Probabilistic data structures. Part 1. Membership

Presentation · July 2016

DOI: 10.13140/RG.2.1.4836.1203

1 author:

Andrii Gakhov
V. N. Karazin Kharkiv National University

PROBABILISTIC DATA STRUCTURES

ALL YOU WANTED TO KNOW BUT WERE AFRAID TO ASK

PART 1: MEMBERSHIP

Andrii Gakhov
tech talk @ ferret

MEMBERSHIP

Agenda:

- Bloom Filter
- Quotient filter

THE PROBLEM

- To determine membership of the element in a large set of elements

BLOOM FILTER

BLOOM FILTER

- proposed by Burton Howard Bloom, 1970
- Bloom filter is a realisation of *probabilistic set* with 3 operations:
  - **add** element into the set
  - test whether an element **is a member** of the set
  - test whether an element **is not a member** of the set
- Bloom filter is described by 2 parameters:
  - **m** - length of the filter
    (proportional to expected number of elements **n**)
  - **k** - number of different hash functions
    (usually, k is much smaller than m)
- It doesn't store elements and require about 1 byte per stored data

BLOOM FILTER: ALGORITHM

- Bloom filter is a *bit array* of **m** bits, all set to 0 at the beginning
- **To insert** element into the filter - calculate values of all **k** hash functions for the element and set bit with the corresponding indices
- **To test** if element is in the filter - calculate all **k** hash functions for the element and check bits in all corresponding indices:
  - if all bits are set, then answer is "**maybe**"
  - if at least 1 bit isn't set, then answer is "**definitely not**"
- **Time** needed to insert or test elements is a fixed constant **O(k)**, independent from the number of items already in the filter

BLOOM FILTER: EXAMPLE

- Consider Bloom filter of 16 bits (m=16)

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- Consider 2 hash functions (k=2):

  MurmurHash3 and **Fowler-Noll-Vo**

  (to calculate the appropriate index, we divide result by mod 16)

- Add element to the filter: "ferret":

  MurmurHash3("ferret") = 1, FNV("ferret") = 11

0	1	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

BLOOM FILTER: EXAMPLE

- Add element to the filter: “bernau”:

  MurmurHash3(“bernau”) = 4, FNV(“bernau”) = 4

0	1	0	0	1	0	0	0	0	0	0	1	0	0	0	0
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- Test element: “berlin”:

  MurmurHash3(“berlin”) = 4, FNV(“berlin”) = 12

  Bit **12** is not set, so “berlin” **definitely not in the set**

- Test element: “paris”:

  MurmurHash3(“paris”) = 11, FNV(“paris”) = 4

  Bits **4** and **11** are set, so the element **maybe in the set (false positive)**

BLOOM FILTER: PROPERTIES

- **False positives** are **possible.** $P(e \in \Im \mid e \notin \Im) \approx \left(1 - e^{\frac{kn}{m}}\right)^k$

  (element is not a member, but filter returns like it is a member)

- **False negatives** are **not possible.** $P(e \notin \Im \mid e \in \Im) = 0$

  (filter returns that elements isn't a member only if it's not a member)

- Hash functions should be **independent** and **uniformly distributed**. They should also be as **fast as possible.**

  (don't use cryptographic, like sha1)

- By choice of **k** and **m** it is possible to decrease false positives probability.

  $$k^* = \frac{m}{n} \ln 2$$

BLOOM FILTER: APPLICATIONS

- **Google BigTable**, **Apache HBase** and **Apache Cassandra** use Bloom filters to reduce the disk lookups for non-existent rows or columns
- **Medium** uses Bloom filters to avoid recommending articles a user has previously read
- **Google Chrome** web browser used to use a Bloom filter to identify malicious URLs (moved to PrefixSet, Issue 71832)
- The **Squid Web Proxy** Cache uses Bloom filters for cache digests

BLOOM FILTER: PROBLEMS

- The basic Bloom filter **doesn’t support deletion**.
- Bloom filters work well when they **fit in main memory**
  - ~1 byte per element (3x-4x times more for Counting Bloom filters, that support deletion)
- What goes wrong when Bloom filters **grow too big to fit in main memory**?
  - On disks with rotating platters and moving heads, Bloom filters choke. A rotational disk performs only 100–200 (random) I/Os per second, and each Bloom filter operation requires multiple I/Os.
  - On flash-based solid-state drives, Bloom filters achieve only hundreds of operations per second in contrast to the order of a million per second in main memory.
- Buffering can help. However, buffering **scales poorly** as the Bloom-filter size increases compared to the in-memory buffer size, resulting in only a few buffered updates per flash page on average.

BLOOM FILTER: VARIANTS

- **Attenuated Bloom filters** use arrays of Bloom filters to store shortest path distance information
- **Spectral Bloom filters** extend the data structure to support estimates of frequencies.
- **Counting Bloom Filters** each entry in the filter instead of a single bit is rather a small counter. Insertions and deletions to the filter increment or decrement the counters respectively.
- **Compressed Bloom filters** can be easily adjusted to the desired tradeoff between size and false-positive rate
- **Bloom Filter Cascade** implements filtering by cascading pipeline of Bloom filters
- **Scalable Bloom Filters** can adapt dynamically to the number of elements stored, while assuring a maximum false positive probability

BLOOM FILTER: PYTHON

- https://github.com/jaybaird/python-bloomfilter
  pybloom is a module that includes a Bloom Filter data structure along with an implementation of Scalable Bloom Filters
- https://github.com/seomoz/pyreBloom
  pyreBloom provides Redis backed Bloom Filter using GETBIT and SETBIT

BLOOM FILTER: READ MORE

- http://dmod.eu/deca/ft_gateway.cfm.pdf
- https://en.wikipedia.org/wiki/Bloom_filter
- https://www.cs.uchicago.edu/~matei/PAPERS/bf.doc
- http://gsd.di.uminho.pt/members/cbm/ps/dbloom.pdf
- https://www.eecs.harvard.edu/~michaelm/postscripts/im2005b.pdf

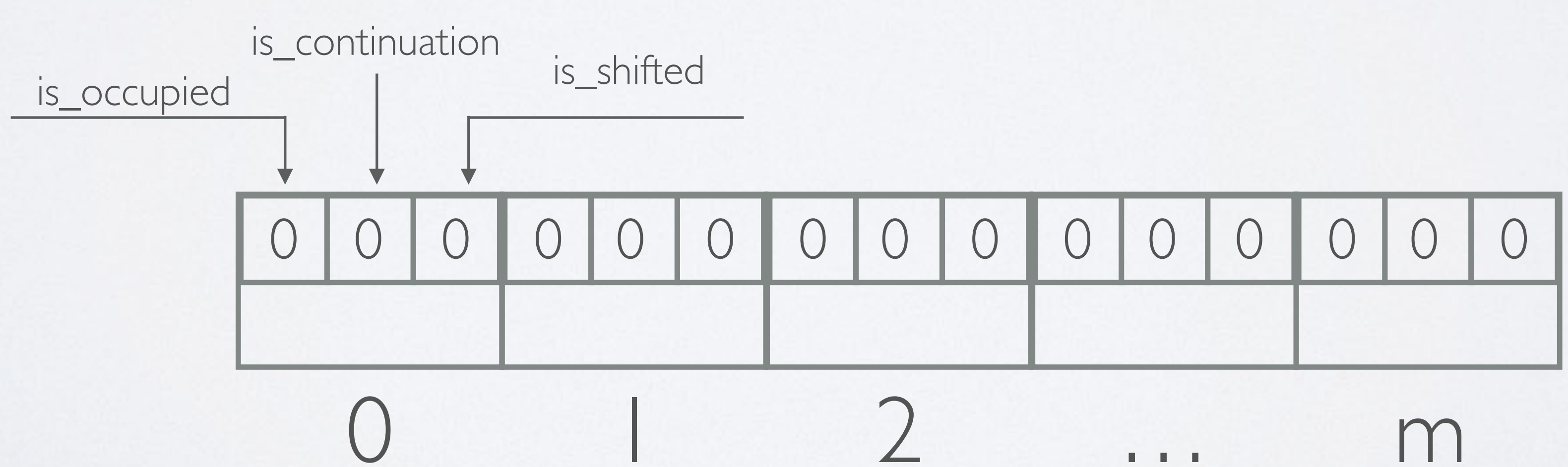
QUOTIENT FILTER

QUOTIENT FILTER

- introduced by Michael Bender et al., 2011
- Quotient filter is a realisation of *probabilistic set* with 4 operations:
  - **add** element into the set
  - **delete** element from the set
  - test whether an element **is a member** of a set
  - test whether an element **is not a member** of a set
- Quotient filter is described by:
  - **p** - size (in bits) for fingerprints
  - **single** hash function that generates fingerprints
- Quotient filter stores **p-bit fingerprints** of elements

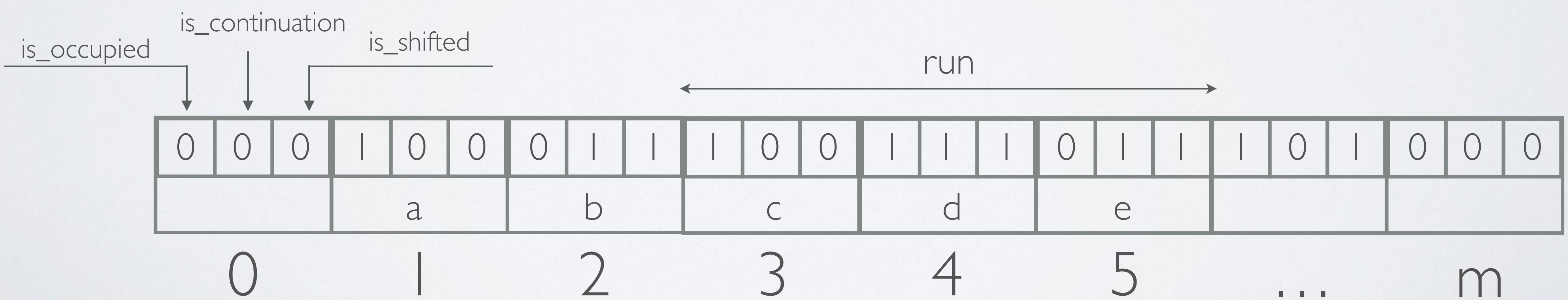
QUOTIENT FILTER: ALGORITHM

- Quotient filter is a compact **open hash table** with $\mathbf{m = 2^q}$ buckets**.** The hash table employs ***quotienting***, a technique suggested by D. Knuth:
  - the fingerprint **f** is partitioned into:
    - the **r least significant bits** ($f_r = f \bmod 2^r$, the remainder)
    - the **q = p - r most significant bits** ($f_q = \lfloor f / 2^r \rfloor$, the quotient)
- The remainder is stored in the bucket indexed by the quotient
- Each bucket contains 3 bits, all 0 at the beginning: **is_occupied**, **is_continuation**, **is_shifted**



QUOTIENT FILTER: ALGORITHM

- If two fingerprints **f** and **f′** have the same quotient ($\mathbf{f_q}=\mathbf{f'_q}$) - it is a **soft collision**. All remainders of fingerprints with the same quotient are stored contiguously in a **run**.
- If necessary, a remainder is **shifted** forward from its original location and stored in a subsequent bucket, wrapping around at the end of the array.
  - **is_occupied** is set when a bucket **j** is the canonical bucket ($\mathbf{f_q}=\mathbf{j}$) for some fingerprint **f** , stored (somewhere) in the filter
  - **is_continuation** is set when a bucket is occupied but not by the first remainder in a *run*
  - **is_shifted** is set when the remainder in a bucket is not in its canonical bucket

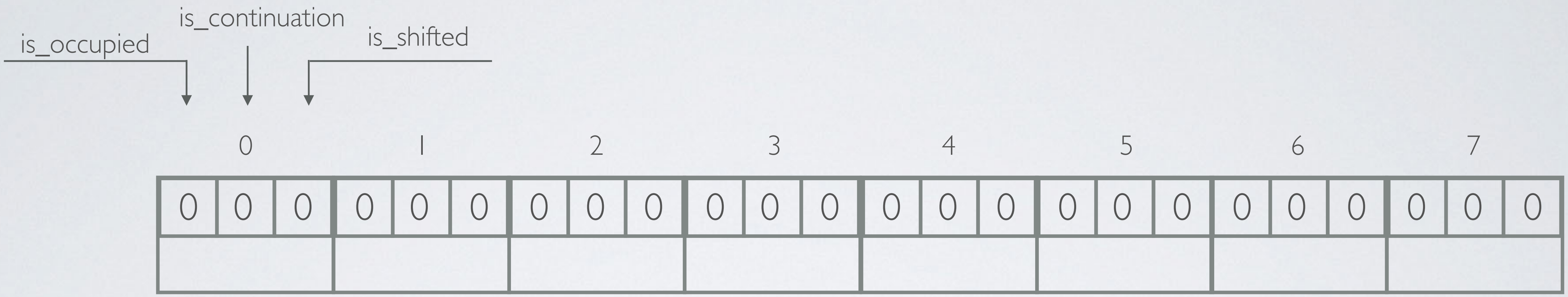


QUOTIENT FILTER: ALGORITHM

- To **test** a fingerprint **f**:
  - if bucket $\mathbf{f_q}$ is not occupied, then the element with fingerprint **f** definitely not in the filter
  - if bucket $\mathbf{f_q}$ is occupied:
    - starting with bucket $\mathbf{f_q}$, scan **left** to locate bucket without set **is_shifted** bit
    - scan **right** with running count (is_occupied: +1, is_continuation: -1) until the running count reaches 0 - when it's the quotient's *run*.
    - compare the remainder in each bucket in the quotient's run with $\mathbf{f_r}$
    - if found, than element ***is (probably)*** in the filter, else - it is ***definitely not*** in the filter.
- To **add** a fingerprint **f**:
  - follow a path similar to test until certain that the fingerprint is definitely not in the filter
  - choose bucket in the current *run* by keeping the sorted order and insert reminder $\mathbf{f_r}$ (set is_occupied bit)
  - shift forward all reminders at or after the chosen bucket and update the buckets' bits.

QUOTIENT FILTER: EXAMPLE

- Consider Quotient filter with size **p=8** and 32-bit signed **MurmurHash3** as **h**

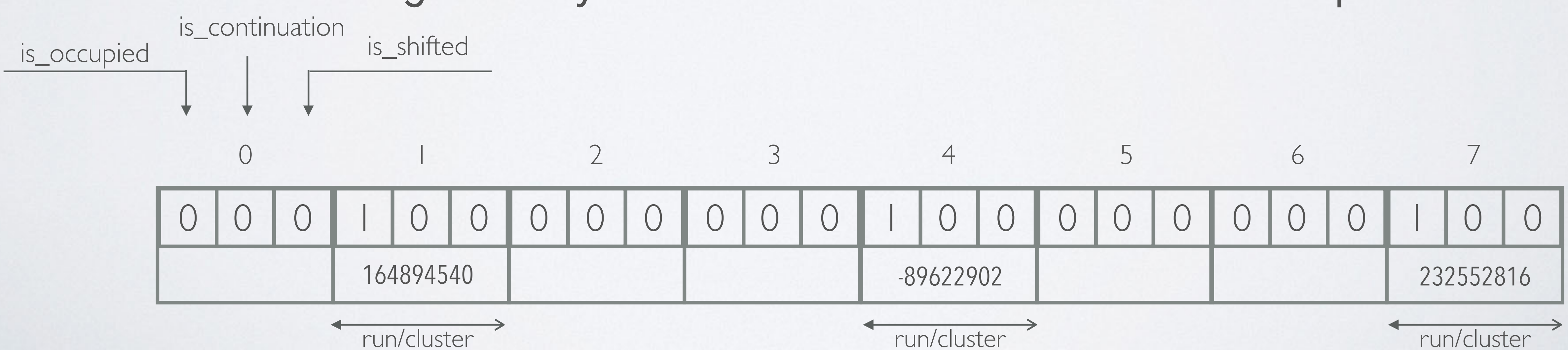


- Add elements: "amsterdam", "berlin", "london"

f_q("amsterdam") = 1, f_q("berlin") = 4, f_q("london") = 7

f_r("amsterdam") =164894540, f_r("berlin") = -89622902, f_r("london") = 232552816

Insertion at this stage is easy since all canonical slots are not occupied



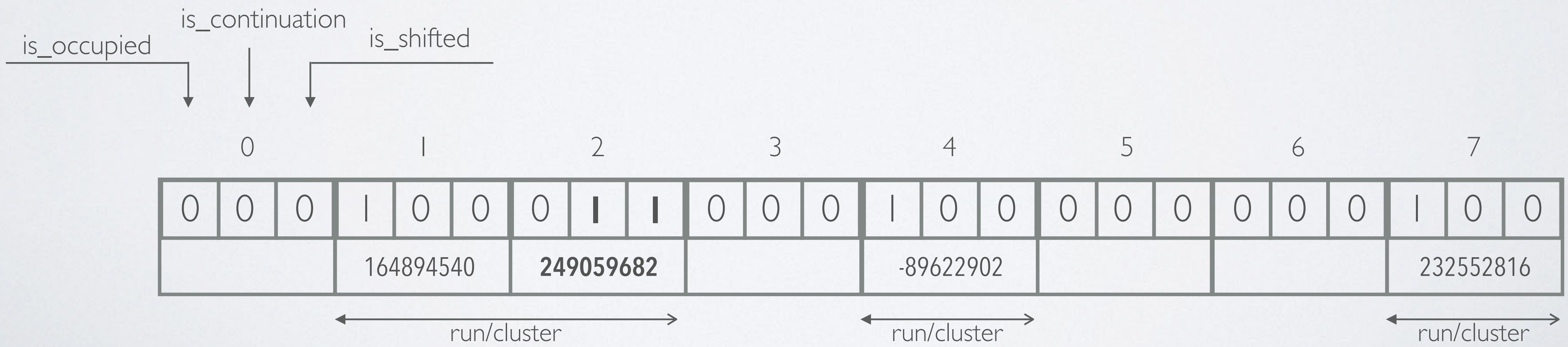
QUOTIENT FILTER: EXAMPLE

- Add element: "madrid"

  f_q("madrid") = 1, f_r("madrid") = 249059682

The canonical slot 1 is already occupied. The *shifted* and *continuation bits* are not set, so we are at the beginning of the cluster which is also the run's start.

The reminder f_r("madrid") is *strongly bigger* than the existing reminder, so it should be shifted right into the next available slot 2 and *shifted/continuation bits* should be set (but not the *occupied bit*, because it pertain to the slot, not the contained reminder).

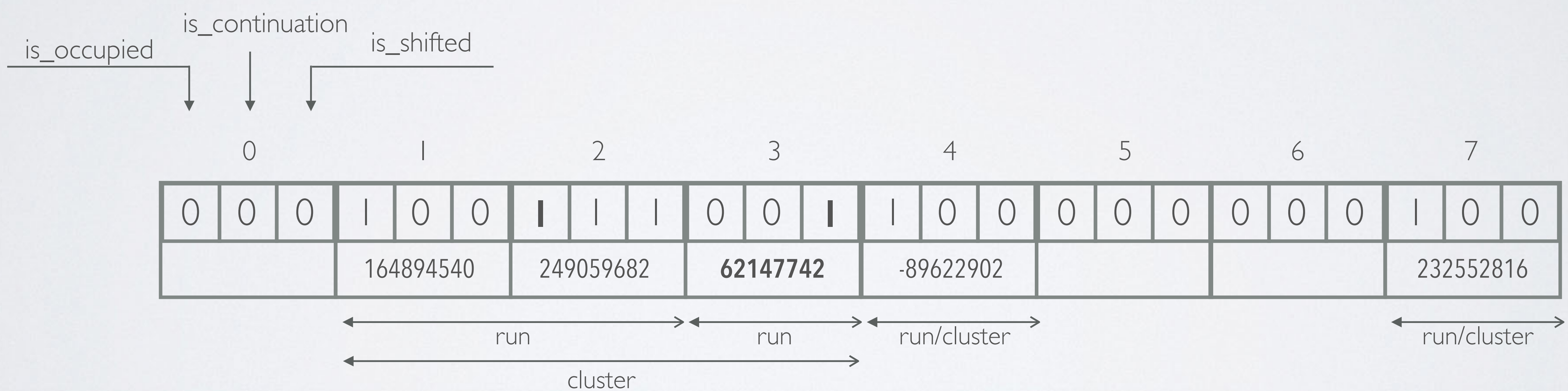


QUOTIENT FILTER: EXAMPLE

- Add element: "ankara"

  f_q("ankara") = 2, f_r("ankara") = 62147742

The canonical slot 2 is not occupied, but already in use. So, the fr("ankara") should be shifted right into the neared available slot 3 and its *shifted bit* should be set. In addition, we need to flag the canonical slot 2 as occupied by setting the *occupied bit*.



QUOTIENT FILTER: EXAMPLE

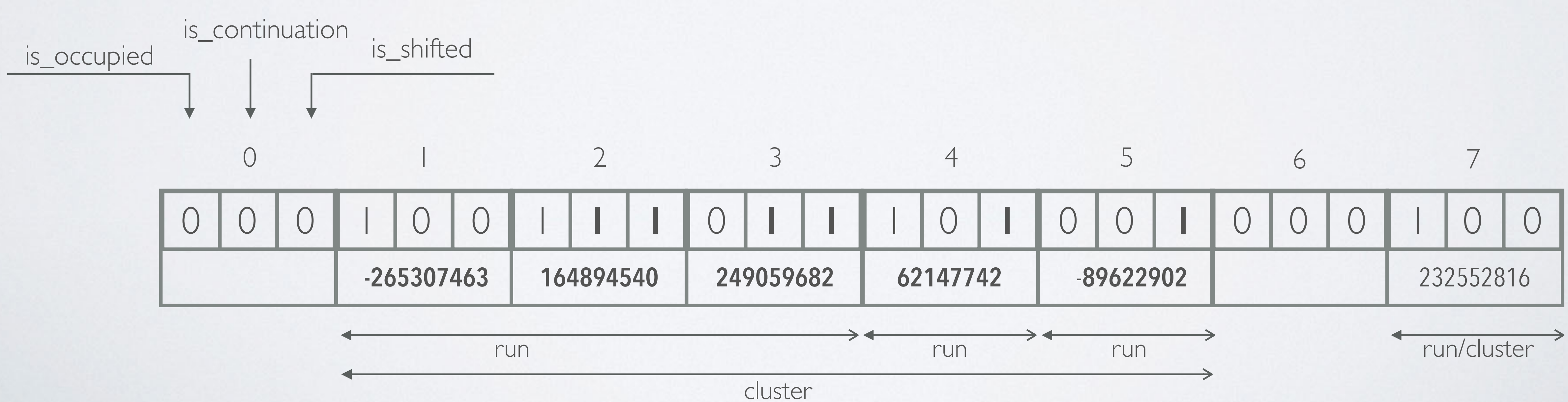
- Add element: "abu dhabi"

  f_q("abu dhabi") = 1, f_r("abu dhabi") = -265307463

The canonical slot 1 is already occupied. The *shifted* and *continuation bits* are not set, so we are at the beginning of the cluster which is also the run's start.

The reminder f_r("abu dhabi") is *strongly smaller* than the existing reminder, so all reminders in slot 1 should be shifted right and flagged as continuation and shifted.

If shifting affects reminders from other runs/clusters, we also shift them right and set *shifted bits (and mirror the continuation bits if they are set there)*.



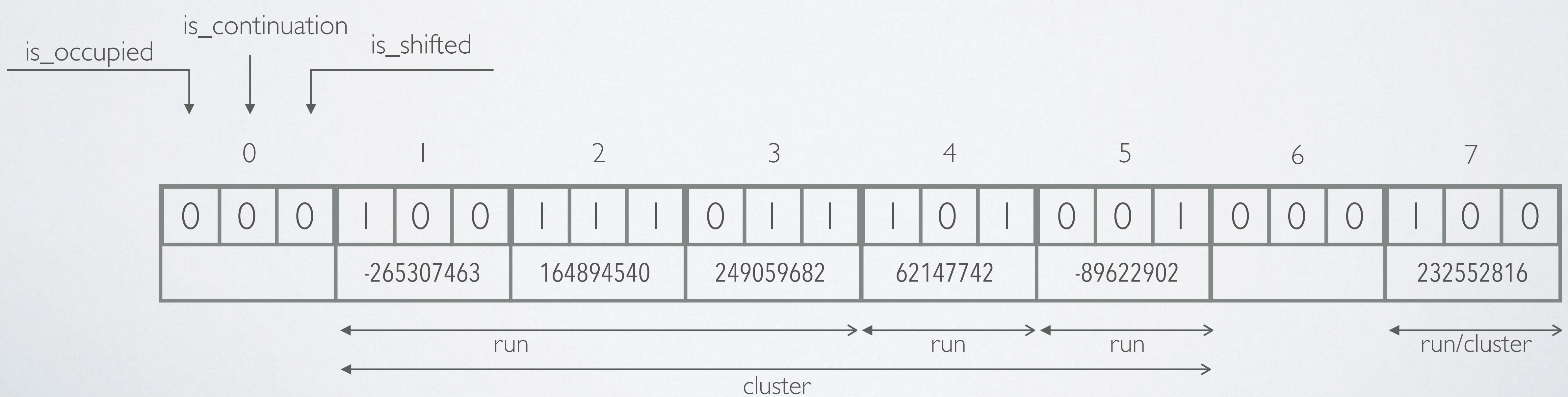
QUOTIENT FILTER: EXAMPLE

- Test element: "ferret"

  f_q("ferret") = 1, f_r("ferret") = 122150710

The canonical slot 1 is already occupied and it's the start of the run. Iterate through the run and compare f_r("ferret") with existing reminders until we found the match, found reminder that is strongly bigger, or hit the run's end.

We start from slot 1 which reminder is smaller, so we continue to slot 2. Reminder in the slot 2 is already bigger than f_r("ferret"), so we conclude that "ferret" **definitely not in the filter**.



QUOTIENT FILTER: EXAMPLE

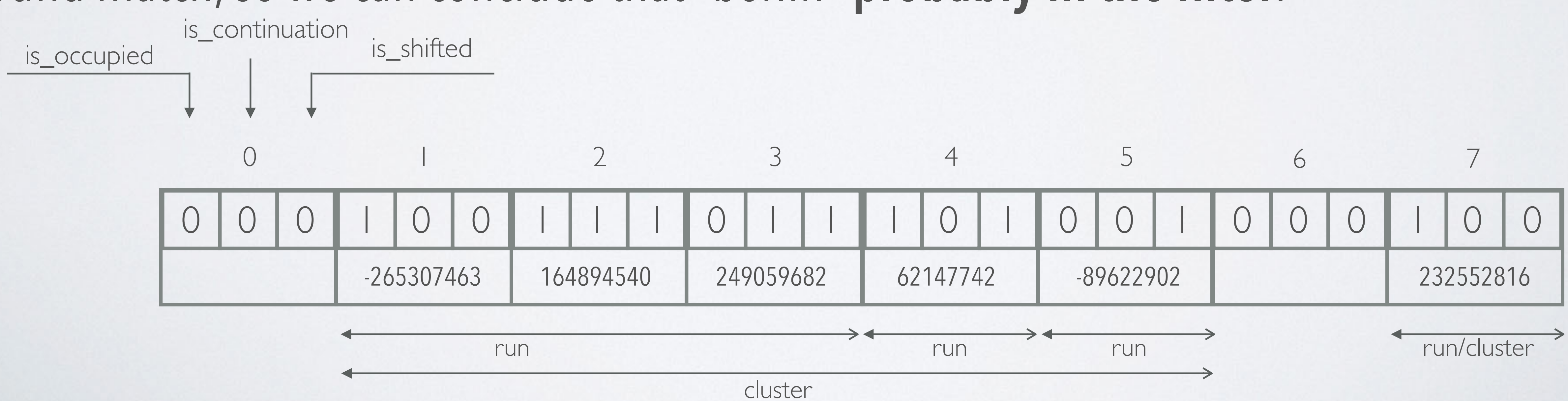
- Test element: "berlin"

  f_q("berlin") = 4, f_r("berlin") = -89622902

The canonical slot 4 is already occupied, but shifted bit is set, so the run for which it is canonical slot exists, but is shifted right.

First, we need to find a run corresponding to the canonical slot 4 in the current cluster., so we scan left and count occupied slots. There are 2 occupied slots found (indices 1 and 2), therefore our run is the 3rd in the cluster and we can scan right until we found it (count slots with not set *continuation bit*).

Our run starts in the slot 5 and we start comparing f_r("berlin") with existing values and found match, so we can conclude that "berlin" **probably in the filter**.



QUOTIENT FILTER: PROPERTIES

- **False positives** are **possible.** $P\left(e \in \Im \mid e \notin \Im\right) \leq \frac{1}{2^r}$

  (element is not a member, but filter returns like it is a member)

- **False negatives** are **not possible.** $P\left(e \notin \Im \mid e \in \Im\right) = 0$

  (filter returns that elements isn't a member only if it's not a member)

- Hash function should generate **uniformly distributed** fingerprints.

- The length of most runs is O(1) and it is highly likely that all runs have length O(log m)

- Quotient filter efficient for large number of elements (~1B for 64-bit hash function)

QUOTIENT FILTER VS BLOOM FILTER

- Quotient filters are about 20% **bigger** than Bloom filters, but **faster** because each access requires evaluating only a **single hash function**
- Results of comparison of in-RAM performance (M. Bender et al.):
    - **inserts**: BF: 690 000 inserts per second, QF: 2 400 000 insert per second
    - **lookups**: BF: 1 900 000 lookups per second, QF: 2 000 000 lookups per second
- Lookups in Quotient filters incur a single cache miss, as opposed to at least two in expectation for a Bloom filter.
- Two Quotient Filters can be efficiently **merged without affecting their false positive** rates. This is not possible with Bloom filters.
- Quotient filters support deletion

QUOTIENT FILTER: IMPLEMENTATIONS

- https://github.com/vedantk/quotient-filter
  Quotient filter in-memory implementation written in C
- https://github.com/dsx724/php-quotient-filter
  Quotient filter implementation in pure PHP
- https://github.com/bucaojit/QuotientFilter
  Quotient filter implementation in Java

QUOTIENT FILTER: READ MORE

- http://static.usenix.org/events/hotstorage11/tech/final_files/Bender.pdf
- http://arxiv.org/pdf/1208.0290.pdf
- https://en.wikipedia.org/wiki/Quotient_filter
- http://www.vldb.org/pvldb/vol6/p589-dutta.pdf

THANK YOU

- @gakhov
- linkedin.com/in/gakhov
- www.datacrucis.com