Filters

Filters are a class of data structures that support efficient *membership queries*. In other words, filters store approximate representations of sets, and filters let users quickly check whether an item is present in a set or not.

If a filter says that an item is not present in the set, the item is *definitely* not present in the set. In other words, "No" answers are always firm. On the other hand, filters generally allow for *false positives*. A false positive occurs when the filter says that an item is in the set, but the item is actually not present in the set. In many contexts, this is still useful behavior.

Filters are often used to save unnecessary work (typically expensive I/Os), and when a false positive shows up, the the result is just extra that would have been done anyway (in the absence of the filter).

This unit covers Bloom filters (the most popular and well-studied filter) and Quotient Filters (a filter that often yields better performance due to its cache-friendly access patterns).

Bloom Filters

Bloom filters are tuned based on three parameters

- **m** - the size of the bit array
- **k** - the number of hash functions
- **N** - the size of the set

False negatives are never tolerated, but for a given **N**, **m** and **k** can be chosen to minimize the false positive rate.

operations

To **insert** into a Bloom filter, you hash the element using each of the **k** independent hash functions. You then set the corresponding bits in the **m**-bit array.

To **query** a Bloom filter, you hash the element using each of the **k** independent hash functions. Then you check whether the corresponding bits in the **m**-bit array are set. The Bloom filter returns

- definitely *not* in the set if any of the bits are 0
- *possibly* in the set if all of the bits are 1

Note that the negative answer is conclusive, but the positive answer does not provide a guarantee. A Bloom filter query could return a *false positive* if some combination of other elements previously inserted into the array had caused the bits that correspond to the queried element's **k** hash function's bits.

Standard Bloom filters do not support delete operations. This is because the filter does not track which element is responsible for setting any given bit. If there is a collision at bit *i*, then reseting bit *i* will remove *all* elements with at least one hash function that sets bit *i*. There are Bloom filter variants, such as "counting Bloom filters", that support deletes. However, counting Bloom filters require additional metadata.

Quotient Filters (QFs)

Quotient filters (QFs) are essentially hash tables that combine linear probing with a few extra metadata-bits-per-bucket to manage collisions. However, as described in the paper, QFs Quotient filters do not store key-value pairs; instead, QFs just track set membership.

When storing an item *K*, the QF calculates *h = hash(K)*, and stores a subset of *h* in the array. When looking up a item *K*, the QF calculates *h = hash(K)* and looks for that subset of *h* in the array. Like Bloom filters, QF parameters can be adjusted to tradeoff memory consumption with accuracy.

Quotient filters rely on a technique called (wait for it...) *quotienting* to reduce the size of the hash value that the filter stores.

A quotient filter breaks a fingerprint into two parts:

- the most significant **q** bits (the quotient), and
- the least significant **r** bits (the remainder).

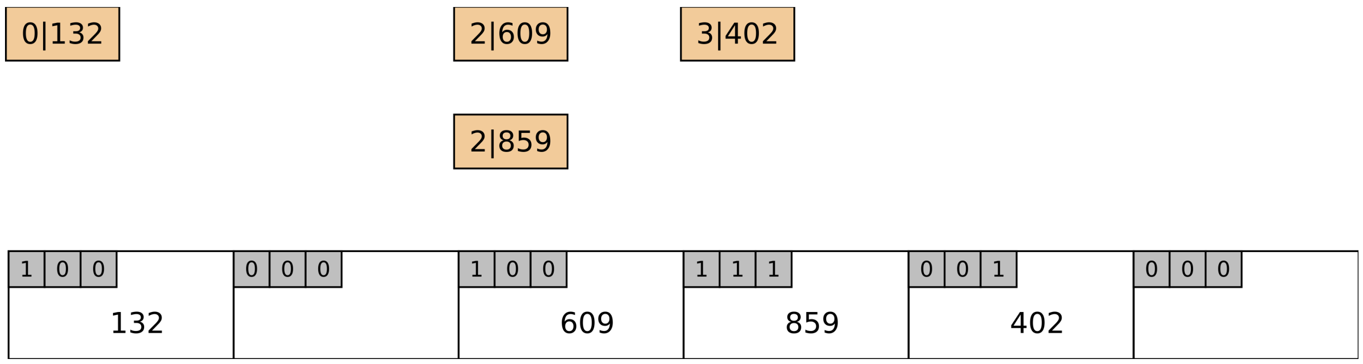
The quotient determines the fingerprint's index in the array, and it is never explicitly stored. The bucket indexed by an element's quotient is called its **canonical slot**, and the remainder is stored in a fingerprint's canonical slot (in the absence of collisions).

In the case of a collision, remainders are stored within a **run**. A run is a sorted list of remainders that share the same canonical slot. A maximal sequence of adjacent runs is called a **cluster**. Collision resolution can cause an entire cluster to shift --- including remainders that do not share canonical slots. In fact, only the first colliding run in a cluster will start at its canonical slot.

Three additional bits are kept per bucket in order to help resolve collisions:

- The *is_occupied* bit denotes that a bucket is the canonical slot for at least one fingerprint present in the quotient filter.
- The *is_continuation* bit signifies that a bucket stores an element of a run
- The *is_shifted* bit signifies that the element in the bucket is not in its canonical slot.

A quotient filter example can be seen in the image below, followed by an explanation that gives more details about the usage of these bits.



- The four members inserted into the quotient filter are each shown above their canonical slot, with quotient and remainder separated by a vertical bar (note the collision at the third canonical slot in the image, where 2|609 and 2|859 both map).
- If there is a `1` in the first metadata bit associated with a bucket, it signifies that the bucket is the canonical slot for some element in the quotient filter. The buckets at indexes 0, 2, and 3 are all canonical slots. The buckets at indexes 1, 4, and 5 are not.
- If there is a `1` in the second bit of a bucket's metadata, then the element stored in the bucket is part of a run. Bucket index 4 stores 859, which is shifted from its canonical slot, and is part of a run.
- If there is a `1` in the third bit of a bucket's metadata, then the bucket is not the canonical slot for the element it contains. Only elements 132 and 609 are in their canonical slot.

Compared to Bloom filters, Quotient filters are very cache friendly. Each lookup/insert jumps to a single random offset determined by the hash value, as opposed to *k* independent hash functions in a Bloom filter. If the load factor is managed properly, runs and clusters are quite small, so very few cache lines must be accessed in order to perform any QF operations.

Quotient filters form the foundation of a *write-optimized* data structure called a cascade filter, due to their ability to efficiently resize and merge.

- Quotient filters can be efficiently merged (using a process similar to merging sorted lists) halved in size (reinsert runs from left to right), and doubled in size (reinsert runs from right to left).

Questions

1. What types of applications can benefit from using Bloom filters?
2. What types of applications could never use a Bloom filter?
3. How good/bad is the cache behavior of a standard Bloom filter (i.e., do lookups/inserts into the bit array have any locality)?
    - Can you think of any ways to improve cache locality?
4. For the following operations, decide whether a standard Bloom filters could support the operation, and why/why not (or under what conditions it is possible):
    - combining two Bloom filters
    - resizing a Bloom filter

    - deleting an element
5. Can you think of any way to modify the standard Bloom filter to support deletes?
    - you may consider storing auxiliary data in the bit array, but keep in mind that you want to minimize the memory footprint
6. Which data structure do you think is the easiest to implement:
    - Bloom filter
    - Quotient filter
    - Cuckoo filter

What types of things seem most challenging to get right? Do those things affect performance or correctness?