Filters (Bloom & Quotient)

CSCI 333

Operations

- Filters *approximately* represent sets. Therefore, a filter *must* support:
  - Insertions: `insert(key)`
  - Queries: `lookup(key)`
- Filters may also support other operations:
  - Deletion: `remove(key)`
  - Union: merge($filter_a$, $filter_b$)

Why Filters?

- By embracing approximation, filters can be memory efficient data structures
  - Some false positives are allowed
  - But false negatives are never allowed

- Many applications are OK with this behavior
  - Typically used in applications where a wrong answer just wastes work, does not harm correctness
    - Save expensive work (I/O) *most of the time*

Bloom Filters

Goal: approximately represent a set of **n** elements using a bit array

- Returns either:
  - Definitely NOT in the set
  - Possibly in the set

Parameters: **m**, **k**

- **m**: Number of bits in the array
- **k**: Set of **k** hash functions { h_1, h_2, …, h_k }, each with range {**0**…**m−1**}

Concrete Example: `k=3`, `m=10`

M =

0	0	0	0	0	0	0	0	0	0

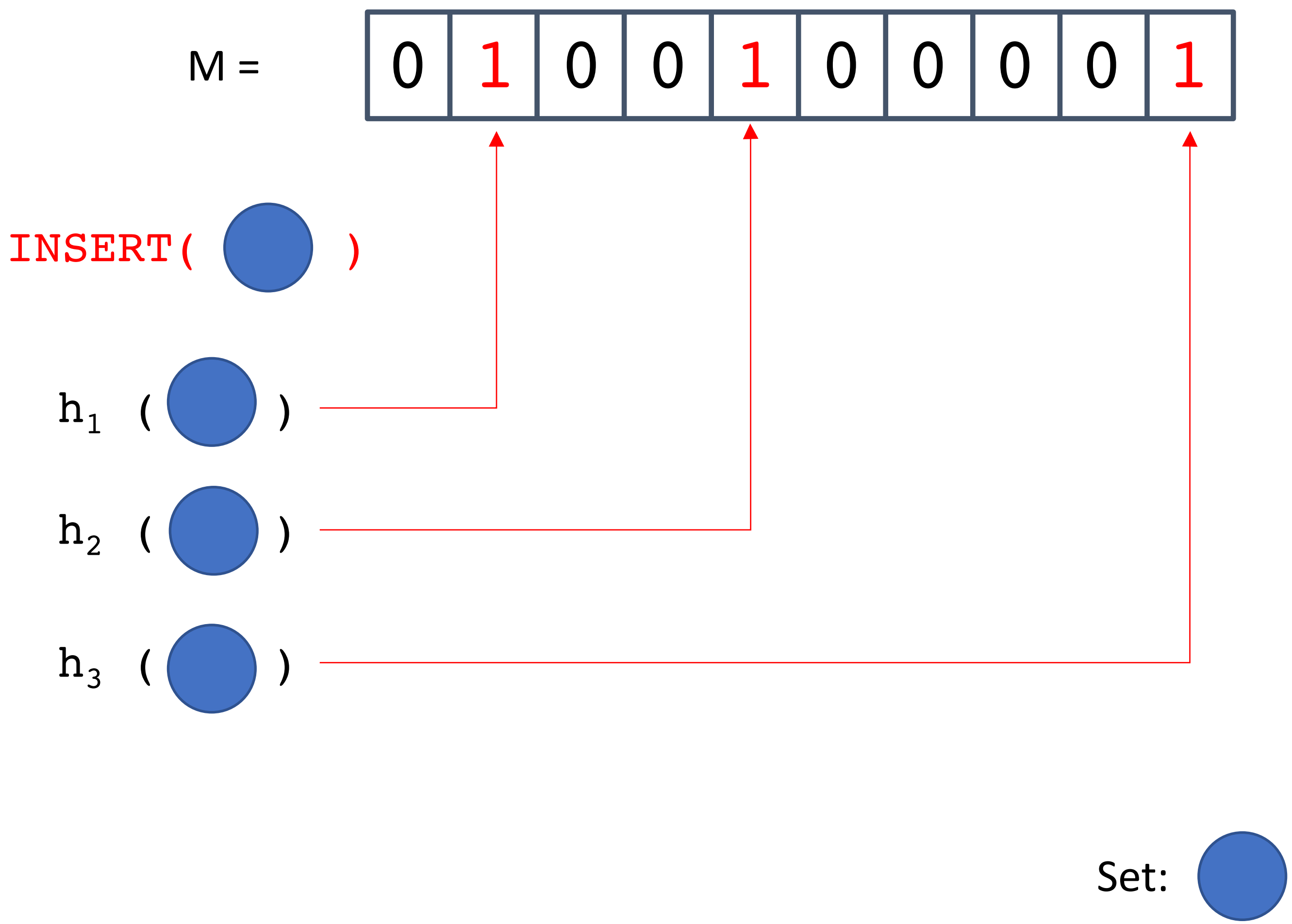
`INSERT(  )`

h_1 ( )

h_2 ( )

h_3 ( )

Concrete Example: $k=3$, $m=10$



Concrete Example: `k=3`, `m=10`

M =

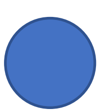
0	1	0	0	1	0	0	0	0	1

`INSERT(`  `)`

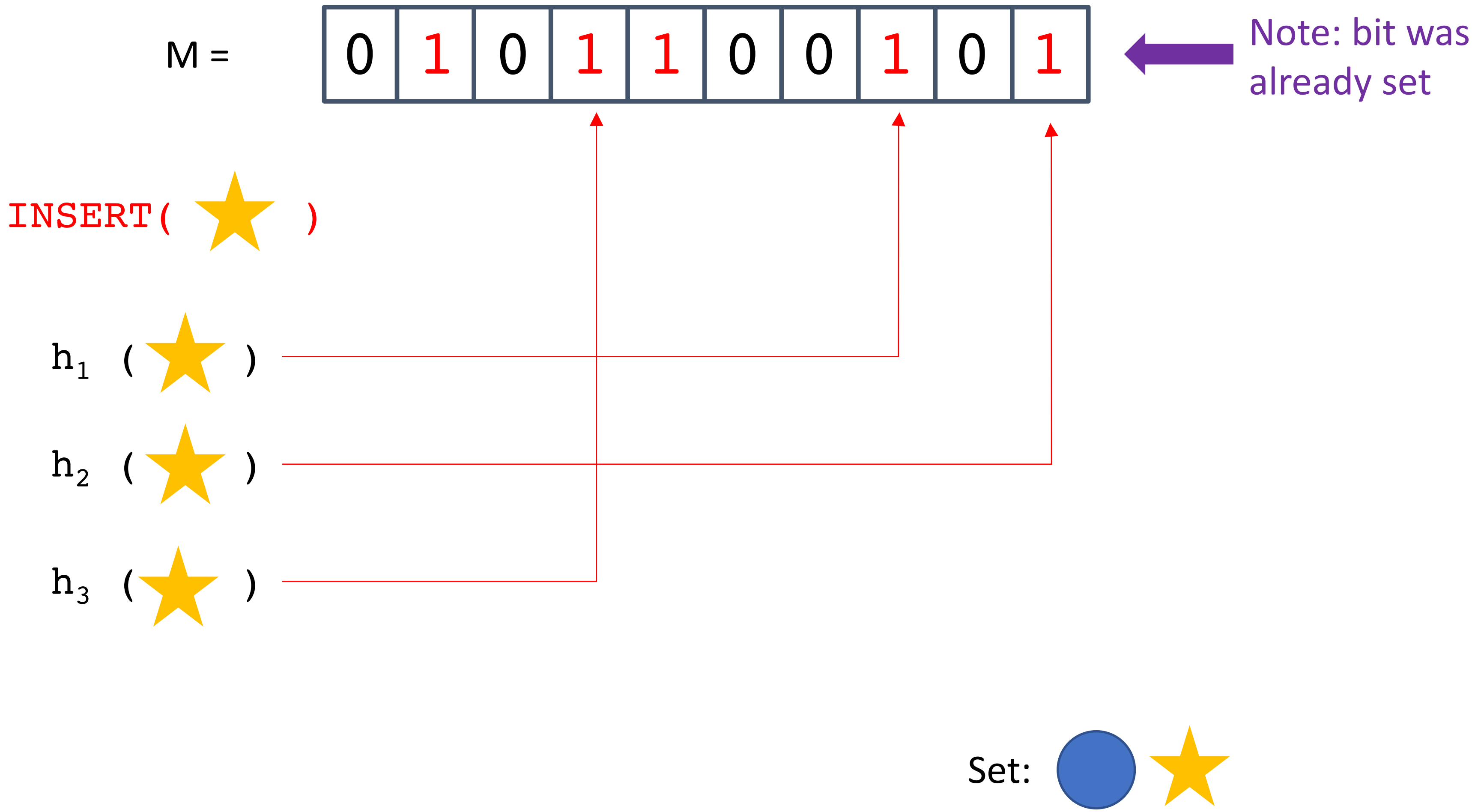
h_1 ()

h_2 ()

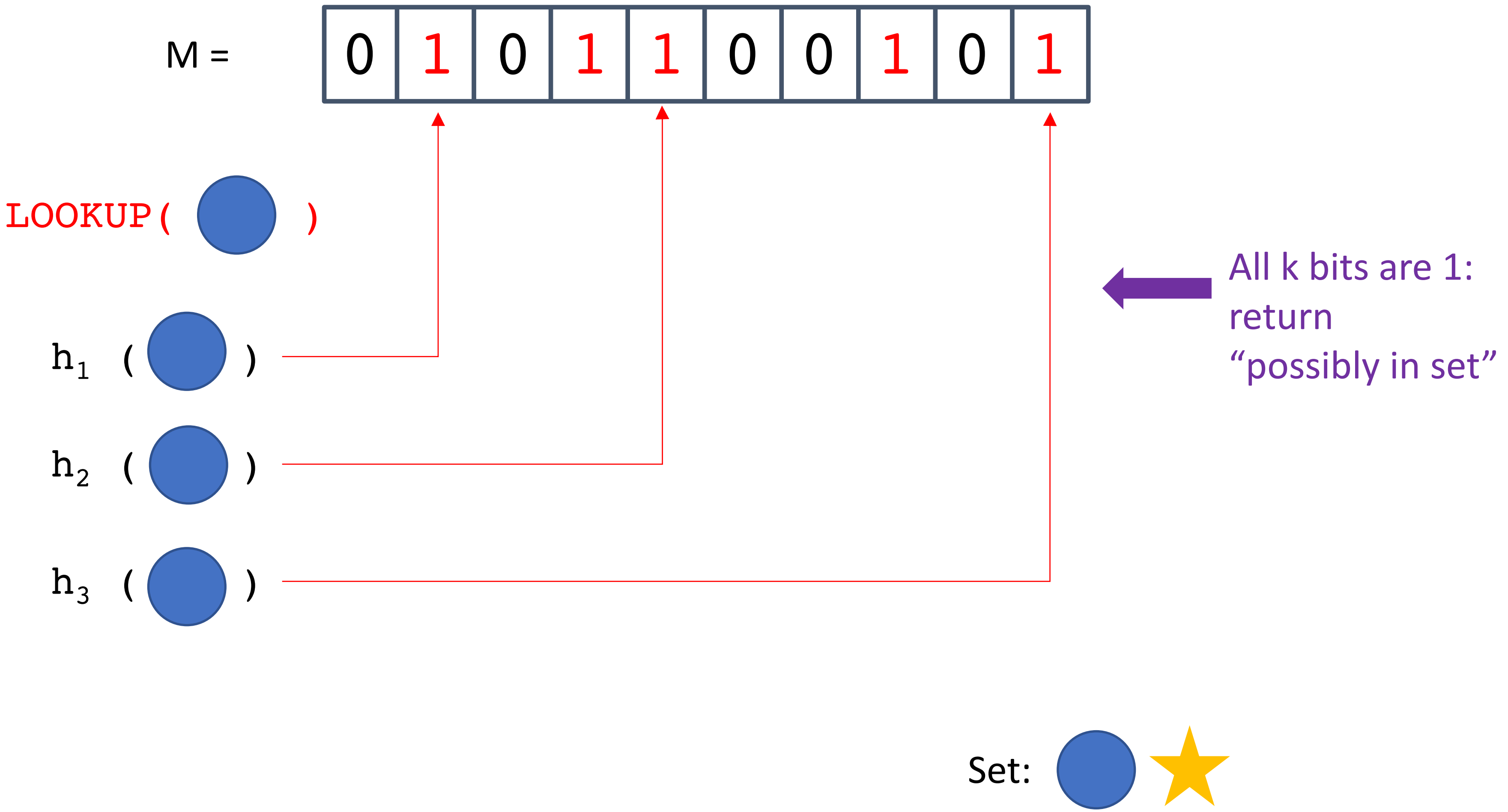
h_3 ()

Set: 

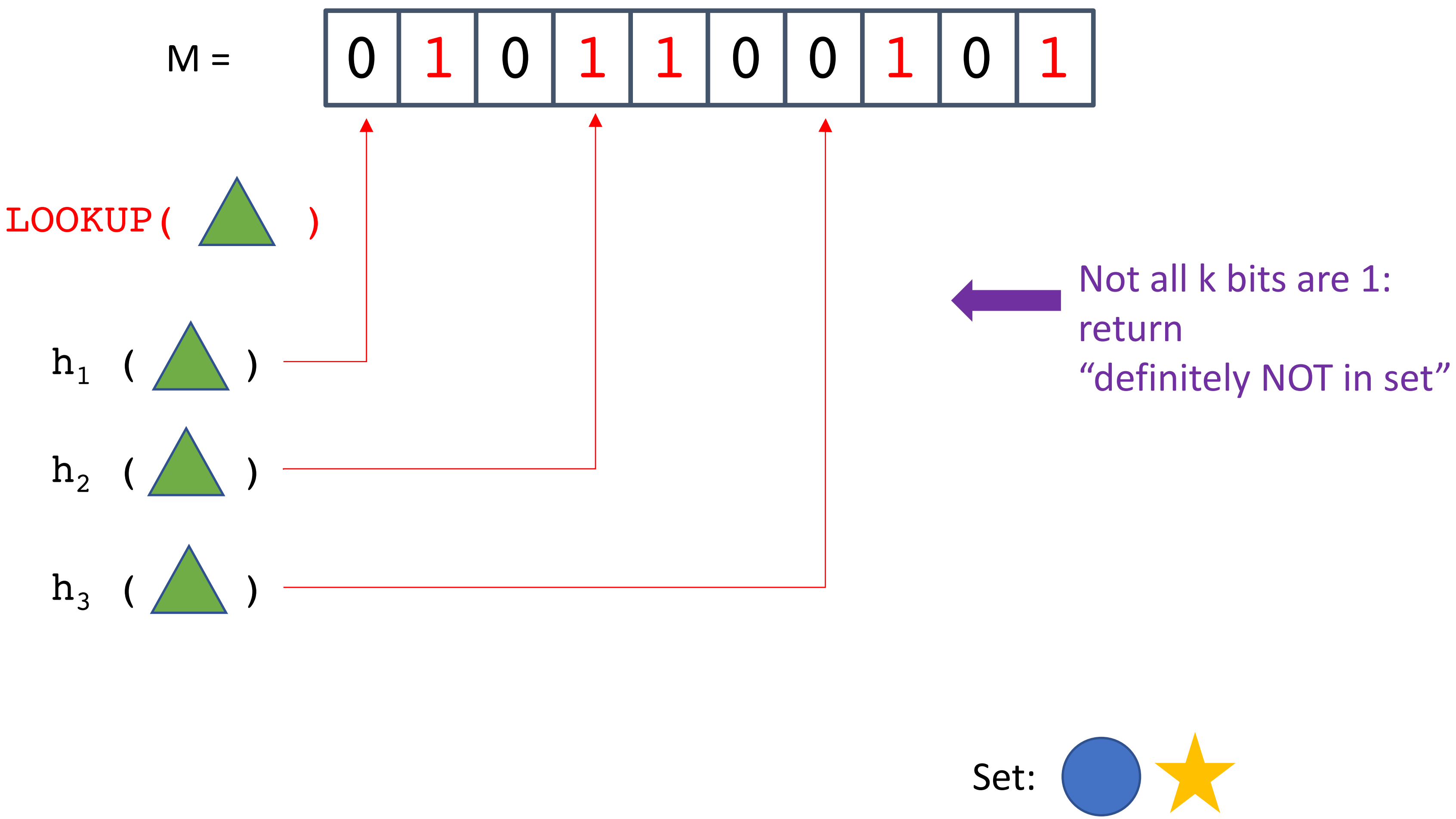
Concrete Example: `k=3`, `m=10`



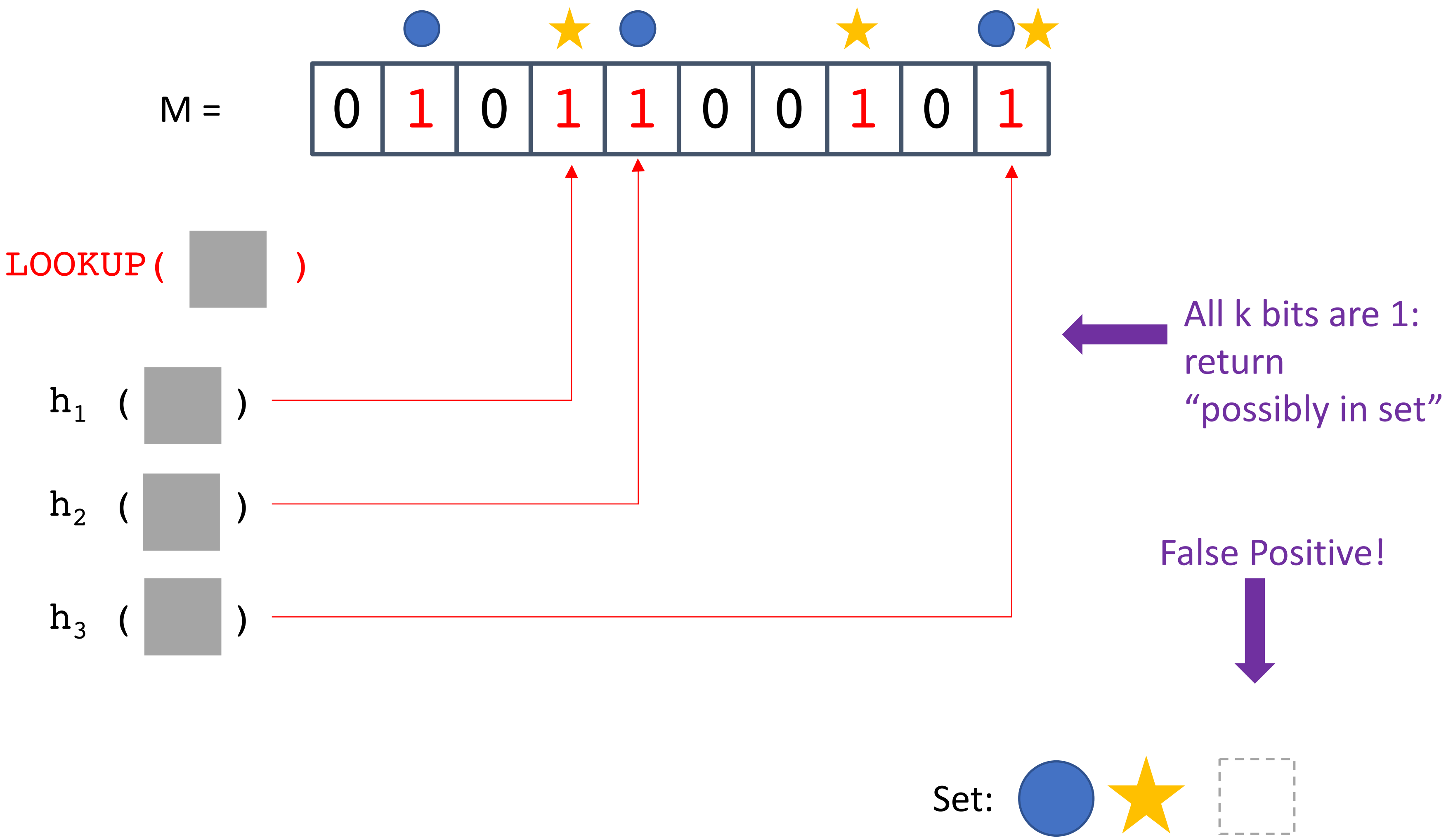
Concrete Example: `k=3`, `m=10`



Concrete Example: k=3, m=10



Concrete Example: `k=3`, `m=10`



Tuning False Positives

- What happens if we increase `m`?
- What happens if we increase `k`?

- False positive rate f is:

$$f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - \mathrm{e}^{-\frac{kn}{m}}\right)^k$$

P(a given bit is still 0) after n insertions with k hash functions

Bloom Filters

- Are there any problems with Bloom filters?
  - What operations do they support/not support?
  - How do you grow a Bloom filter?
  - What if your filter itself exceeds RAM (how bad is locality)?
    - What does the cache behavior look like?

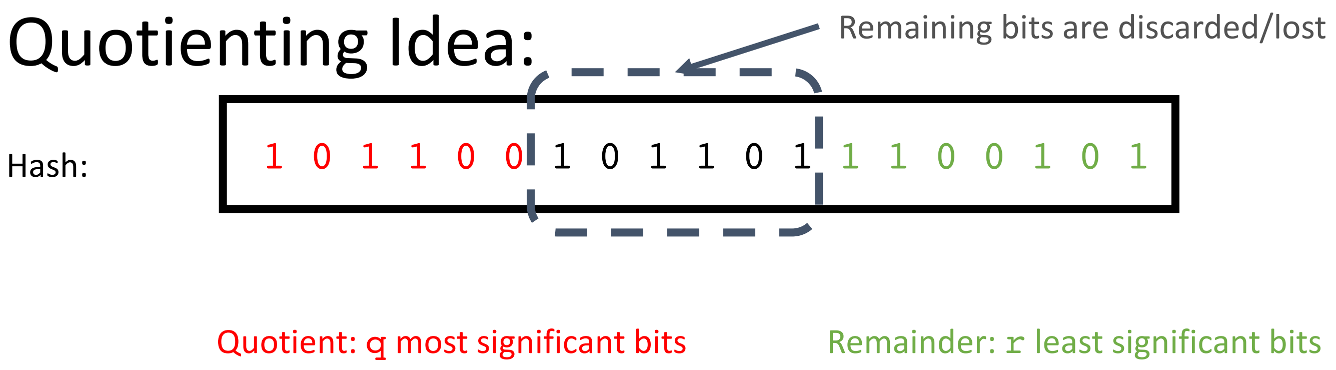
Quotient Filters

- Based on a technique from a homework question in Donald Knuth's "The Art of Computer Programming: Sorting and Searching, volume 3" (Section 6.4, exercise 13)
- Quotienting Idea:

Hash: 1 0 1 1 0 0 1 0 1 1 0 1 1 1 0 0 1 0 1

Quotient Filters

- Based on a technique from a homework question in Donald Knuth's "The Art of Computer Programming: Sorting and Searching, volume 3" (Section 6.4, exercise 13)
- Quotienting Idea:



Building a Quotient Filter

- The quotient is used as an index into an **`m`**-bucket array, where the remainder is stored.
  - Essentially, a hashtable that stores a remainder as the value
  - The quotient is *implicitly* stored because it is the bucket index

- Collisions are resolved using linear probing and 3 extra bits per bucket
  - **`is_occupied:`** whether a slot is the canonical slot for *some* value currently stored in the filter
  - **`is_continuation:`** whether a slot holds a remainder that is part of a run (but not the first element in the run)
  - **`is_shifted:`** whether a slot holds a remainder that is not in its canonical slot

- A canonical slot is an element's "home bucket", i.e., where it belongs in the absence of collisions.

Quotient Filter Example

Hash table with external chaining

Table of objects with quotients/ remainders for reference

Hash table with linear probing + bits

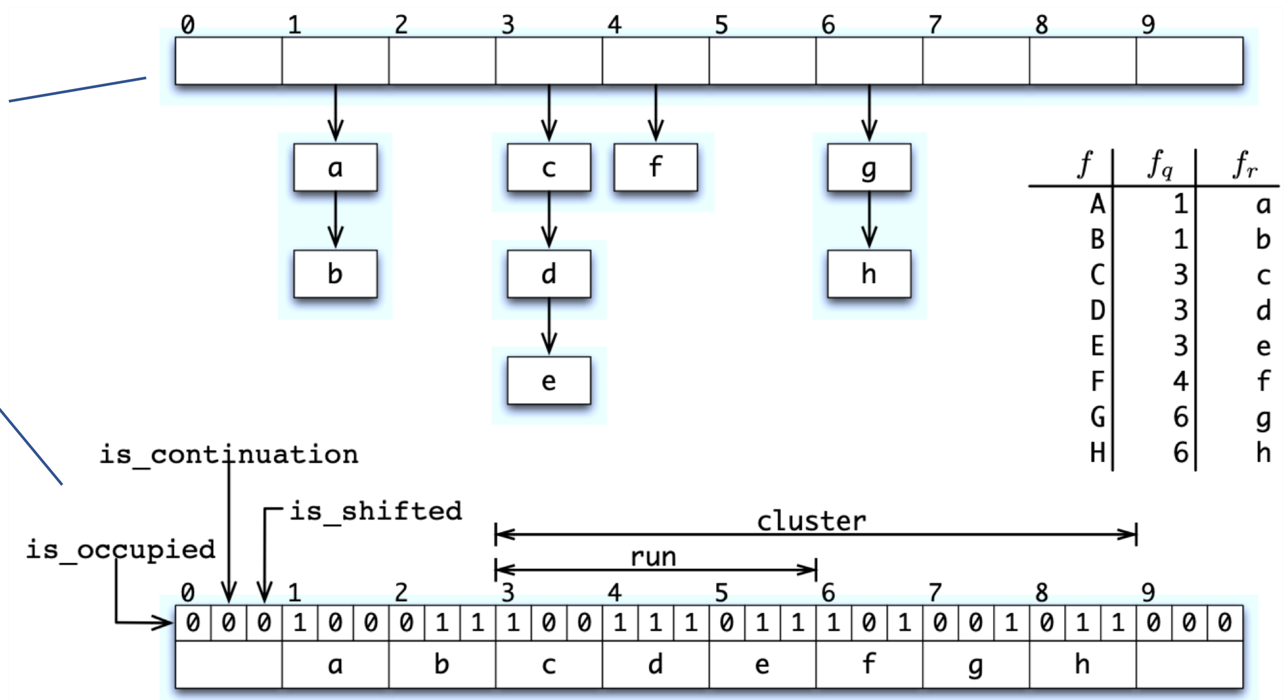


Figure 1: An example quotient filter and its representation.

[https://www.usenix.org/conference/hotstorage11/dont-thrash-how-cache-your-hash-flash]

Quotient Filter Example

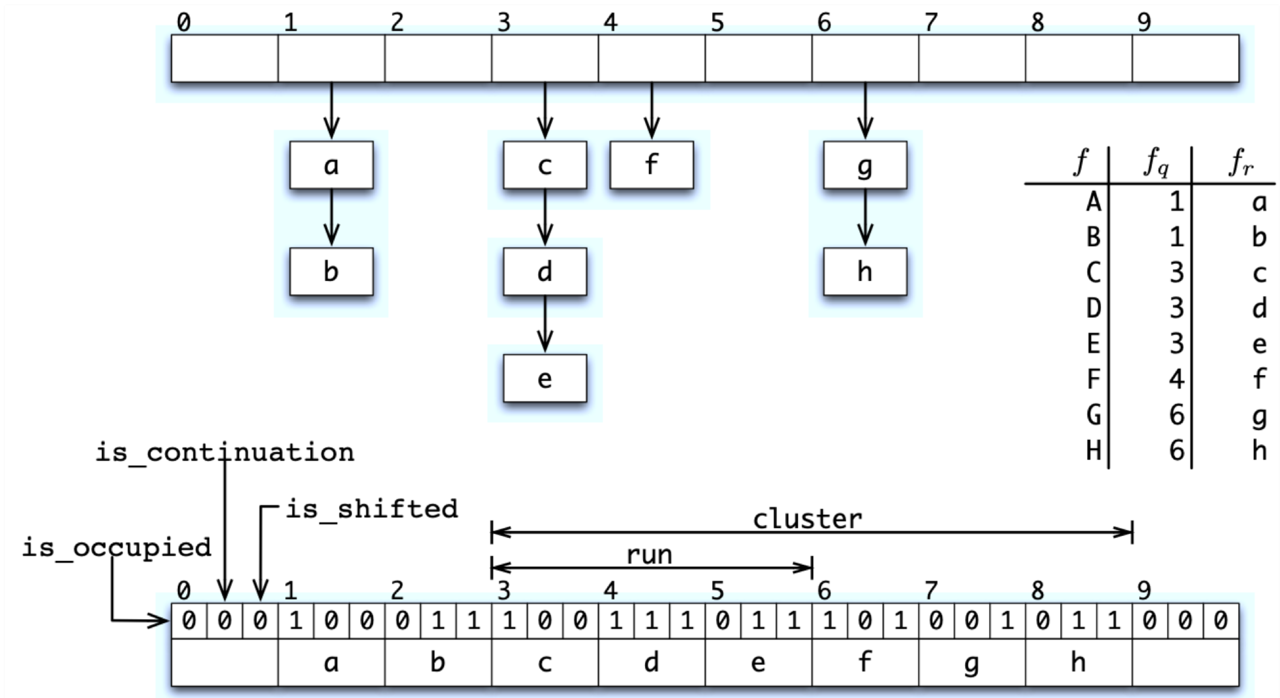
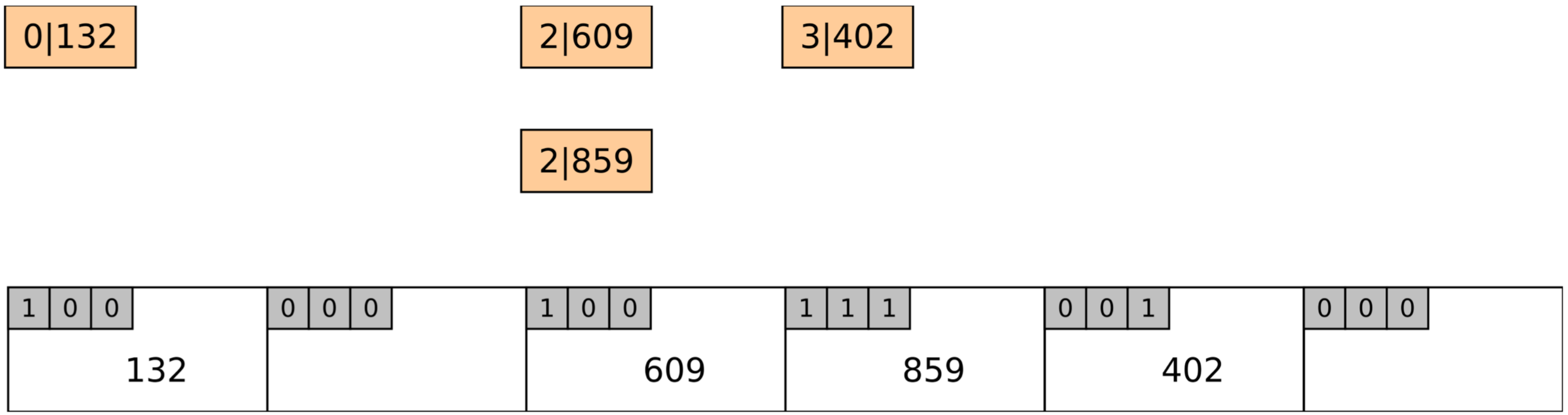


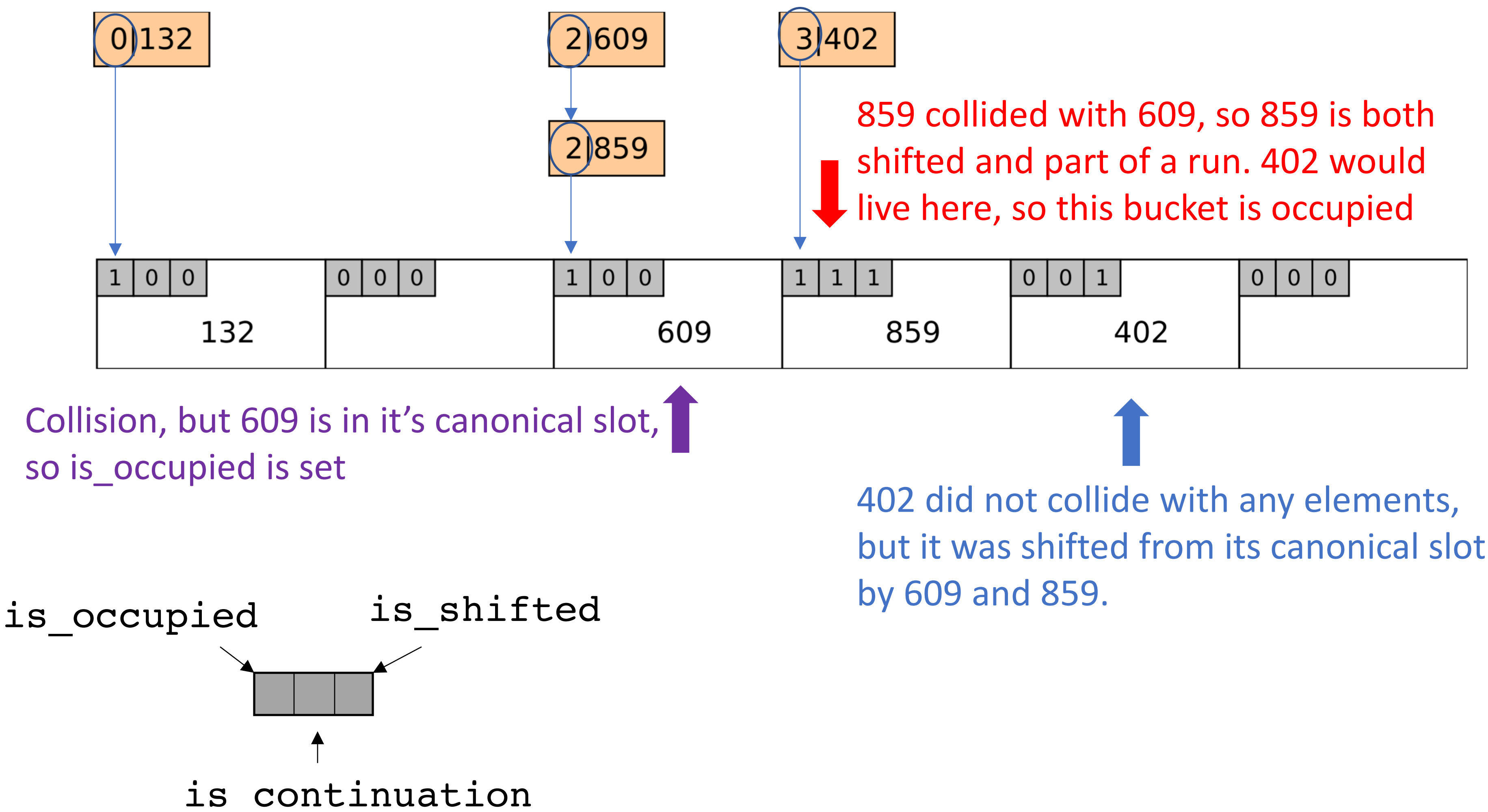
Figure 1: An example quotient filter and its representation.

[https://www.usenix.org/conference/hotstorage11/dont-thrash-how-cache-your-hash-flash]

Quotient Filter Example



Quotient Filter Example



Quotient Filter Concept-check

- What are the possible reasons for a collision?
  - Which collisions are treated as “false positives”
- What parameters does the QF give the user? In other words:
  - What knobs can you turn to control the size of the filter?
  - What knobs can you turn to control the false positive rate of the filter?

Quotient Filter Concept-check

- What are the possible reasons for a collision?
  - Collisions in the hashtable
    - Same quotient, but different remainders cause shifting
  - Collisions in the hashspace
    - Different keys may produce identical quotients/remainders
      - If a hash function collision -> not the QF's fault
      - If due to dropped bits during "quotienting" -> that is the QF's fault
  - Which collisions are treated as "false positives"
    - Collisions in the hash space
- What parameters does the QF give the user? In other words:
  - What knobs can you turn to control the size of the filter?
  - What knobs can you turn to control the false positive rate of the filter?
  - Quotient bits (number of buckets)
  - Remainder bits (how many unique bits per element to store)

Why QF over BF?

- Supports deletes
- Supports “merges”
- Good cache locality
  - How many locations accessed per operation?
  - Some math can show that runs/clusters are expected to be small

- **Don’t Thrash, How to Cache Your Hash on Flash** also introduces the Cascade filter, a write-optimized filter made up of increasingly large QFs that spill over to disk.
  - Similar idea to Log-structured merge trees, which we will discuss soon!

Cascade Filter

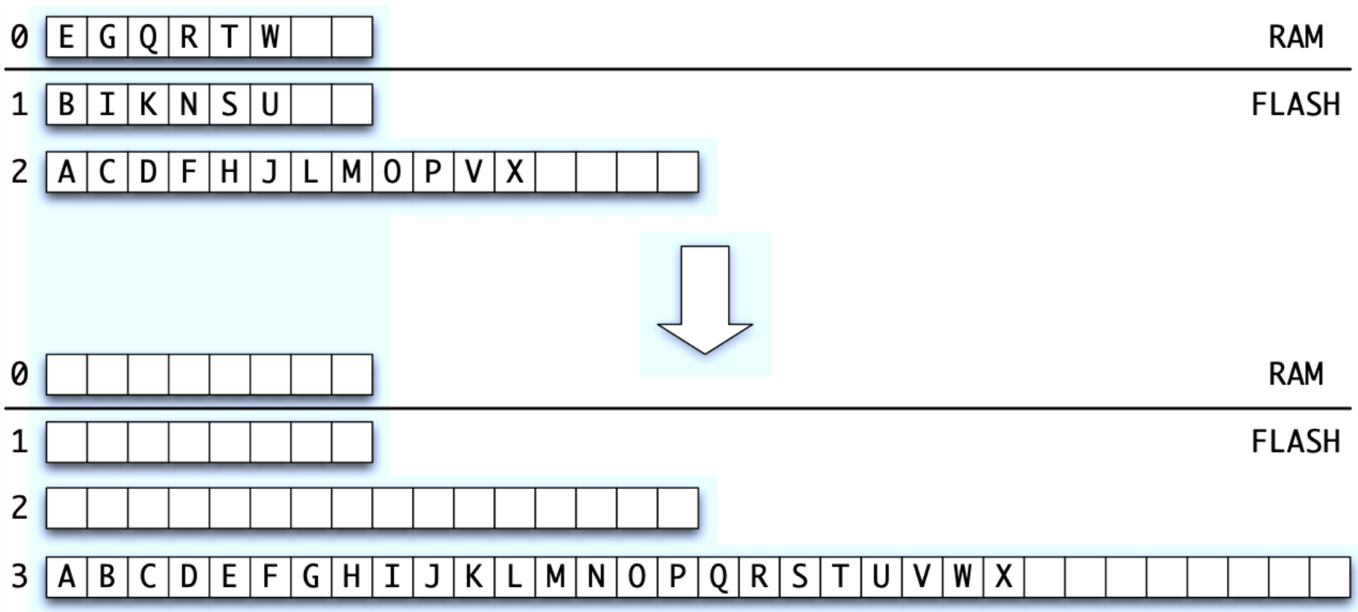


Figure 2: Merging QFs. Three QFs of different sizes are shown above, and they are merged into a single large QF below.

[https://www.usenix.org/conference/hotstorage11/dont-thrash-how-cache-your-hash-flash]