each test was done 15 times, and the mean and the standard deviation is represented. The variables were set according to Table 4.7.

Table 5.5 illustrates the measured false positive rate by the type of hashing. Although both strategies' measured rates are bound to the target false positive rate ϵ, the XOR filter using partitioned range shows better false positives. This is due to reduced collision rate, as partitioned range ensures that hash values are distributed throughout the filter.

Wrapping up, full range hashing might benefit from slightly better query times, but partitioned hashing provides more consistent results with better construction time and false positives due to the ensured hash distribution.

5.2 Comparison of Filters

In this section, we compare the results obtained by each filter, discussing each filter's strength and weakness.

5.2.1 Bits Per Item

Figure 5.7 represents the actual bits per item allocated to reach the target false positive rate ϵ. All of the filters' bits per item tends to decrease as ϵ increases. On small ϵ, for example, $\epsilon = 10^{-4}$, Blocked Bloom filter uses most space among the filters, which is nearly 25 bits per item. In contrast, XOR filter has required least bits per item in order to achieve the false positive rate where ϵ has set to low value. For the high *epsilon* cases, both XOR and Bloom filter seemed to store the least bits per item, which is approximately 5 bits per item to achieve $\epsilon = 0.1$.

Throughout different ϵ, it has been shown that XOR filter consistently shows low bits per item compared to other filters. For the Cuckoo filter, although it is known to use asymptotically low bits per item, as its bits per item also depend on the bucket size b and is added as a constant (Eq.(3.21)), it is revealed that they use more space practically in high ϵ cases. This is because the constants ($\log_2(2b)$) weighs more especially when ϵ is high ($\log_2(1/\epsilon)$). On the other hand, Bloom filter has a good performance in terms of space in high ϵ cases, but it uses more space in case of small target rates. Lastly, as we know that Blocked Bloom filter uses nearly 30% more space than Bloom filter, it uses the most space in small ϵ conditions, and moderate space in terms of high rates.

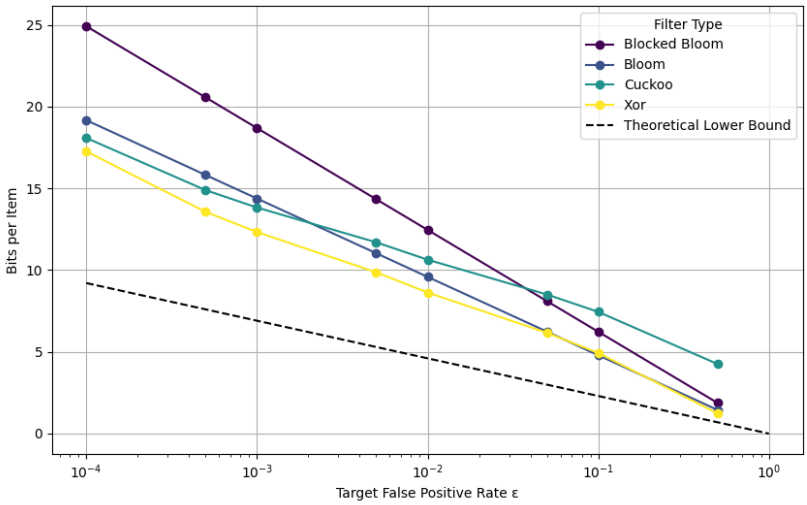

Figure 5.7: **Measured Bits per Item to Achieve Target False Positive Rate (ϵ).** The black dotted line indicates theoretical lower bound $\log_2(1/\epsilon)$. Lines were connected for readability. The variables were set according to Table 4.8.

5.2.2 Measuring False Positive Rate

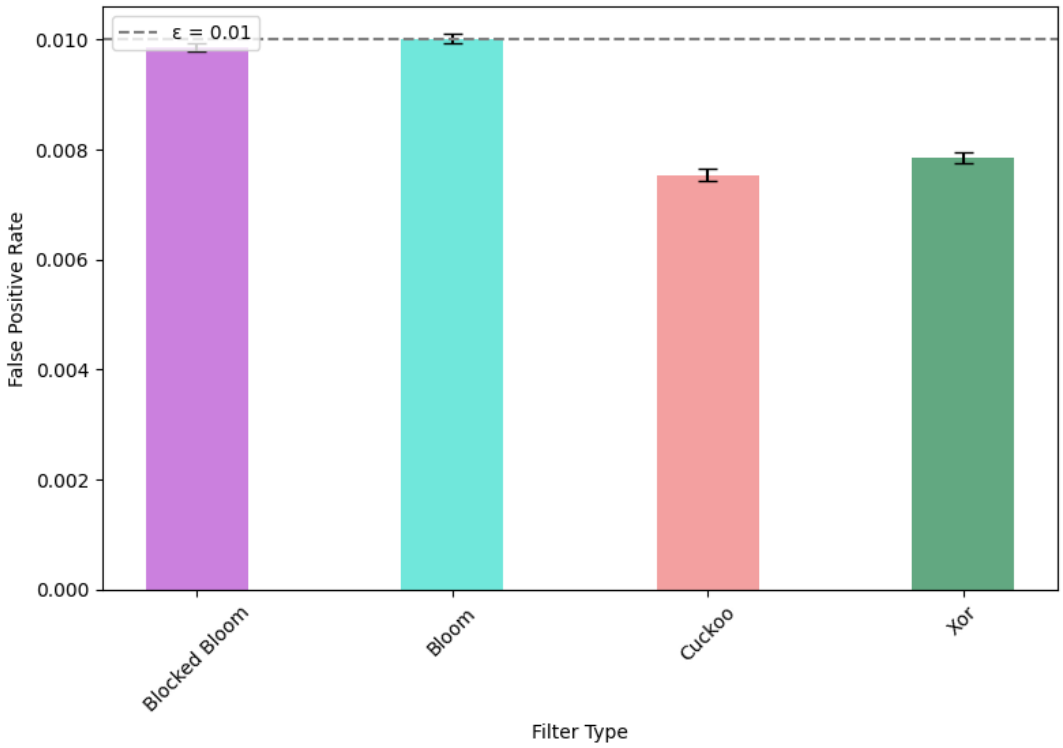

Figure 5.8: **Measured False Positive rate by Filters Compared to the Target False Positive Rate** ($\epsilon = 0.01$). The black dotted line shows the target false positive rate $\epsilon = 0.01$. Each case was measured 10 times, and the average was presented. The error bars, indicating the range of the samples, are presented in a black vertical line. The variables were set according to Table 4.9.

Figure 5.8 illustrates the measured false positive rate compared to the target false positive rate $\epsilon = 0.01$. As stated in Section 5.1, the Bloom filter has slightly over the target false positive rate, due to the design depending on incorrect false positive possibility calculation, as it assumed the likelihood of each bit set to 1 is independent. However, Blocked Bloom filter stay slightly under the target rate, as the filter divides the Bloom filter to small independent blocks, which distributes items, leading to less effction on of the gap of the dependence of each bit. The fingerprint based filters, which are Cuckoo and XOR filters, show that their actual false positive rates are under the target false positive rate by a significant margin. This is due to the size of the fingerprint, as increasing one more bit in the fingerprint halves the possibility of false positive. Therefore, fingerprint based filters need to be benchmarked with the target false positive rate being power of 2, to observe that these filters actually are bounded tightly. Therefore, we have tested on the target false positive rate set to $\epsilon = 0.016$, as it is the rounded value of $2^{-6} = 0.015625$ and we demonstrate the result in Figure 5.9:

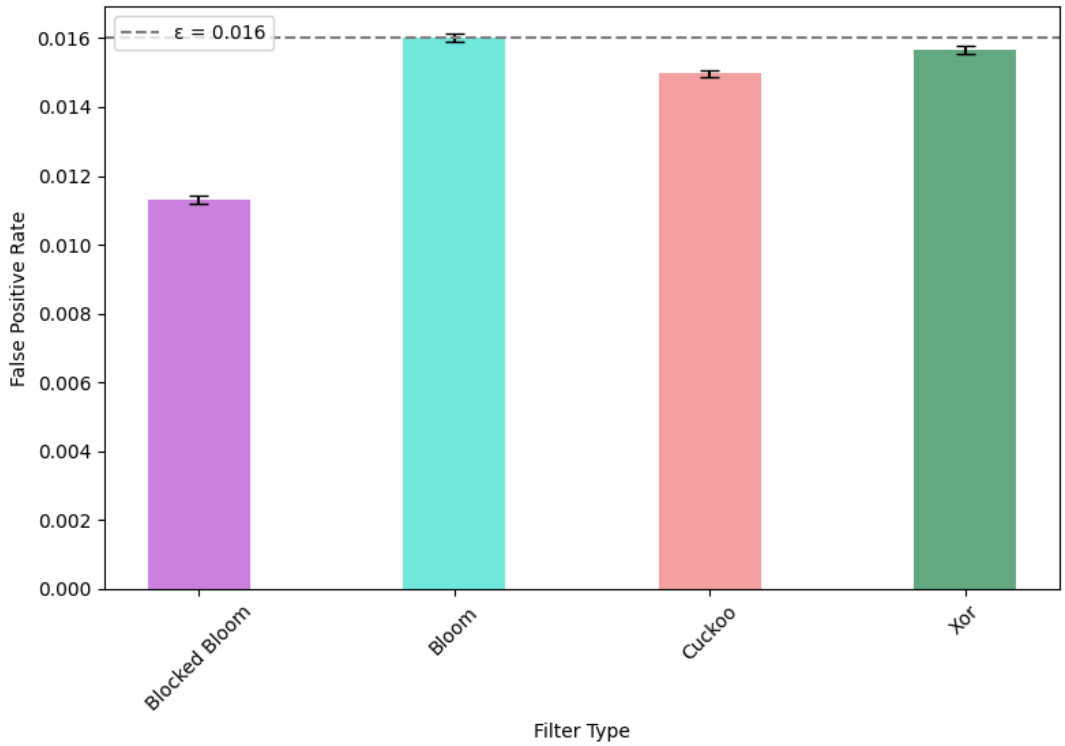

Figure 5.9: **Measured False Positive rate by Filters Compared to the Target False Positive Rate ($\epsilon = 0.016$).** The black dotted line shows the target false positive rate $\epsilon = 0.016$. Each case was measured 10 times, and the average was presented. The error bars are shown in black lines. The variables were set according to Table 4.9.

As shown, if the target false positive rate is close to the power of 2, the fingerprint based filters tend to be tightly bound to the target rate. Therefore, we can see that the measured false positive rate in the fingerprinted based filter is consistent if the target rate is bounded by k where $\frac{1}{2^{k+1}} < \epsilon \leq \frac{1}{2^k}$. Therefore, we can conclude that fingerprint based filters are over-compensating for space when the target rate is slightly smaller than power of 2, as they are required to increase the bit by 1 in order to achieve the target rate, which will have a significant impact in terms of space usage.

5.2.3 Construction Time

Filter Type	ϵ	Construction Time per Key (ns)	S.D.
Blocked Bloom	0.001	15.05	4.14
Bloom	0.001	26.93	8.15
Cuckoo	0.001	175.23	20.9
XOR	0.001	602.97	68.91
Blocked Bloom	0.01	7.78	1.08
Bloom	0.01	19.29	8.74
Cuckoo	0.01	123.74	7.70
XOR	0.01	510.28	32.80
Blocked Bloom	0.1	3.99	2.10
Bloom	0.1	7.84	0.15
Cuckoo	0.1	119.30	14.15
XOR	0.1	645.89	163.32

Table 5.6: **Average Construction Time per Key to Target False Positive Rate (ϵ).** Each cases were tested 10 times, and each of the average and the standard deviation were presented. The variables were set according to Table 4.10.

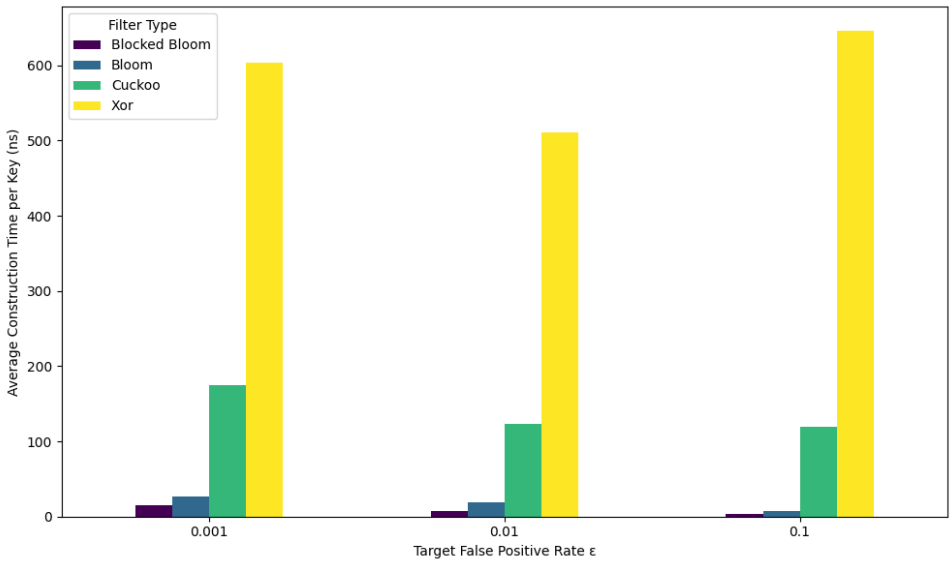

Figure 5.10: **Average Construction Time per Key when Inserted $n = 1,000,000$ Elements**. This graph is a simple visualisation where the data was collected from Table 5.6.

.

Table 5.6 and Figure 5.10 illustrate the average construction time per key by different target false positive rate, ϵ. First glancing the table and the figure, the most noticeable part is that the XOR filter takes huge computational overhead throughout all ϵ. This is due to the peeling algorithm, as the filter first has to assign every item into a slot without any collision. Even though rehashing does not occur due to low construction failure rate, unlike other filters, the filter involves additional counter for each slot to keep track of hash assignments. This is very costly, as it not only requires additional computation of adding and deleting, but also at most $r \cot n$ cache miss could occur at the worst case in accessing the number in counter. Also, holding a counter requires additional space in runtime, which will even increase the possibility of cache miss as essential memories, such as the filter itself and the key set can easily get evicted from the cache by the memory held by the counter. Even after the peeling algorithm has terminated, the filter determines the order by pushing the items into stack, which consumes additional time as well. The evidence of multiple number of cache misses occurring can be also spotted as a form of high standard deviation, where the general construction time per key lies further from the average, which indicates that the cache misses might occur in a lot of cases.

Also, Cuckoo filter consumes quite a lot of time, compared to Blocked Bloom filter and Bloom filter. Even though the Cuckoo filter adapts partial-key hashing to keep their two hash values close to each other which benefits from two buckets being physically close to each other, the value was high enough for exploring the reason. The high construction time is due to searching empty spaces inside of a bucket, which can be at most b slot per bucket. In addition, if the bucket is full, the filter has to access another bucket determined by partial-key hashing, which could introduce a cache miss. Obviously, another b searches of slot can occur here. Last but not least, in worst case, one of the keys might get evicted, which would repeat the process of new insertion. Therefore, Cuckoo filter has relatively high construction time per key, and also a high standard deviation is observed compared to Blocked Bloom filter and Blocked Bloom filter.

Finally, in cases of Blocked Bloom filter and Bloom filter, we can see that the motivation of Blocked Bloom filter, which is to decrease at most r cache misses generated from Bloom filter, is achieved. In every situation, Blocked Bloom filter excels the construction time per key, which makes the operation even faster. These two filters also demonstrated a linear decrease of time as ϵ increases in log scale, as the size of the filter is dependent on the log scale of ϵ.

5.2.4 Lookup Time

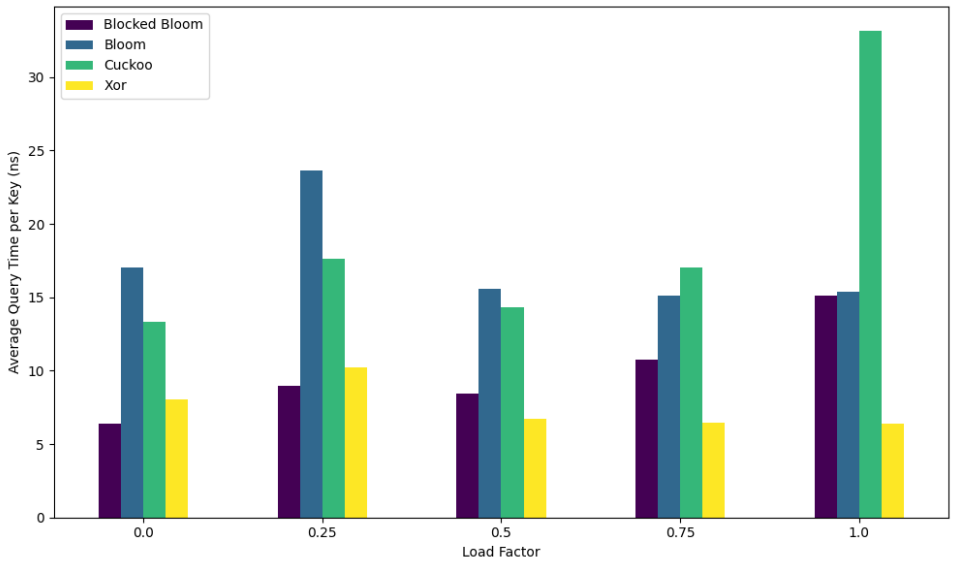

Figure 5.11: **Average Lookup Time per Key on Inserted Items in terms of Load Factor** α. Each case was measured 10 times, and the average was presented. The variables were set according to Table 4.11. The standard deviation for each filter was 4.22, 7.10, 7.88 and 4.29, for Blocked Bloom, Bloom, Cuckoo and XOR, respectively.

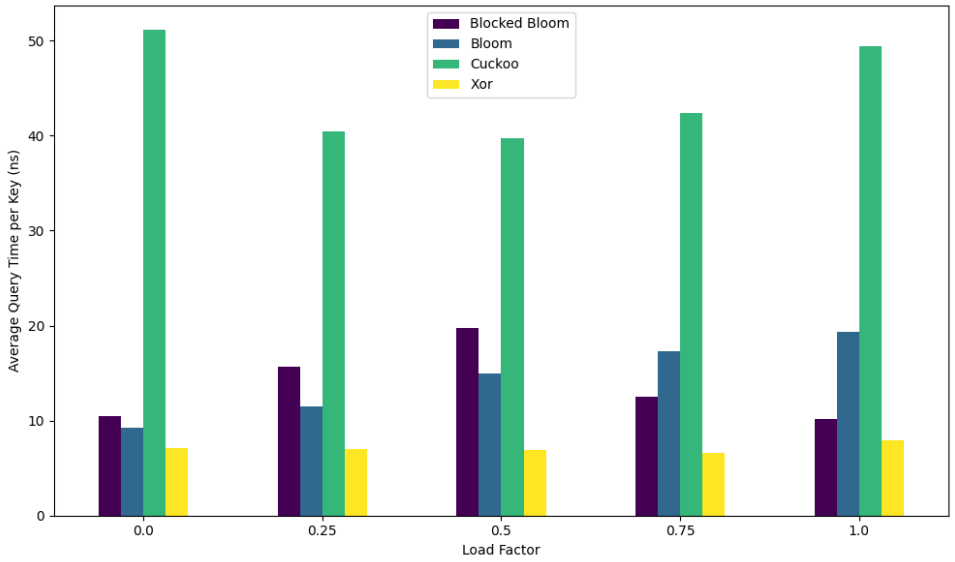

Figure 5.12: **Average Lookup Time per Key on Half of Inserted Items and Half of Items that were Never Inserted in terms of Load Factor** α. Each case was measured 10 times, and the average was presented. The variables were set according to Table 4.11.The standard deviation for each filter was 8.29, 3.89, 5.95 and 1.05, for Blocked Bloom, Bloom, Cuckoo and XOR, respectively.

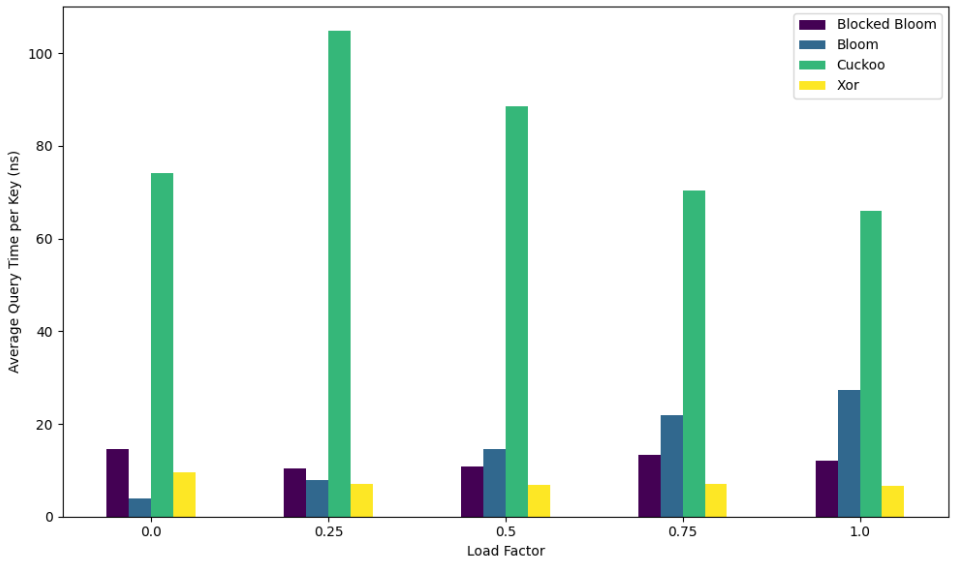

Figure 5.13: **Average Lookup Time per Key on Items that were Never Inserted in terms of Load Factor** α. Each case was measured 10 times, and the average was presented. The variables were set according to Table 4.11. The standard deviation for each filter was 2.69, 8, 92, 29.53 and 1.96, for Blocked Bloom, Bloom, Cuckoo and XOR, respectively.

Figure 5.11 offers the average lookup time per key in terms of load factor α, where each lookup is a positive query, meaning that we query the keys that have been inserted into the filter in advance. Following the discussion of Cuckoo filter in Construction time, the positive lookup time of Cuckoo filter is also expensive compared to other filters. The reason is the same as in the case of construction time, as the filter has to search at most $2b$ slots, comparing each fingerprint to the target item's fingerprint. In the case of Bloom filter, it also shows high lookup time especially on low load factors. This is due to cache misses occurring, and since this is a positive lookup, compared to negative lookup, the filter must access all r positions as every bits must be set to 1, and the filter has to carry on AND operation continuously. In contrast, Blocked Bloom filter and XOR filter has shown fast lookup times consistently, as they require at most 1 and 3 cache misses, and their operation on lookup is independent to the load factor.

Compared to the positive lookup, Figure 5.13 illustrates average lookup time per key in terms of α, but each lookup being a negative query. Similar to positive lookup, Cuckoo filter has the most expensive query time. Here, the query time takes even longer compared to the one on positive query. Compared to the situation where Cuckoo filter can return true whenever searching between slots in positive lookup, in negative lookup, the filter must check all $2b$ slots, except for a small possibility of false positives. The obligation of checking all the slots required increases the possibility of cache misses, as it must check 2 buckets. In contrast, Bloom filter shows decreased time in negative lookup compared to the positives, since Bloom filter can return false whenever encountering the required bit set to 0. The linearity is also shown, as the possibility of the filter encountering 1 in an array increases when the load factor increases. In the cases of Cuckoo filter and Blocked Bloom filter, as they are irrelevant with load factor and types of lookup, they remain consistent, similar to positive lookup.

Figure 5.12 illustrates the average query time per key, but a mixture of inserted and not inserted keys, as the features and the actual time appear as a mixed behaviour of positive and negative lookups.

In conclusion, Bloom filter is a better option in regard to high target false positive rates, $\epsilon \geq 0.01$, as it satisfies low construction time and query time accompanied with using relatively less space. If the users desires an even faster operation time in general, selecting Blocked Bloom filter could be a good option, but there is a risk of using 30% more space. On the other hand, if the space usage and the functionality after construction ($lookup(k)$) is prioritised, XOR filter might be suggested. Finally, Cuckoo filter may be advisable if the target false positive rate is low ($\epsilon < 0.01$), as it benefits space usage with the operation

time being a moderate operation time.

Chapter 6

Conclusion

This research started from the introduction which emphasized the need to store large datasets efficiently, leading to the research of Approximate Membership Query. We have offered an exhaustive background on several AMQ filters, Bloom, Blocked Bloom, Cuckoo and XOR filters, focusing on space/time and accuracy trade-offs of each filter. The goals for the thesis were to analyse the theoretical performance, implementing them in a controlled environment, and testing their efficiency.

We have successfully analysed the space complexity and false positive rates of the selected filters theoretically. For the fields which can be only determined empirically, we have validated the output of the former research. Notably, our project contributed to bridging the gap between theory and practice, providing critical insights into optimisations of these filters. In addition, our empirical testing on intrinsic parameters was able to explain the practical limitations.

However, there were some challenges of implementing it in practice. For example, the Cuckoo filter, which was found to have superior performance in theory, showed high dependence on the implementation of the Cuckoo hash table. Also, we were not able to provide exact values via our experiments as the output always varied due to the external factors such as hardware conditions. In addition, due to lack of time, we couldn't perform the demonstration of actual cache misses occurring, by using the profiler.

The research could be extended to offer an alternative type of AMQ, by implementing the strengths and modifying the weaknesses from the existent filters. Also, the current implementation which is optimised to our environment, can be extended to support a cross-platform. Lastly, deeper analysis on hardware memory by using CPU profiles could enrich the analysis.

Bibliography

[1] bitcoinj. https://bitcoinj.org/. Accessed: 2024-5-8.

[2] Crypto++ 5.6.0 benchmarks. http://www.cryptopp.com/benchmarks.html. Accessed: 2024-4-11.

[3] Operating Modes — Bitcoin. https://developer.bitcoin.org/devguide/operating_modes.html. Accessed: 2024-5-8.

[4] Dimitris Achlioptas and Michael Molloy. The solution space geometry of random linear equations. *Random Struct. Algorithms*, 46(2):197–231, March 2015.

[5] Mohammad Al-hisnawi and Mahmood Ahmadi. Deep packet inspection using Cuckoo filter. In *2017 Annual Conference on New Trends in Information & Communications Technology Applications (NTICT)*, 2017.

[6] Austin Appleby. Smhasher. https://github.com/aappleby/smhasher/tree/master. Accessed: 2024-4-11.

[7] Claude Berge. *Graphs and Hypergraphs*. North-Holland Publisihing Company, 1973.

[8] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[9] Jonas Bonér. Latency Numbers Every Programmer Should know. https://gist.github.com/jboner/2841832. Accessed: 2024-5-4.

[10] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrah, Sushil Singh, and George Varghese. Beyond bloom filters. In *Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, August 2006. ACM.

[11] Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. On the false-positive rate of Bloom filters. *Inf. Process. Lett.*, 108(4):210–213, October 2008.

[12] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet Math.*, 1(4):485–509, January 2004.

[13] Thomas H Cormen and Charles E Leiserson. *Introduction to Algorithms, fourth edition*. MIT Press, London, England, April 2022.

[14] cppreference.com. std::uniform_int_distribution. https://en.cppreference.com/w/cpp/numeric/random/uniform_int_distribution. Accessed: 2024-5-1.

[15] M Dekking. *A Modern Introduction to Probability and Statistics: Understanding Why and How*. Springer, 2010.

[16] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership (extended abstract). *Automata, Languages and Programming*, pages 385–396, 2008.

[17] Peter C Dillinger and Stefan Walzer. Ribbon filter: practically smaller than Bloom and Xor. 2021.

[18] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, December 2014.

[19] Afton Geil, Martin Farach-Colton, and John D Owens. Quotient filters: Approximate membership queries on the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018.

[20] Thomas Mueller Graf and Daniel Lemire. Xor filters. *J. Exp. Algorithmics*, 25(1):1–16, December 2020.

[21] Thomas Mueller Graf and Daniel Lemire. Binary fuse filters: Fast and smaller than xor filters. *J. Exp. Algorithmics*, 27:1–15, December 2022.

[22] Ronald M Graham, Donald E Knuth, and Oren Patashnik. *Concrete Mathematics, second edition.* Addison-Wesley, 1994.

[23] John L Hennessy and David A Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, 4 edition, January 2006.

[24] Dmytro Ivanchykhin, Sergey Ignatchenko, and Daniel Lemire. Regular and almost universal hashing: an efficient implementation. *Softw. Pract. Exp.*, 47(10):1299–1323, October 2017.

[25] Jiayang Jiang, Michael Mitzenmacher, and Justin Thaler. Parallel peeling algorithms. *ACM Trans. Parallel Comput.*, 3(1):1–27, June 2016.

[26] Shachar Lovett and Ely Porat. *A lower bound for approximate membership data structures.* IEEE 51st Annual Symposium on Foundations of Computer Science, 2010.

[27] Constantin Pohl, Kai-Uwe Sattler, and Goetz Graefe. Joins on high-bandwidth memory: a new level in the memory hierarchy. *VLDB J.*, 29(2-3):797–817, May 2020.

[28] Elakkiya Prakasam and Arun Manoharan. A cache efficient one Hashing Blocked Bloom filter (OHBB) for random strings and the k-mer strings in DNA sequence. *Symmetry (Basel)*, 14(9):1911, September 2022.

[29] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *J. Exp. Algorithmics*, 14:4.4, December 2009.

[30] Pedro Reviriego, Jorge Martinez, David Larrabeiti, and Salvatore Pontarelli. Cuckoo filters and bloom filters: Comparison and application to packet classification. *IEEE Trans. Netw. Serv. Manag.*, 17(4):2690–2701, December 2020.

[31] Pedro Reviriego, Alfonso Sanchez-Macian, Elena Merino-Gomez, Ori Rottenstreich, Shanshan Liu, and Fabrizio Lombardi. Attacking the privacy of approximate membership check filters by positive concentration. *IEEE Trans. Comput.*, pages 1–12, 2022.

[32] Yekang Zhao, Wangchen Dai, Shiren Wang, Liang Xi, Shenqing Wang, and Feng Zhang. A review of Cuckoo filters for privacy protection and their applications. *Electronics (Basel)*, 12(13):2809, June 2023.