Project Report

Evaluating the Performance of Approximate Membership Queries

Ahmed El-farra, Ahmed Faid, Issar Farid, Muhammad Zakar
aelfarr2@uwo.ca, afaid@uwo.ca, ifarid@uwo.ca, mzakar@uwo.ca

The University of Western Ontario
Department of Computer Science

1 Introduction

Many software applications are frequently interested in solving the membership problem; that is, they want to know if a datum exists in a very large data set. These applications range from user verification, routing-table look up, DNS look up, and database query processing. This problem can be solved using many data structures. For example, a dictionary can provide a fast solution with $O(n)$ time complexity and a 100% accuracy rate. However, this approach comes with a substantial memory utilization trade-off, growing linearly in relation to the querying set [1]. Approximate Membership Queries (AMQs) are a solution to the membership problem focused on balancing this trade-off.

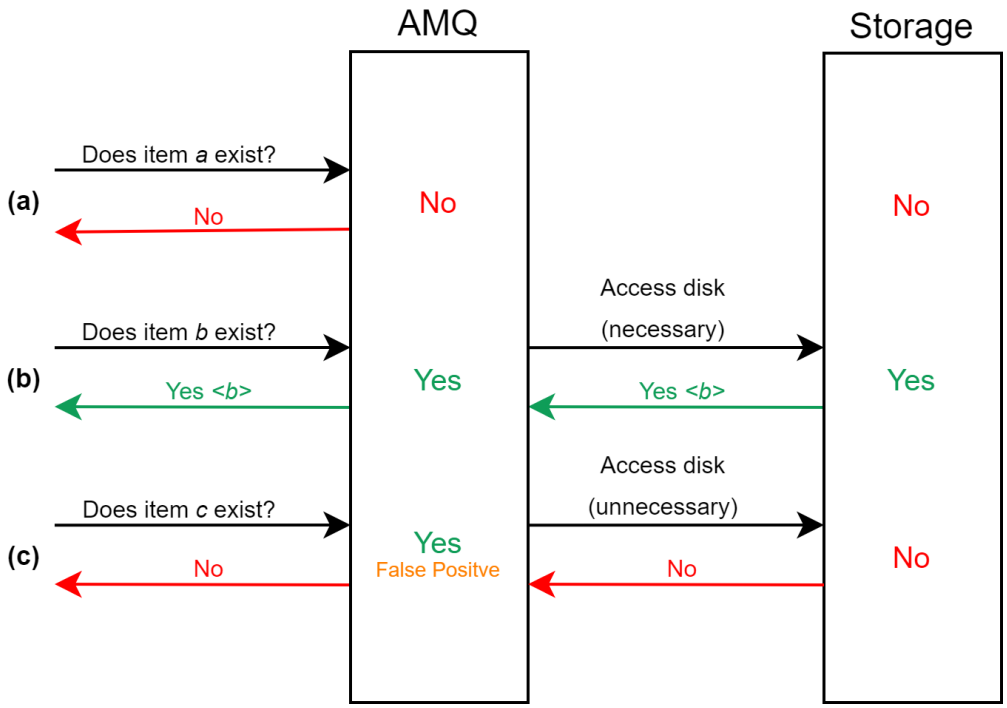

Figure 1. A generalized usage scheme for AMQs as an intermediary between an application and the physical storage. The figure highlights when: **(a)** both the AMQ and storage do not contain an item, **(b)** both the AMQ and storage do contain an item, and **(c)** the AMQ produces a false positive and initiates an unnecessary disk access. Adapted from [2].

Approximate Membership Queries, such as Bloom filters, are probabilistic data structures that support lookup and update operations on a set of keys [3]. AMQs leverage accuracy for space and time efficiency, making them a better solution for the membership problem than Exact Membership Queries (EMQs) such as dictionaries when dealing with very large amounts of data. Figure 1 showcases the generalized usage scheme for AMQs as an intermediary between an application and the physical storage. AMQs are more space efficient than EMQs because they often do not store the elements themselves; instead they generally store a representation of the element such as a hash.

Furthermore, as Figure 1 highlights, AMQs are more efficient in terms of time than EMQs because they eliminate most unnecessary disk accesses.

Many different types of AMQs have been designed and implemented over the years, each with its own advantages and disadvantages in different applications. In this project, we investigated AMQs such as Bloom, cuckoo, and quotient filters by evaluating their performance and accuracy in different applications under various constraints. Through evaluating these three AMQs, we will cover a wide range of applications and, ultimately, improve the current understanding of AMQs and provide a basis for discussion on the advantages and disadvantages of AMQs.

2 Implementation and Methodology

Evaluating the performance of each AMQ in a uniform manner is challenging due to intrinsic differences in how they function. Specifically, the behaviour of each filter is controlled by different parameters. Certain aspects such as the memory usage (size) cannot be experimentally controlled directly between filters. As a result, we adopted hybrid evaluation approach where core aspects of each filter are tested individually and a cross-filter analysis is performed using these results.

2.1 Filter Implementation

Since the project is focused on the performance behaviour of Bloom, cuckoo, and quotient filters, we first implemented the filters from scratch in Python. We implemented these filters from scratch to avoid any implementation dependent factors such as performance enhancements from using another library. This choice helps us to avoid any bias when testing the performance of these filters. Additionally, by using our own implementations, we can easily access details such as memory utilization which are often hidden by external libraries.

Our Bloom filter implementation used the `BitVector` library to provide the underlying bit array data structure. Fixed or growable size containers used by any filter used Python's built in `list` data type. The `mmh3` library that provides the MurmurHash3 function was used for all hashing operations by the filters.

2.2 Testing Methodology

We used a data set of 26 million Reddit usernames for all of our testing. This data set was chosen as the problem of identifying whether a certain username is available (when a new user wishes to register) perfectly models the membership problem. Additionally, by utilizing real user data, we are able to showcase the tangible benefits of AMQs for many practical applications. We extracted the data into several files containing 1, 5, 10 million usernames each. Also, to compare AMQs to standard database systems, we built a sqlite database containing 10 million usernames with a clustered index on usernames.

This data was used to run a set of comprehensive tests to analyze the parameters of the various AMQs under different constraints. The most important aspect that we tested was false positive rate. Our calculation method for false positive rate was very straightforward. At the start of any test, we provisioned a certain set of usernames as known to be uninserted. After inserting the required amount of usernames, we queried the AMQ on whether it contained the uninserted usernames. The amount of such queries that returned true were identified as false positives and the false positive rate was calculated as the number of such false positive queries divided by the total number of uninserted username queries. For tests that interacted with the sqlite database, we ensured that the optimal query execution plan was used by manually inspecting the output of the `EXPLAIN QUERY PLAN` command. Finally, tests that involved measuring the performance in terms of time used Python's standard library utilities for measuring the process time.

We focused on using a large portion of our overall data set for all of our testing. This methodology allowed us to focus on the asymptotic behaviour of the AMQs and ultimately made our results more meaningful. On average, tests used AMQs that contained at least 1,000,000 usernames. However, in some cases, we decreased this amount to improve the running time of tests. Without this limitation, some of the tests would require several hours to run.

3 Bloom Filter

Bloom filters, invented in 1970 by Burton Howard Bloom, are space-efficient probabilistic data structures used to solve the membership problem [4]. Initially, Bloom filters were invented for applications where typical error-free hashing

techniques on the data would require an impractically large amount of memory. Bloom showcased the technique using a program for automatic hyphenation on a dictionary with 500,000 words where 90% followed a few simple rules and the remaining 10% required expensive dictionary lookups (disk accesses) to retrieve specific hyphenation patterns [5]. If this dictionary could fit into memory, then an error-free hash could eliminate all unnecessary disk accesses. However, as is the case for most meaningful applications, if this dictionary is too big to fit into memory, then Bloom's technique could use a smaller hash area (for the 10% of words that can be falsely identified) to still eliminate most unnecessary disk accesses.

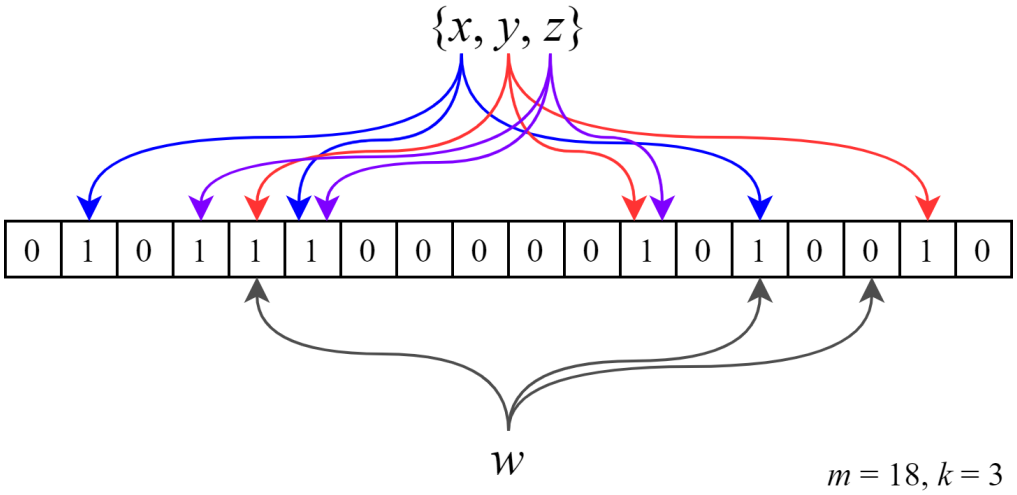

Figure 2. An example Bloom filter with $m = 18$ and $k = 3$ that contains the elements x, y, and z. Arrows represent the bit array positions that each element is mapped to by the three hash functions. The element w is not in the filter because one of its respective bit array positions is 0. Adapted from [6]

Bloom filters are still widely used in various applications such as Google Chrome's malicious URL lookup, internet cache protocol, and wallet synchronization in Bitcoin [7]. At their core, Bloom filters are an array of m bits and k different hash functions that map a given element to one of the m array positions, ideally generating a uniform random distribution [8]. Typically, m and k are determined from the target false positive rate p. Figure 2 showcases how elements are mapped to bit array positions. Elements are inserted into a Bloom filter by setting the bit positions returned by each of the k hash functions to 1. Consequently, an element is contained inside a Bloom filter if all of its corresponding bits (as returned by the hash functions) are set to 1. Algorithm 1 describes the Bloom filter insertion and lookup procedures further. Finally, Bloom filters do not support deleting elements since it is impossible to determine whether a certain bit was also set by another element from just the bit array and hash functions. The effects of a delete operation can be mimicked by recreating the filter afresh. However, this workaround is clearly expensive and likely only worthwhile once the collection of elements has changed significantly.

Algorithm 1 Bloom Filter Insertion and Lookup

```
1: procedure BLOOM-INSERT(B, x)
2:     for i from 1 to k do
3:         B[k_i(x) mod m] = 1
4: procedure BLOOM-CONTAINS(B, x)
5:     for i from 1 to k do
6:         if B[k_i(x) mod m] ≠ 1 then
7:             return false
8:     return true
```

Counting Bloom filters allow for deletion by extending the array buckets from a single bit to a multibit counter [9]. As such, insert operations increment the value of a element's respective buckets, lookup operations check that the buckets are not zero, and delete operations decrement the buckets. One side effect of this variant is that the counter can overflow. As a consequence, to correctly choose the number of bits used for a counter, the maximum number of

elements expected to be stored in the filter must be determined in advance.

As evident by the insert and contains operations, Bloom filters are straightforward to implement. The most challenging implementation concern is ensuring that the k hash functions are different and independent. However, most programming languages already provide strong hashing libraries. Additionally, with larger m and k values, the independence requirement between hash functions can be relaxed with negligible increases in the false positive probability [10]. Our Python implementation of a Bloom filter used the MurmurHash library and was less than 100 lines of code (including white space and comments). This simplicity accounts for the success and continued adoption of Bloom filters for a wide variety of applications.

3.1 Bits per Element

The primary advantage of Bloom filters over other data structures that represent a collection of items is their space efficiency. Although Bloom filters can produce false positives, they are significantly more efficient than data structures such as self-balancing trees, hash tables, and arrays or linked lists. While these data structures store at minimum the elements themselves and potentially auxiliary information such as pointers to other elements, Bloom filters do not store the elements at all. Instead, as highlighted earlier, Bloom filters (and AMQs in general) offset existing storage by acting as an intermediary to provide faster lookup and eliminate most disk accesses. A Bloom filter with a 1% false positive probability requires approximately 9.6 bits per element regardless of the number of elements. Furthermore, adding approximately 4.8 bits per element reduces the false positive probability by a factor of 10 each time.

Theoretically, the false positive probability of Bloom filters decreases exponentially as the bits per element increases. For a given false positive probability (p) and optimal number of hash functions, the optimal number of bits per element is given by

$$\frac{m}{n} = -\frac{\log_2 p}{\ln 2} \approx -1.44 \log_2 p \tag{1}$$

where n is number of inserted elements [8]. Equation 1 highlights that under an optimal number of hash functions, the required bits per element only depends on the target false positive probability.

We tested the false positive probability of Bloom filters while increasing the bits per element and using an optimal number of hash functions for the practically acceptable false positive probability of 0.01 (Figure 3). For each bits per element value, 5,000,000 usernames were inserted into an empty filter and 1,000,000 uninserted usernames were checked to calculate the false positive rate.

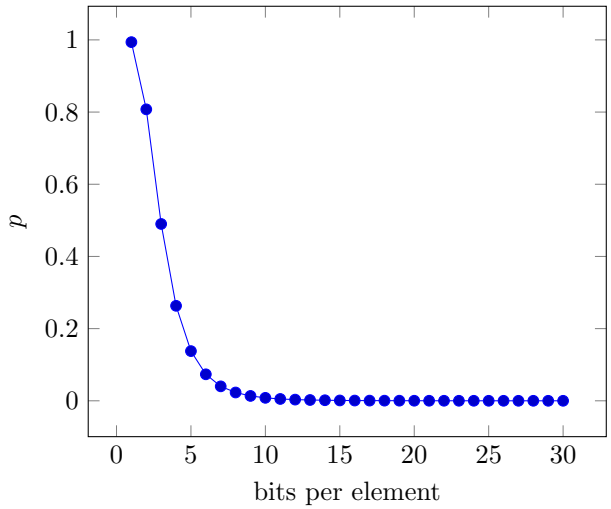

Figure 3. The false positive rate (p) of Bloom filters under a different number of bits per element using an optimal number of bits per element for a target p of 0.01.

Our experimental results agree with those predicted by the theoretical model. The equation predicts that for $p = 0.01$, the required number of bits per element is $\frac{m}{n} = -\frac{\log_2 0.01}{\ln 2} \approx 9.585$. Our results match this prediction such

that $p = 0.01$ is achieved between bits per element values 9 and 10. Furthermore, Figure 3 showcases that diminishing reduction in the false positive rate as the bits per element is increased past 10. In essence, there exists little benefit to increasing the bits per element past the required value. Instead, doing so, fails to leverage the excellent space efficiency of Bloom filters as compared to other data structures.

3.2 Hash Functions

Theoretically, the effect of hash functions on the false positive probability and performance of Bloom filters is well understood. The number of hash functions k that minimizes the false positive probability for a given m and n is $\frac{m}{n} \ln 2$ [8]. Consequently, assuming an optimal number of bits per element, the optimal number of corresponding hash functions in $k = -\log_2 p$ [8].

The above equation indicates that under an optimal bits per element, the required number of hash functions only depends on the target false positive probability. We tested the false positive probability while increasing the number of hash functions using an optimal number of bits per element for the practically acceptable false positive probability of 0.01 (Figure 4). For each k value, 1,000,000 usernames were inserted into an empty filter and 100,000 uninserted usernames were checked to calculate the false positive rate.

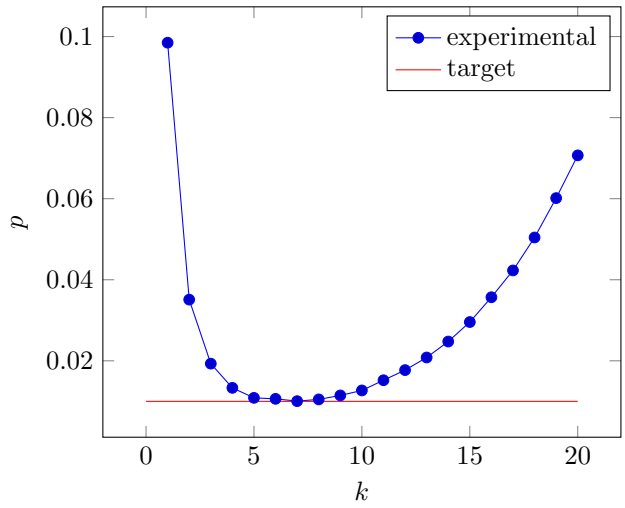

Figure 4. The false positive rate (p) of Bloom filters under a different number of hash functions (k) using an optimal number of bits per element for a target p of 0.01.

Our experimental results agree with those predicted by the theoretical model. The equation predicts that for $p = 0.01$, the optimal number of hash functions is $k = -\log_2 0.01 \approx 6.664$. Furthermore, increasing the number of hash functions past the optimal value results in a significantly worse false positive probability. Altogether, the number of hash functions serves to minimize the false positive probability to the target value and decreasing the false positive probability further requires increasing the number of bits per element.

3.3 Insert and Lookup Performance

Another advantage of Bloom filters is that the time required to insert or look up elements is constant. That is, as Algorithm 1 highlights, the cost of these operations depends only on the number of hash functions k. As such, the complexity of these operations is $O(k)$, which is independent of n (the number of elements in the filter). As a point of comparison, most self-balancing tree data structures have a element look up time of $O(\log n)$. Additionally, it should be noted that the k worst case look ups necessary for a Bloom filter look up operation are independent. As such, these can be performed in parallel to further improve performance.

We tested the insert and look up performance in terms of time while increasing the number of hash functions using an optimal number of bits per element for the practically acceptable false positive probability of 0.01 (Figure 5). For each k value, 750,000 usernames were inserted into an empty filter, 750,000 usernames were looked up such that

500,000 had been inserted into the filter and 250,000 were not inserted, and the process time was used to determine the performance.

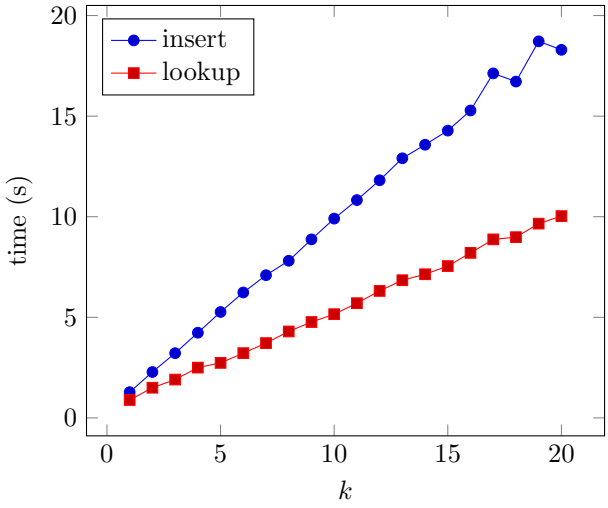

Figure 5. The time (in seconds) to insert and look up elements in a Bloom filter under a different number of hash functions (k) using the optimal bits per element value for a target false positive probability of 0.01.

Our results showcase the expected linear relationship between the number of hash functions and insert performance. Additionally, the look up time also increases linearly with the number of hash functions. However, observe that the look up time is bounded by the insert time since look up operations can return early if any of the k bits for an element are not set.

3.4 Membership Performance

The ultimate purpose of AMQs is to solve the membership problem by improving the look up performance of large data collections by practically eliminating the need for expensive disk accesses. Large databases are the most obvious applications that can benefit from the improved performance since most database operations require disk accesses. We tested the look up performance in terms of time of Bloom filters as compared to sqlite databases. 10,000,000 usernames were inserted into both the filter and the database. The Bloom filter used the optimal bits per element and hash function for the practically acceptable false positive probability of 0.01. The database was set up with a clustered index on username (using the `WITHOUT ROWID` sqlite keywords) and the look up query used the `EXISTS` to ensure an optimal execution plan was used. Figure 6 showcases the results as we increased the number of inserted usernames that were looked up.

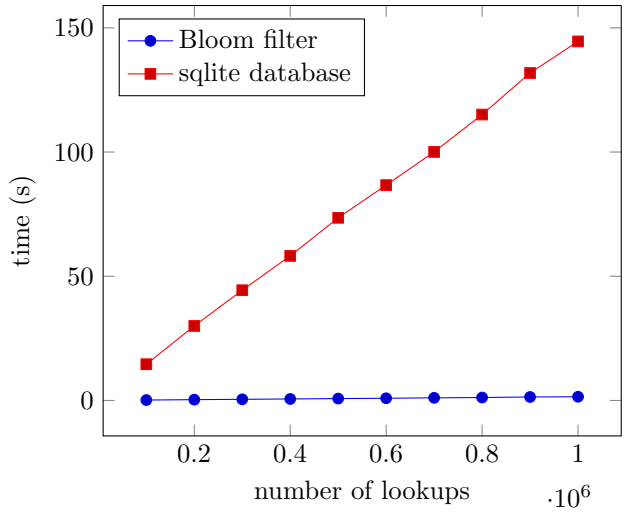

Figure 6. The time (in seconds) to look up usernames using a Bloom filter and a sqlite database that both contain 10,000,000 usernames. The Bloom filter was set up for a target false positive probability of 0.01 and the sqlite database used a clustered index.

Our results clearly highlight the advantage of Bloom filters (and AMQs in general) as compared to exact membership queries. For both the Bloom filter and the sqlite database, the time required increased linearly with the number of lookups. However, because Bloom filters are space efficient data structures they can fit into memory and eliminate disk accesses. As such, while querying the database for 1,000,000 usernames required 144.5 seconds, querying the Bloom filter only required 1.48 seconds. Although the Bloom filter will average a false positive rate of 1%, the additional overhead to support these false positives through unnecessary disk accesses is negligible compared to the overall cost of using exact membership queries.

4 Cuckoo Filter

The cuckoo filter Approximate Membership Query (AMQ) is a probabilistic data structure for efficiently checking if a record item is not part of the database. Cuckoo filters were created to improve on the weaknesses of the bloom filter while maintaining the same level of space efficiency. The improvements provide a higher level of control over the filter by offering a bounded false positive probability (FPP), dynamic insertions, and deletions of values in the filter [11].

4.1 Cuckoo Hashing

Cuckoo filters are designed using a minimized hashed table which resolves collisions with a type of open addressing called cuckoo hashing. Originating from 2001, cuckoo hashing resolves collisions by using two hash functions rather than a single one. When a collision occurs the value is hashed again using the second hash function, finding another location in the table for the key. The lookup for a value in the hash table requires a constant time complexity of $O(1)$, this holds true even in the worst case because at the very maximum, only two different locations in the table need to ever be visited for a lookup. In cuckoo hashing, if a key insertion faces a collision in both hashed locations, one of the keys in the two locations will be displaced and moved to its other possible location. However, the displaced key may also face the same issue when it's moved to the other location, where it might displace the key that already exists there resulting in a series of recursive displacements [12]. The process is repeated until each key finds a location on the table. As the table progressively fills up, an infinite loop of key displacements can occur when a new key insertion is made since there is not enough locations for all the keys in the hash table. The solution to this is forcing an insertion to fail once a certain number of repeated displacements are performed.

4.2 Cuckoo Filter Components

The cuckoo filter implements cuckoo hashing using a hash table that contains a bucket in each index of the table. The bucket is an array that can hold a set number of fingerprint values. Fingerprints are allocated with an f-byte

Algorithm 2 Cuckoo Hashing Insertion

```
1: procedure CuckooInsert(x)
2:     if B[hash1(x)] is not occupied then
3:         B[hash1(x)] ← x.
4:     if B[hash2(x)] is not occupied then
5:         B[hash2(x)] ← x.
6:     else
7:         y ← B[hash2(x)].
8:         B[hash2(x)] ← x.
9:         CuckooInsert(y)
```

amount of space to represent the value of an entry. This f-byte value is determined using the desired FPP for the cuckoo filter [11]. Reduction of the FPP can come at the cost of space efficiency since more bytes need to be allocated to represent each value, subsequently increasing the accuracy of the cuckoo filter. In addition, the f-byte size of the fingerprint also influences the amount of time needed to compare fingerprints. The more f-byte allocated to each fingerprint the longer the lookup will take. The use of fingerprints increases the space efficiency of the cuckoo filter since the whole value does not need to be stored in the bucket [13]. Part of the goal of our testing is to determine what parameter values and ratios are best suited given a condition where a cuckoo filter must be used.

4.3 Dynamic Deletes and Insertions

Dynamic deletion of values in cuckoo filters is executed by looking up both buckets that the value can be stored in, using the two hash functions. If the fingerprint in the bucket matches the fingerprint being looked up for deletion, it will be removed from the bucket. Dynamic deletion ensures that safety of other values in the cuckoo filter. Even when a delete is executed on a value x where there happens to be a value y stored in the same bucket with the same fingerprint, partial key cuckoo hashing *bucket* $i_2 = i_1 \oplus HASH(fingerprint)$ would ensure that y will remain safe from deletion. As mentioned before, the insertion of new values in cuckoo hashing can lead to previous values being displaced to their second location. However, this is more difficult in cuckoo filters because only fingerprints are stored, rather than the full value. This is circumvented using partial key cuckoo hashing *bucket* $i_2 = i_1 \oplus HASH(fingerprint)$ which calculates both buckets that the fingerprint can be stored in, allowing proper execution of cuckoo hashing key displacements [13].

4.4 Fingerprint size vs False Positive Probability

Cuckoo filters are applied to a DBMS to offer a fast and space-efficient look up to ensure a record item does not already exist in the database. However, one of the challenges is finding the balance of speed, space efficiency and the FPP required by the program. One component of the cuckoo filter that is used to find that balance is the fingerprint size, which is the number of f bytes allocated for the storage of each record value in the filter. The tests are conducted on a standard cuckoo filter loaded with an occupancy of 96.1% fingerprints, and this is derived using the a non-empty fingerprints and the t total capacity of fingerprints, $load = \frac{a}{t}$. The tests in figure 7 indicated that as the number of bytes f allocated to a fingerprint increases, the FPP rate decreases. As the number of f bytes increases, the space allocated to the cuckoo filter increases, $Space\ Allocated = f * t$. The results in figure 7 compare the experiment's results to the theoretical FPP p vs fingerprint size f results derived from the following equation $f = \frac{log(\frac{1}{p})+2}{load \times 8}$.

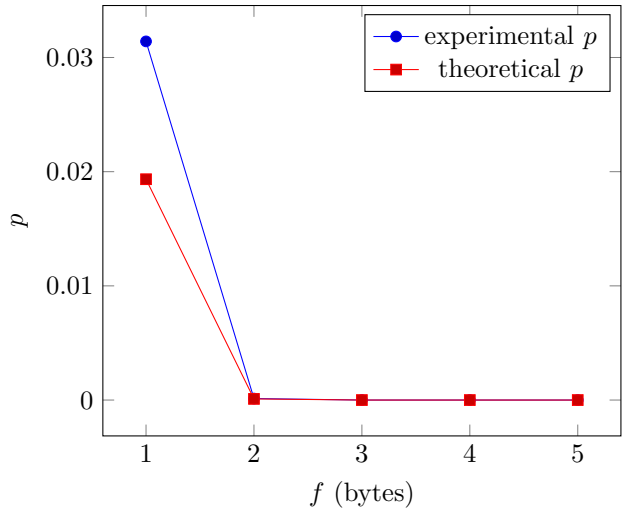

Figure 7. The false positive rate (p) of the cuckoo filter after 100,000 username insertions with a load of 96.1% under different fingerprint sizes (f). This is mapped against the theoretically expected p values under the same conditions.

4.5 Space Efficiency vs Bucket Size

Choosing the right parameters for a cuckoo filter can significantly improve its space efficiency. For the purposes of our testing, we define space efficiency as the number of bits required by a filter to represent a value. Space efficiency is calculated by dividing the total hash table size in bits by the number of values that can be effectively stored in the filter. It is important to note that space efficiency differs from fingerprint size in most applications. Although each bucket in a cuckoo filter's hash table can store one value in each fingerprint, not all buckets in the hash table are fully occupied. In many cases a hash table is left at a 80% or 50% load to achieve a desired false positive rate.

One of the parameters correlated with space efficiency in a cuckoo filter is bucket size. For instance, increasing the bucket size while keeping the size of the filter/fingerprint constant will result in a higher false positive rate[14]. This is because increasing bucket size increases the chance of a collision. Therefore, to maintain the false positive rate, a bigger fingerprint size is required. The relationship between fingerprint size, bucket size, and false positive rate is modeled by the equation:

$$\text{fingerprint size} \geq \log\left(\frac{1}{\text{false positive rate}}\right) + \log(2 \cdot \text{bucket size}) \tag{2}$$

To further illustrate this relationship, we ran an experiment to measure the false positive rate for a cuckoo filter by varying the bucket size while maintaining a constant cuckoo filter size/fingerprint size. Figure 8. illustrates the results of our experiments. The figure 8. shows a positive coo-relation between the size of the bucket and the false positive probability. This matches with the expected theoretical value since an increase in bucket size requires a corresponding increase in false positive probability for the inequality in Equation 2 to hold.

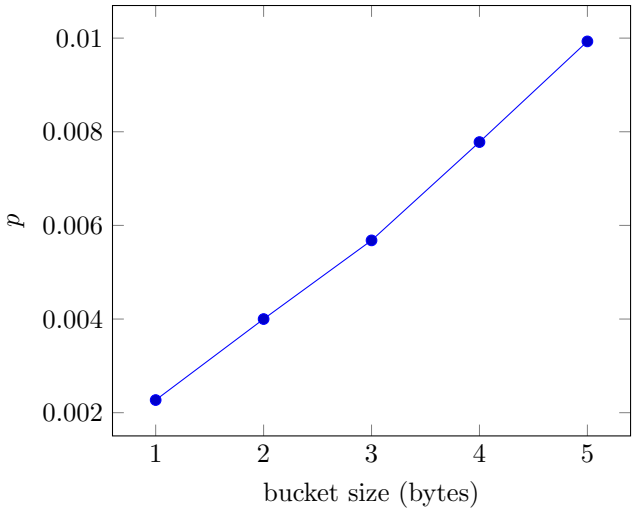

Figure 8. The false positive rate (p) of a cuckoo filter with varying bucket sizes. The cuckoo filter has a fixed size of 100,000 fingerprints with each fingerprint having a constant size of 1 byte. 25,000 items were inserted into the filter initially and 250,000 were used to calculate p.

4.6 Filter Size vs False Positive Probability

Another factor affecting the false positive probability of the cuckoo filter is the filter's size ratio, which is the ratio between the number of entries stored in the filter vs the number of entries the filter can hold. In our testing we define the size of the cuckoo filter to be the total number of fingerprints the filter consists of. To measure the effect the size ratio has on the false positive probability, we measure the false positive probability for cuckoo filters with varying size ratio allocations after inserting 100,000 entries. We chose the smallest size ratio to be a little greater than 1:1 ratio as cuckoo filters insertions become more and more costly once and failures becomes more likely when the filter approaches a 100% load[11].

Figure 9a shows the results of the experiment. Keeping the fingerprint size/fingerprints per bucket constant, there is a significant drop in the false positive rate going from a 1:1 to a 4:1 size ratio. The effect the filter's size ratio has on the false positive rate diminishes as the size ratio increases. Once we reach a size ratio of above 100:1, the false positive probability is negligible. Figure 9a clearly shows the trade-off between space and performance. While having the filter operate at a 2:1 or 4:1 load seems to be the optimal trade-off between space and performance, some applications might be very sensitive to false positives and require a much higher size ratio allocation.

Figure 9b models the same problem from a different perspective, looking at the false positive rate as a function of the filters load. We see that as the load approaches 100% the false positive probability steadily increases. The filter fails at a load of above 95% so we limited our testing to a maximum of 95% load. The results from this test mirror the results from our test on the effects of size ratio on false positive probability.

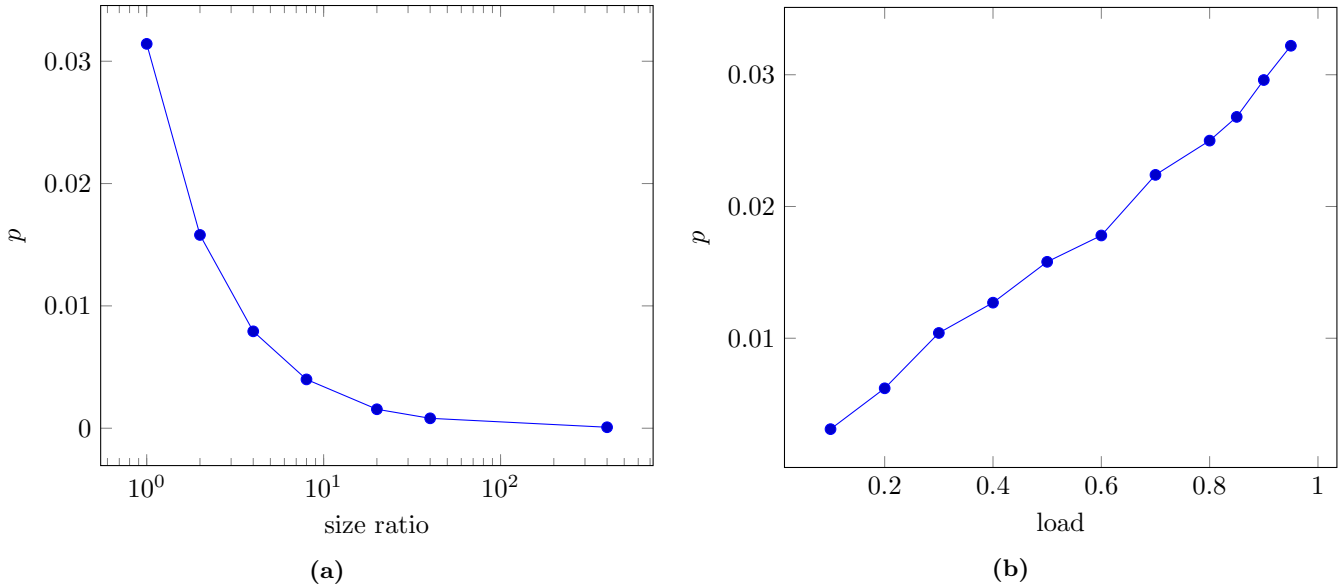

Figure 9. **(a)** The false positive rate (p) under different the filter sizes while keeping the number of entries (inserts) constant. For this experiment, 100,000 usernames were initially inserted into the filter and 1,000,000 were used to calculate p. **(b)** The false positive rate (p) when keeping the size of the filter constant but varying the filter's load. 100,000 usernames were used to calculate p.

5 Quotient Filters

One of the limitations of Bloom filters is their performance when they reach a high-capacity of elements in dynamic storage. One way to solve this problem is to physically store the data, but this solution has its own limitations as well. Inserting or looking up an element in a Bloom filter requires as many random accesses as we have hash functions. This means the performance of storing and querying through the data physically will significantly increase the memory and reduces the speed of the Bloom filter [15].

The quotient filter, introduced by Michael Bender et al. in 2011 is an efficient cached approximate membership query (AMQ) data structure, which tests whether the element is a member of a set. It is a solution to overcome the limitations of the Bloom filter that prevent it from being unable to be used with enormous data sets. The quotient filter uses only one hash function to handle a high collision probability, so when storing all the data, we can only perform a random access once. The quotient filter supports three basic operations; insertion of an element into the set, deletion of the element in the set, test when the element is a member or not a member of the set, this event is also known as a lookup. The filter is parameterized by the size of the fingerprint (in bits), also known as p, and the linear probing of the hash function that generates a hash fingerprint [15].

Quotient filter are also much more cache-friendly than Bloom filters by offering a significant increase in performance when stored on cache memory or even SSDs. One of the disadvantages of the quotient filters is its hash function algorithm which uses linear probing. When the quotient filter starts to overflow and the occupancy rate reaches 60%, the performance drops sharply. This is due to its hash function algorithm for linear probing becomes computationally expensive [16].

The quotient filter is used to index m - bucket arrays, where the remainder is stored. The hash table stores the remainder as the value and the quotient is implicitly stored as the bucket index. The algorithm first starts by generating a p-bit hash value for the element being inserted. Upper bits of the hash are used to determine the slot to use in the quotient filter. The remaining lower bits of the hash are stored in the identified slot [17]. Figure 10 uses an illustration to showcase how a quotient filter functions.

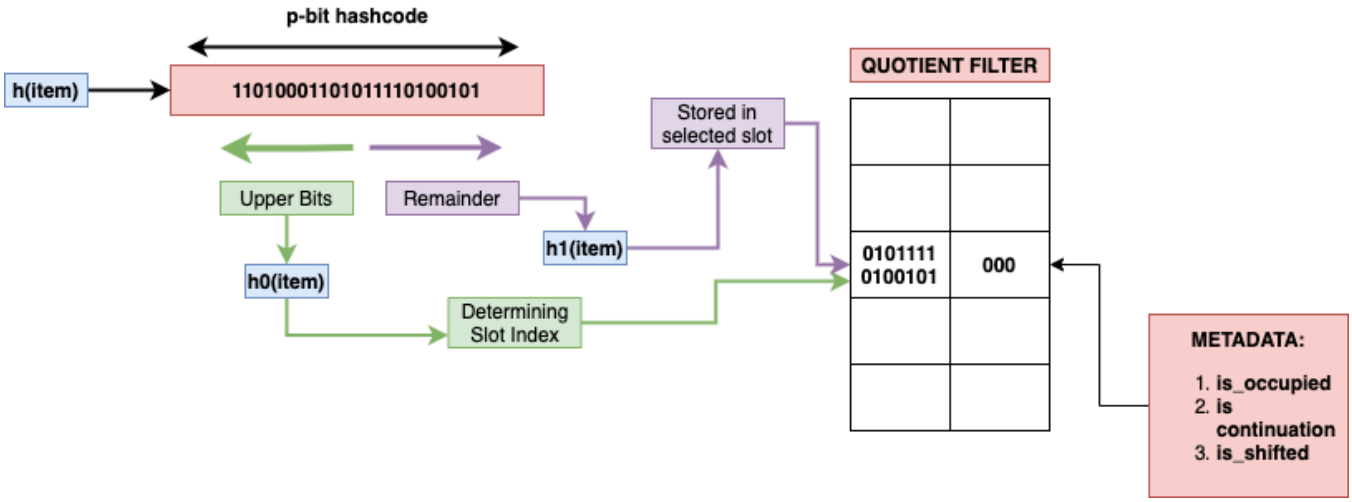

Figure 10. Quotient filters support insertion, deletion, lookups, resizing, and merging. The essence of the idea is depicted in the figure above.

A problem that occurs when more than one element are shared by the same hash upper bits is called a soft collision. The quotient filter handles this problem and tracks the three bits of information for each entry. Collisions are resolved using linear probing and 3 extra bits per bucket. The three situations for handling collisions are as follows [17]:

1. is occupied: whether a slot the canonical slot for some value currently stored in the filter.
2. is continuation: whether a slot occupied by a remainder that is not the first entry is a run.
3. is shifted: whether a slot holds a remainder that is not in its canonical slot.

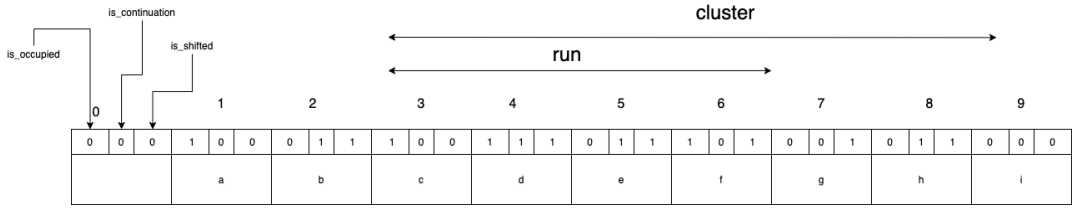

Figure 11. A 10 slot quotient filter as a hash table with chaining where the filter stores the set of fingerprints. Starting at the canonical slot, search to the left to find the beginning of the cluster then search to the right to find the item's run. The quotient filter stores the contents of each bucket in contiguous slots, the elements are shifted as needed, and three metadata bits are used to enable decoding.

The section will describe how the quotient filter performs several operations of this filter, such as insert, delete, lookup, and properties of quotient filters.

5.1 Lookups

To test whether an element is a part of a test, apply the hash function to the element to calculate its fingerprint which is commonly denoted as f. The next step is to calculate the quotient also commonly denoted as f_q, and the remainder of that fingerprint denoted as f_r for the fingerprint. We then proceed to check if the bucket f_q is not occupied, if it is not occupied, we will then return that the element is definitely not in the filter. If the bucket is already occupied, we will start the search from the leftmost slot to find the unset bit shifted by the bucket. Next, start searching from unset bit, and then decrement/increment the occupied or continuation bit. This is performed until the count reaches 0. We compare the remainder to the quotient and if it is true, return that the element is found [15].

Algorithm 3 Quotient Filter Lookup

```
1: procedure LOOKUP(S,inputs)
2:     for i from 1 to k do
3:         f_q ← f/2^r
4:         f_q ← modf/2^r
5:         if !is occupied(S[f_q]) then
6:             return FALSE
7:     for i from 1 to k do
8:         if is shifted(S[f_q]) then
9:             decrease f_q
       → search for slots in the run for f_r
10:    repeat
11:        if S[f_q] = f_r then
12:            return TRUE
13:    until is continuation(S[f_q])
14:    return FALSE
```

5.2 Insertions

The insertion is similar to the lookup, achieved by applying a hash function to the elements and calculating the quotient f_q and remainder f_r. The algorithm is the same as the algorithm in the search until we are sure that the fingerprint is definitely not in the bucket. In the current operation, keep the elements in sorting and inserting the remainder f_r to its occupied bit. Then, the remainders will be shifted forward at or after the selected bucket, followed by an update to the bucket bit [15].

5.3 Deletions

Deletes use an algorithm similar to that of the lookup. We divided the filter into independent cluster regions, similar to the regions used in the insert method, but modified: We require that the first slot in the cluster is occupied and not shifted. This event means the slot is actually the head of a cluster. We know that the value in this slot, and any shifted slots to its immediate right, will not be affected by any deletes to the left because the item is in its canonical slot. This event also prevents the cluster from only being composed of empty slots (no items to delete). We perform a bidding process to choose, which items to delete while avoiding collisions. Then, we delete one item at a time in each cluster, move the item to the left and modify the metadata bits as needed. After the process is finished, we delete the successfully deleted items from the queue and repeat [18].

5.4 Properties of Quotient Filters

The following list outlines several noteworthy properties of quotient filters:

- An estimated probability of a false positive occurring is

$$P(e \in S \mid e \notin S) \leq \frac{1}{2^r}$$

  The hash functions generate uniformly distributed fingerprints. This makes it possible to adjust the probability of the false positives that do occur. The probability is dependent on the size of the fingerprints reminder [15].

- The false positive rate of a linear probing quotient filter [19] with $\delta = n/m$ is

$$p^+ = \frac{E[\#comparsions]}{2^{r+3} - 1} = \frac{1}{2}(1 + \frac{1}{(1-\delta)^2})\frac{1}{2^{r+3} - 1}$$

- False negatives are not possible. The quotient filter returns the data item if it is not a member only if it's definitely not a member [15].

$$P(e \notin S \mid e \in S) = 0$$

- The length of the quotient filter of most runs is $O(1)$ and that the likelihood of that all runs have a length of $O(log(n))$ which makes this AMQ extremely fast [15].

- When scaling and using quotient filters concurrently we can use two quotient filters can be efficiently merged without affecting their false positive rates which cannot be done with Bloom filters [18].

6 Comparisons

In this section we combine our research and results to compare the various filters and offer potential guidelines for when each is most appropriate.

6.1 Bloom vs. Cuckoo Filter

In general, comparing the Bloom and cuckoo filters requires the consideration of many factors such as look-ups, insertions, deletion, false-positive probability, and space efficiency. Coming to understand the pros and cons of each filter in the given factors allows the programmer to decide which filter will best suit their implementation. Bloom filter look-ups require a constant $O(1)$ time complexity regardless of the number of hash functions dedicated to the filter. Cuckoo filters look-ups also require a constant $O(1)$ time complexity by only having to check up to two hash functions [13]. Even in the worst case, both bloom and cuckoo filters only require $O(1)$ time complexity to execute a lookup for a value. Another factor to address is the accuracy of the look-ups. A Bloom filter's hash table marks each position as empty or non-empty with a 0 or 1 respectively [4]. On the other hand, cuckoo filters store the fingerprint of a value within each bucket of the hash table. Although it's only a fingerprint of the value, it produces a higher level of accuracy when checking if the value already exists in the table compared to the empty/non-empty labels of the bloom filter. The cuckoo filter's use of fingerprints reduces the odds that different values overlap in the same of value at the hash table.

Bloom filter value insertions are executed by hashing the value with each of the hash functions to find the indexes in the hash table that will need to be flagged as non-empty. Cuckoo filter insertions are performed by applying up to two hash functions to store the fingerprint in a bucket. One of the challenges of cuckoo filters is accounting for the displacements made by cuckoo hashing, seen in the section 4.1's algorithm. As the filter becomes fuller, the number of displacements required to fit the fingerprints will increase [20]. Bloom filters and cuckoo filters deal with duplicate value insertions differently. Duplicate values in bloom filters don't require additional space since all the positions that the hash functions produce would already be marked as non-empty in the hash table. On the other hand, cuckoo filter duplicate values can take up two positions because the hash functions offer two possible locations for each fingerprint. If duplicate insertions are not handled, this could be one of the cuckoo filter's drawbacks. Bloom filters are limited because they cannot perform any deletes on any values that have been inserted. The system of marking hash table positions as empty/non-empty results in an overlap of values where deleting the positions for one value could impact many other values stored in the table. If a bloom filter needs to delete a set of values, it must rebuild the filter from scratch [4]. Cuckoo filters offer dynamic deletions that ensure the security of other fingerprint values using partial key cuckoo hashing $bucket\ i_2 = i_1 \oplus HASH(fingerprint)$ [13]. This can make the cuckoo filter very beneficial in implementations that require deletions since the filter won't have to be consistently rebuilt.

Another factor to consider when comparing bloom filters and cuckoo filters is space efficiency. For a false positive probability of less than 10%, the theoretical bound of the number of bits required to represent one entry is $log\frac{1}{FPP}$ [14]. On the other hand, the theoretical bound for the number of bits required by a bloom filter to represent an entry is 1.44 $log\frac{1}{FPP}$[13]. Therefore when the FPP $< 3\%$, Cuckoo filters are the more space efficient option between the two[14]. Looking at Figure 9b, we can see that even for loads as high as 90% the FPP for a cuckoo filter remains below 3%. From this we can conclude that for almost all applications Cuckoo filters require a lower number of bits/entry.

Another important factor to consider when comparing filters is the FPP. At low loads, Bloom filters have a slightly better performance than cuckoo filters in terms of FPP. However, as the load of the filter approaches its maximum capacity, cuckoo filter's FPP still remains relatively low (around 3%) while bloom filter's FPP increases exponentially. It is important to note however, that the theoretical capacity of the Bloom filter is at 100%, while for cuckoo filter a load above 95% usually results in an exponential increase in insertion time and failed inserts become more likely[14].

6.2 Bloom vs. Quotient Filter

The quotient filter has many advantages compared with the traditional Bloom filter, such as the ability to delete elements and adjust its own size. Similarly, two quotient filters can be effectively merged (the fingerprints in the

quotient filters are sorted) in a manner similar to the way they merge subroutine merges sort. Quotient filters have an advantage since they can scan elements and merge in sequential order which makes the quotient filter larger when storing the data physically. Items are saved in one place. Hence, the quotient filter is efficient when stored in main memory since it produces fewer cache misses than the Bloom filter. This advantage has a significant difference between the quotient filter and the random read/write access of the Bloom filter. This difference in Bloom filter becomes fairly large in contrast to quotient filters. Even if only part of the fingerprint is stored in the quotient filter, metadata bits can be used to recover the entire fingerprint, and the false positives can only occur at the same fingerprint/hash level. That is, only when two different keys generate the same fingerprint, the quotient filter may mistake them for the other. [21].

7 Conclusions

In this report we outlined increased performance and better space utilization as the motivation behind AMQs. Next, we supported their use by presenting several experiments that highlight the practical advantages of AMQs compared to EMQs. Finally, we performed a cross-filter analysis to showcase the differences between the investigated AMQs. We hope that this report serves as a basis for discussion and inspires future work on AMQs.

This report presents multiple avenues for future work. We discussed practical advantages of filters such as support for distributed computations through independent sub-operations as well as shortcomings such as Bloom filter's lack of support for element deletion. Specifically for Bloom filters, future work can focus on investigating the qualities and performance of Bloom filter variants as compared to the basic Bloom filter detailed in this report. For example, we discussed counting Bloom filters as a variant that support delete operations at the cost of increased space utilization. Future work could investigate the extent of this space utilization trade-off to outline when the support for delete operations is worth it. Furthermore, we believe that the potential parallelization of Bloom filters warrants further investigation. Specifically, the impact of parallelized operations on performance should be investigated from a practical standpoint. That is, whether the increased complexity and implementation concerns that arise from parallelization are worth it with the already fast performance of Bloom filters as compared to EMQs in mind. Additionally, decentralized usage of Bloom filters to compute aggregate functions should be investigated. Moreover, quotient filters are more cache-friendly than Bloom filters. This cache-friendly aspect has the potential to offer a significant increase in performance when stored on cache memory or even SSDs. However, there is a notable lack of research that investigates this characteristic.

Altogether, as evident by this report, AMQs are reliable data structures despite the false positives that arise from their probabilistic nature and their potential to offset an increasingly data driven world warrants further investigation.

References

[1] Elias Szabo-Wexler. Approximate membership of sets: A survey. 2014.

[2] Alexmadon. In memory bloom filter used to speed up access to a key value storage on disk. https://commons.wikimedia.org/wiki/File:Bloom_filter_speed.svg, 2010.

[3] 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)*, pages 451–462, 2018.

[4] Bloom filters – introduction and implementation, 2020.

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

[6] David Eppstein. Example of a bloom filter. https://commons.wikimedia.org/wiki/File:Bloom_filter.svg, 2007.

[7] OpenGenus Foundation. Applications of bloom filter.

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

[9] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, 2000.

[10] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. volume 4168, pages 456–467, 01 2006.

[11] Alex Chumbley and Christopher Williams. Cuckoo filter.

[12] Michael Mitzenmacher. Some open questions related to cuckoo hashing. 2009.

[13] David G. Andersen Bin Fan and Michael Kaminsky. Cuckoo filter: Better than bloom. page 5, 2013.

[14] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. CoNEXT '14, page 75–88, New York, NY, USA, 2014. Association for Computing Machinery.

[15] Andrii Gakhov. Probabilistic data structures. quotient filter. pages 1–4. gakhov, 2019.

[16] Colyer Adrian. A general purpose counting filter: making every bit count. The morning paper, 08 2017.

[17] Rob Johnson Bradley C. Kuszmaul Dzejla Medjedovic Pablo Montes Pradeep Shetty Richard P. Spillane Erez Zadok Michael A. Bender, Martin Farach-Colton. Don't thrash: How to cache your hash on flash. USENIX, 2011.

[18] John D. Owens Afton Geil, Martin Farach-Colton. Quotient filters: Approximate membership queries on the gpu. University of California, Davis, 2014.

[19] Robert Williger Tobias Maier, Peter Sanders. Concurrent expandable amqs on the basis of quotient filters. Karlsruhe Institute of Technology, 2012.

[20] Fast Forward Labs. Applications of bloom filter.

[21] B. H. Bloom. concept quotient filter in category algorithms. living book, 2018.