

Filters (Bloom & Quotient)

CSCI 256

Announcements/Logistics

- Last day of classes!!!!
- Review session: Monday @ 7pm, room TBA
- TA office hours: Tuesday @ around 3:30pm
(Johnathan after Ward prize presentations)
- Office hours: Wednesday 10:30-noon, 1:30-2:30pm
- Final Exam: Thursday @ 9:30
 - Any questions about the exam?

Today's Outline: Filters

- Filter Operations
- Filter Motivation
- Filter Designs
 - Bloom Filter
 - Parameters
 - Example
 - Limitations
 - Quotient Filter
 - Quotienting idea
 - Data structure
 - Example

Filter Operations

- Filters *approximately* answer questions about set membership. Therefore, a filter *must* support:
 - Insertions: `insert(key)`
 - Queries: `lookup(key)`
- Filters *may* also support other operations:
 - Deletion: `remove(key)`
 - Union: `merge(filtera, filterb)`

Filter Motivation

- Sometimes, we're asked to perform an operation on a piece of data that doesn't actually exist
 - Consider a library. If we know the call number of a book we want, we can: (1) walk to the correct floor, (2) find the shelf with the correct range, and (3) scan for the book on the shelf.
 - If the book was checked out, all of that work was wasted!
 - These steps are surprisingly similar to the process of looking for an item in a data structure so large that it exceeds RAM
- What does this have to do with filters?
 - Filters compactly represent a set so we can check for the existence of an item. **If the filter can confirm that the item does not exist, then we don't need to do the expensive search!**

Filters: the BIG idea

- Filters are not exact.
- 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 filters are used in applications where a wrong answer just wastes work, but does not harm correctness
 - Recall the library example from before:
 - If we confirm the book doesn't exist, we don't search (**correct**)
 - If we mistakenly say the book exists, all we do is waste the time that we would have needed in the absence of the filter (**correct, but slow**)

Filter Case Study: Bloom

Bloom Filters

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

- Queries return one of two answers:
 - Item is definitely NOT in the set
 - Item is Possibly in the set

Parameters: m , k

- m : Number of bits in the array (i.e., buckets)
- k : Set of k hash functions $\{h_1, h_2, \dots, h_k\}$, each with range $\{0 \dots m-1\}$

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

$M =$

0	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---

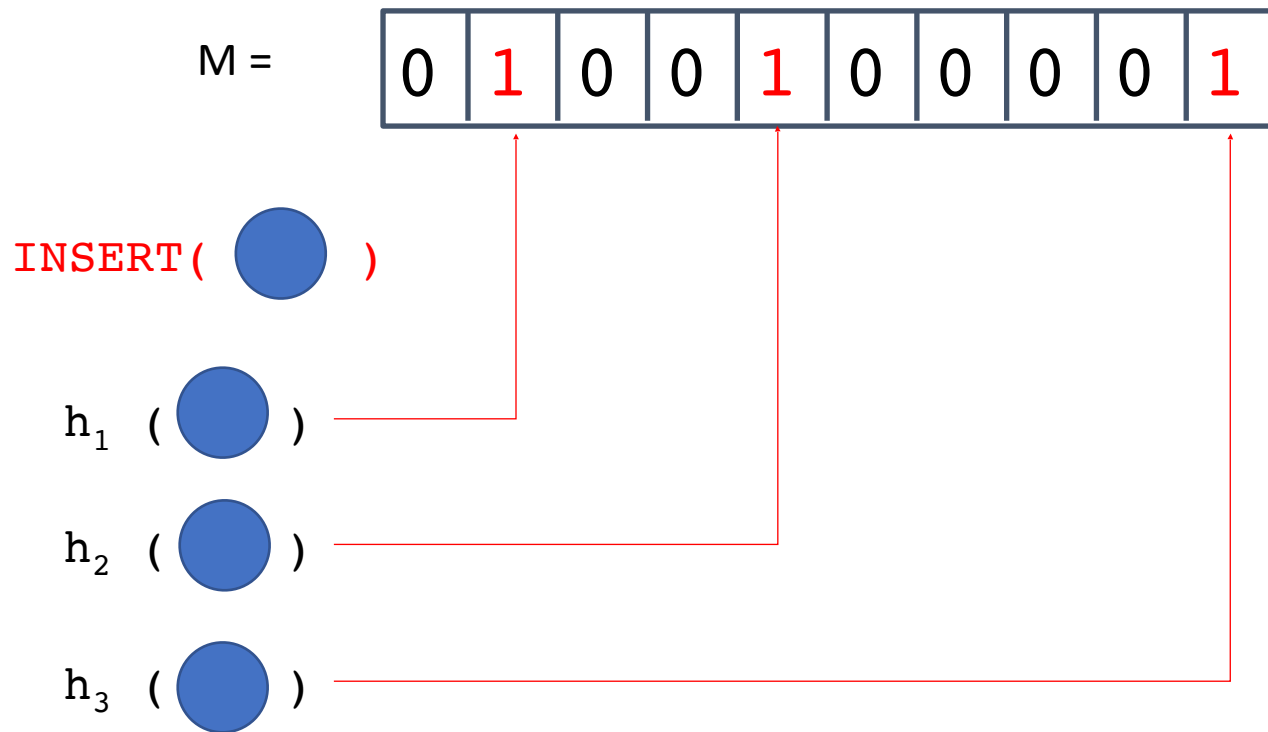
INSERT()

h_1 ()

h_2 ()

h_3 ()

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



Set: ●

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

M =

0	1	0	0	1	0	0	0	0	1
---	---	---	---	---	---	---	---	---	---

INSERT()

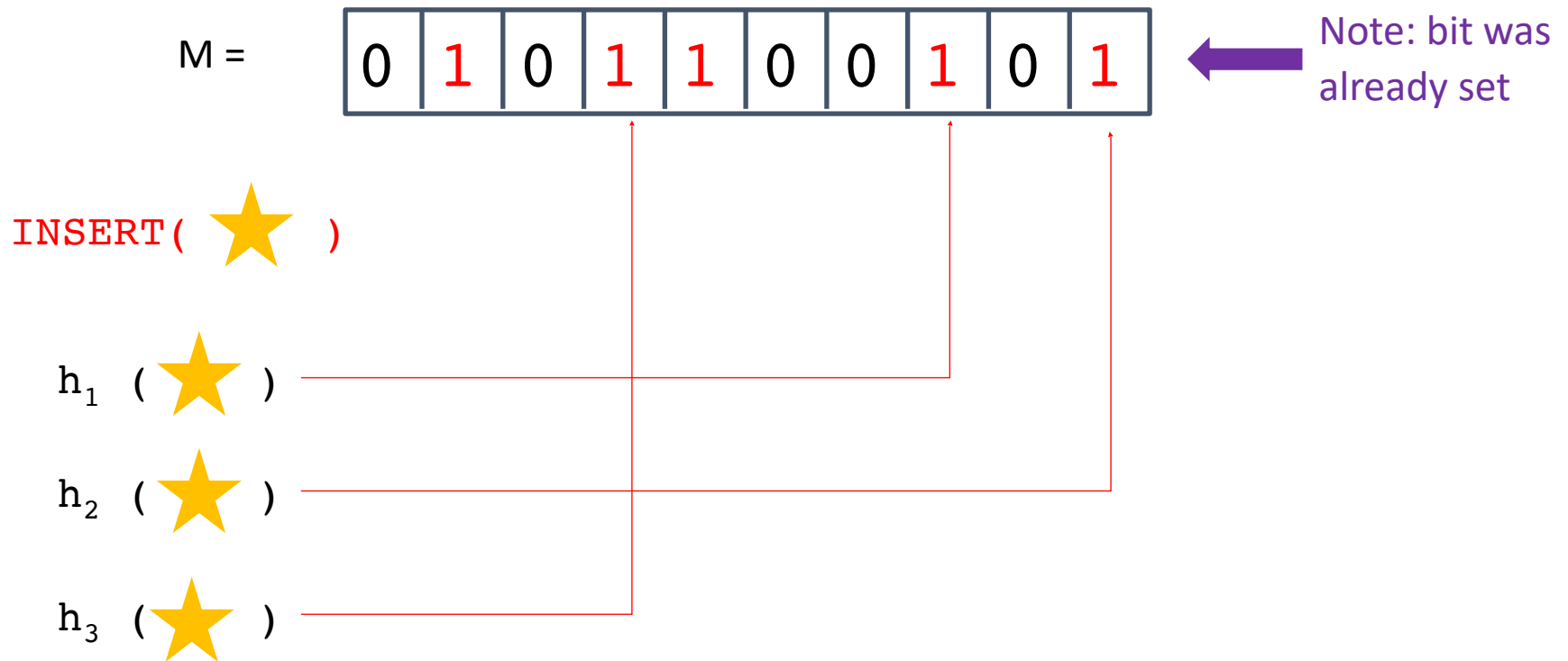
h_1 ()

h_2 ()

h_3 ()

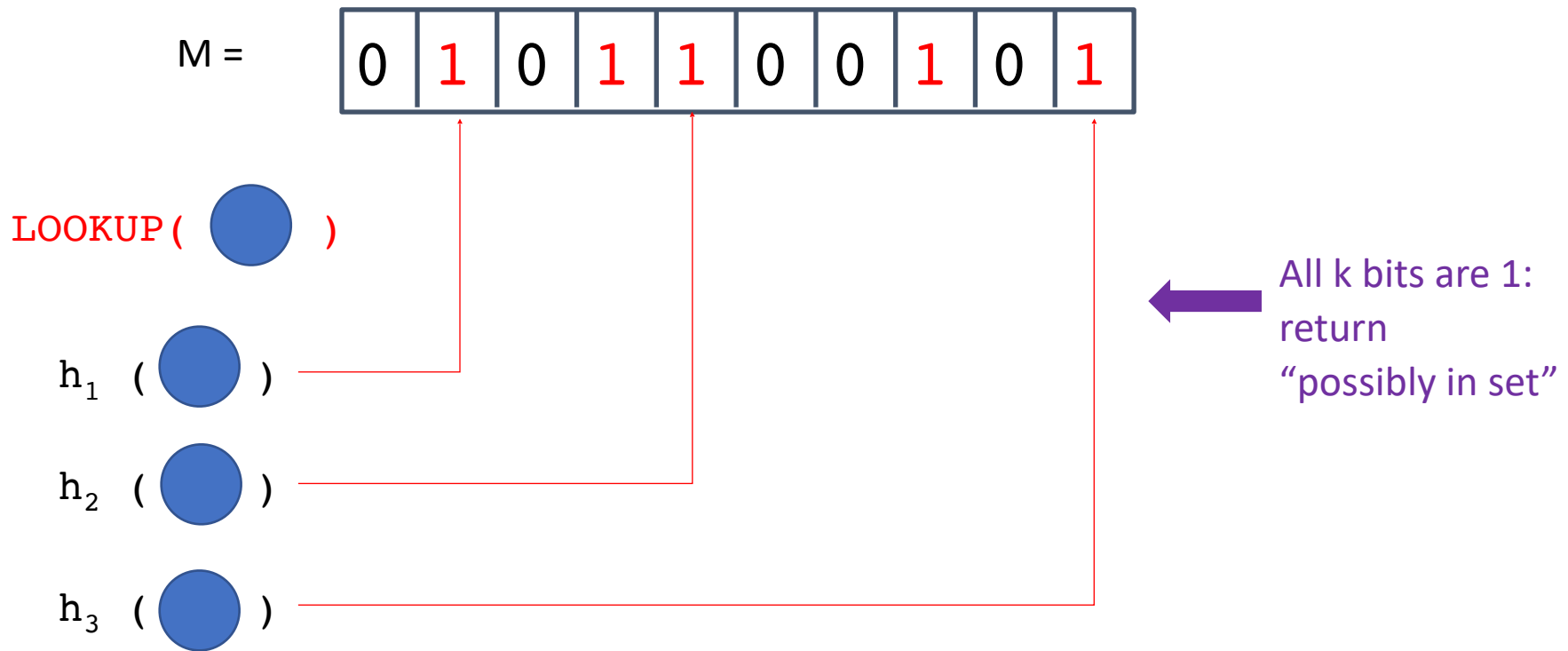
Set: 

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



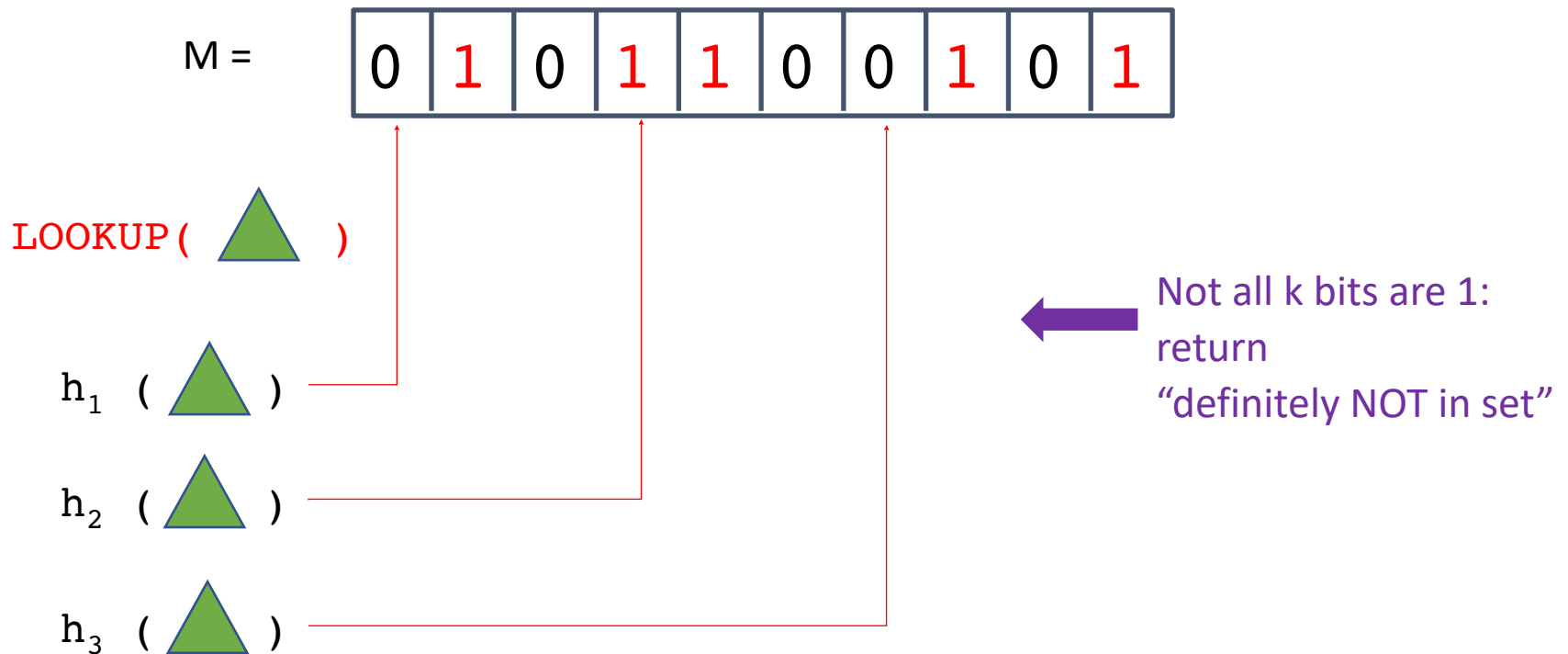
Set: ● ★

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



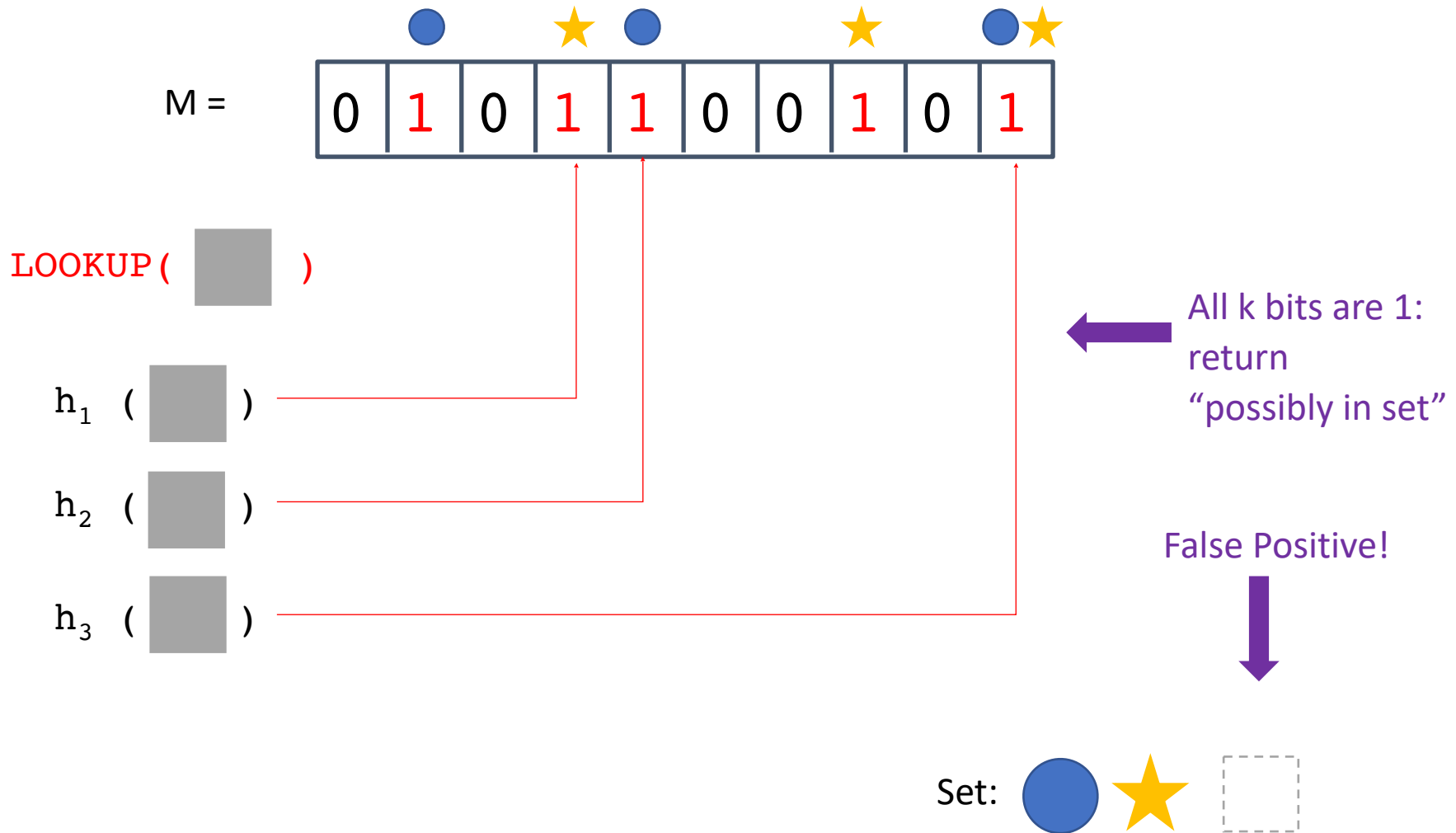
Set: ● ★

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



Set:  

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:

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

P(a given bit is still 0 after n insertions with k independent 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?

Bloom Filters: Challenges

- How do you grow a Bloom filter?
 - Short answer: you can't
 - Filter only stores bits: no way to “invert” bits to recover items
 - Longer answer: rebuild
 - If you wanted to grow a Bloom filter, you could allocate a new (empty) filter of the target size, then read through all items and insert them to the new filter
 - Note: the underlying data may or may not be available!

Bloom Filters: Challenges

- What if your filter itself exceeds RAM?
 - What does the cache behavior look like?
 - Good hash functions intentionally create a uniform distribution to avoid “clumping”
 - So even if the filter fits in RAM, the cache locality is poor
 - If the data set is truly large, there are a few options:
 - Use fewer bits per item (sacrifice precision)
 - Tolerate higher false positive rates
 - Use caching techniques, adding potential for expensive misses

Bloom Filters: Challenges

- What operations do they support/not support?
 - insert? Yes!
 - query? Yes!
 - delete? No! (Multiple items may have “set” any given bit)
 - rename? No! (rename = delete + insert)
 - “count”? No! (maybe/no answers only)

Bloom filter extensions that add support for additional operations do exist, but these operations are not supported by the standard data structure.

Filter Case Study: Quotient

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:

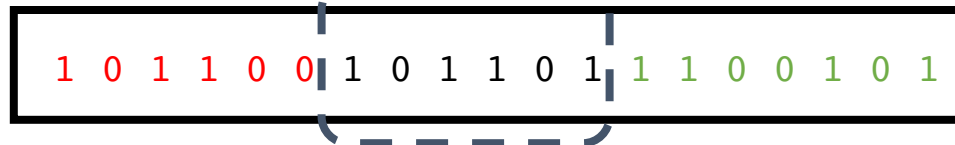
1 0 1 1 0 0 1 0 1 1 0 1 1 1 0 0 1 0 1

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: 

Hash:



Quotient: q most significant bits

Remainder: r least significant bits

Building a Quotient Filter

- The **quotient** is used as an index into an **m**-bucket array, where the **remainder** is stored.
 - Essentially, filter is 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

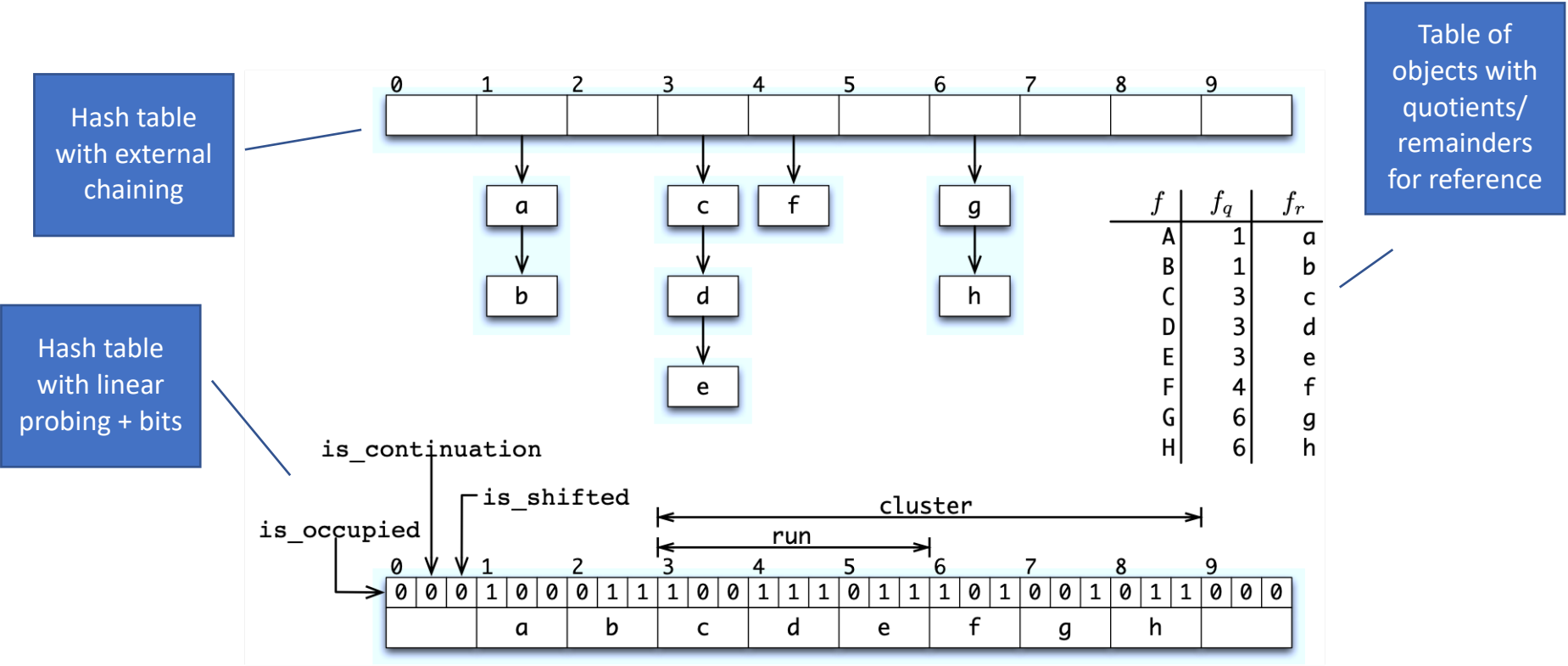


Figure 1: An example quotient filter and its representation.

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

Quotient Filter Example

Example uses decimal digits for readability

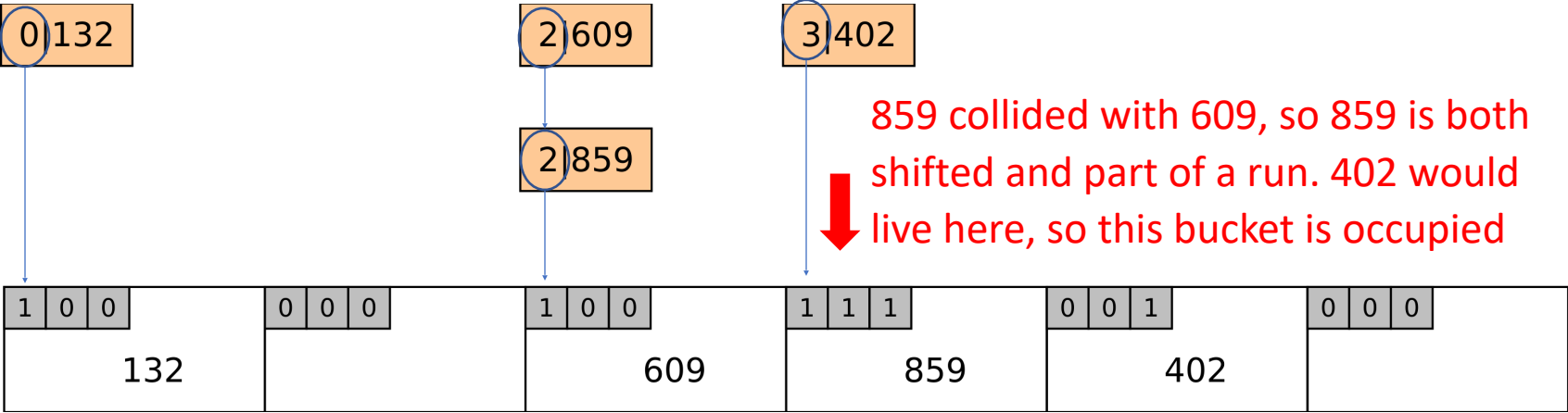
Parameters:

$q = 1$ "bit"


$r = 3$ "bits"

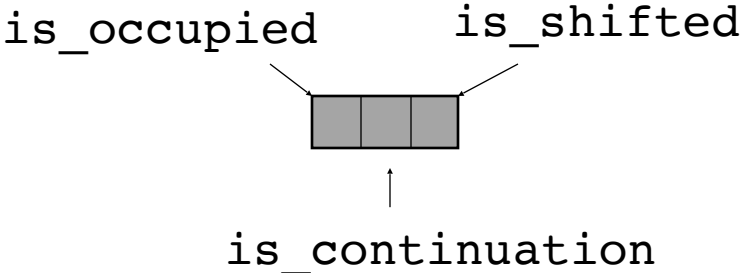
	Quotient (Bucket Index)	Remainder (Item Stored)
put (0132)	0	132
put (2859)	2	859
put (2609)	2	609
put (3402)	3	402

Quotient Filter Example



Collision, but 609 is in its canonical slot,  so `is_occupied` is set

 402 did not collide with any elements, but it was shifted from its canonical slot by 609 and 859.



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/remainers
 - 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?

- QF supports deletes
- QF supports “merges”
- QF has 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 are an exciting data structure for large key-value stores!

Cascade Filter

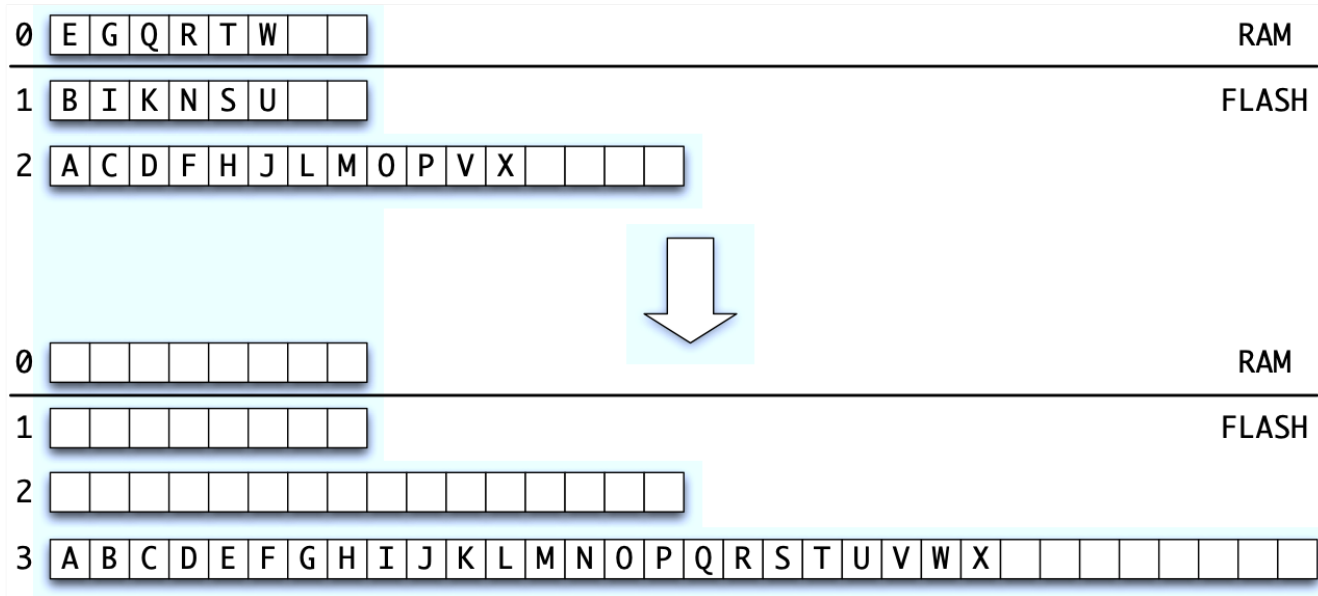


Figure 2: Merging QFs. Three QFs of different sizes are shown above, and they are merged into a single large QF below. The

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

Inserts are fast (duplicates are OK, so inserts only touch RAM level)
Lookups do 1 I/O per level

What next?

- CS256 Unlocks new Williams CS courses:
 - <https://www.cs.williams.edu/~jannen/teaching/cs-prereqs.svg>
- CS256 Prepares you for CS research
 - NSF REUs
 - Williams Summer Research
 - Thesis
- CS256 helps you think in new ways about the world

Thank you!

Artie ->

