Lecture 8: Bloom Filters and Cuckoo Filters

Sam McCauley

October 4, 2024

Williams College

Admin

- Assignment 1 deadline extended to Saturday
- Any lingering Assignment 1 questions?
- Today we'll talk about Homework 3 (in case Mountain Day is Friday)
- Lecture notes for today's class posted

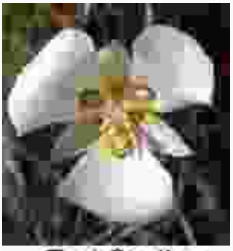
Prep for Homework 3

- Short lab video to help with setting up hashes

- (I made it a couple years ago; there may be very small differences with the current course. E.g. I call it an “assignment” even though it’s a homework. But the main ideas should still be useful.)

Filters: Goals for Today

What we want



- *Worst-case* compression
- Lossy compression with algorithmic guarantees
- That is to say: we know what we're losing and what we're not

Filter

- Stores a set S of size n
- Answers queries q of the form: “is $q \in S$?”
  - Really just a very simple dictionary that only returns whether or not a key exists (no values)
- All elements $x \in S$ and all queries must be from some universe U
- (Only need U to make sure that we can hash everything.)

Filter Guarantees

Guarantee 1 (No False Negatives)

A filter is always correct when it returns that $q \notin S$.
Equivalently, if we query an item $q \in S$, then a filter will always correctly answer $q \in S$.

Guarantee 1 Sanity check

- Can you create a *very* simple data structure that has no false negatives?
- Easiest option: my data structure stores nothing. On every query q, my data structure responds "$q \in S$."
- Another easy option: I store the entire set S using a standard dictionary (perhaps using a hash table). On a query q, I look it up and give the correct answer.

Filter Guarantees

Guarantee 2 (Bounded False Positive Rate)

A filter has a false positive rate ε if, for any query $q \notin S$, the filter (incorrectly) returns "$q \in S$" with probability ε.
We want our filter to have a false positive rate $\varepsilon < 1$.

The filters we will talk about today will work for *any* false positive rate ε, so long as $1/\varepsilon$ is a power of 2.[1]

So we can, if we want, guarantee a false positive rate of $1/2$, or $1/1024$—whatever is best for your use case.

[1] The cuckoo filter will actually need $1 + 1/\varepsilon$ to be a power of 2.

Guarantee 2 Sanity check

- Can you create a very simple data structure that has a good false positive rate?

- I store the entire set S using a standard dictionary (perhaps using a hash table). On a query q, I look it up and give the correct answer. This satisfies Guarantee 2 with $\varepsilon = 0$.

Tradeoff

- Obviously, smaller ε is better-it means we make fewer mistakes.
- So what's the tradeoff?
- We tradeoff space versus accuracy using ε.
  - Smaller ε means the compression is not as lossy
  - We make fewer mistakes, but we need more space
  - Larger ε means more aggressive compression
  - Space is very small, but filter is very inaccurate!
- A filter generally requires $O(n \log 1/\varepsilon)$ bits of space.

Space bounds

We talk about two filters today:

- A Bloom filter requires $1.44n \log_2(1/\varepsilon)$ bits of space.
- The cuckoo filter uses $1.05n \log_2(1 + 1/\varepsilon) + 3.15n$ bits of space.

How can we interpret this?

- Plugging in numbers: if we have a cuckoo filter with $\varepsilon = 1/63$, the filter takes less than 1 byte of space per element being stored.
- Notice that this space does *not* depend on the size of the original elements. We can store very long strings and still require only one byte per string stored.

History and Discussion

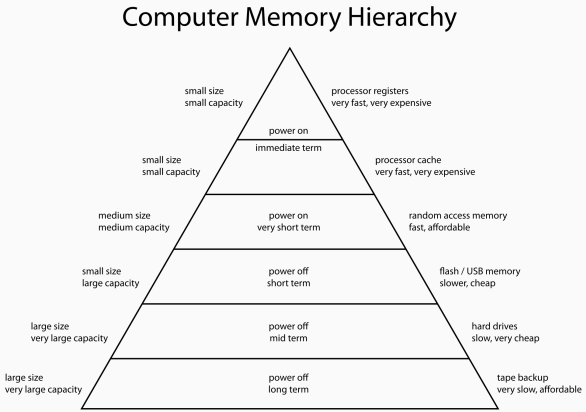
Bloom filter

- Invented by Burton H. Bloom in 1970
- Original publication only talked about good practical performance; theoretical analysis came later.

Cuckoo filter

- Invented by Fan et al. in 2014
- Provides better space usage for small ε (i.e. when the compression is not too lossy)
- Requires fewer hashes; has better cache performance.

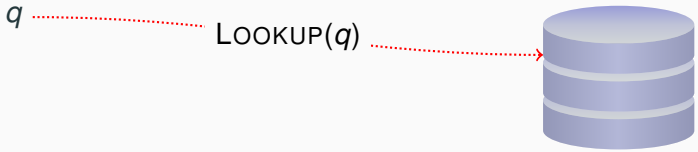
When should you use a filter?



1st example: avoiding cache misses

- Let's say we have a very large table of data
- Large enough that it doesn't fit in L3
  - Maybe it doesn't even fit in RAM
- Frequently query items not in the table

Common filter usage



Queries to the entire dataset are very expensive!

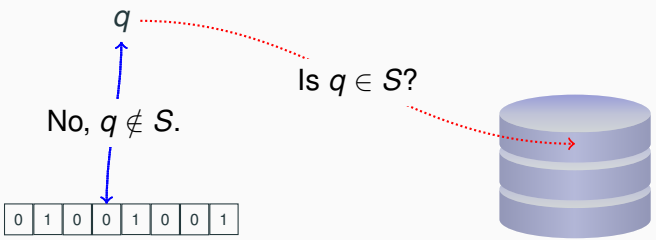
Queries are often “unnecessary”

Many workloads involve mostly “negative” queries: queries to keys not stored in the table. (query $q \notin S$)

- Classic example: dictionary of unusually-hyphenated words for a spellchecker.
- Checking if key already exists before an insert (deduplication in general)
- Check for malicious URLs
- Table with many empty entries

Classic filter usage: succinct data structure that will allow us to “filter out” negative queries.

Common filter usage



Filters are so small that they can fit in local memory.

Filters can be used to avoid unnecessary accesses to the table for negative queries, improving performance.

If $q \notin S$ (false positive), still do an unnecessary access.

Common filter usage

- With $O(n \log 1/\varepsilon)$ local memory (perhaps fitting in L3 cache), can filter out $1 - \varepsilon$ cache misses for keys $q \notin S$.

- Greatly reduces number of remote accesses, thereby reducing time.

When should you use a filter?

2nd example: Approximately storing a set

- Before, we stored the actual set S. (It was expensive to access, but we stored it.)
- But what if we don't want to?
- Example: approximate spell checker

Approximate spell checker

- Want to build a spell checker; don't have room to store dictionary
- Store the words in a filter. What do our guarantees mean?
- Guarantee 1: if we query a correctly-spelled word, it is never marked as misspelled
- Guarantee 2: if we query a misspelled word, we only miss it (don't mark it misspelled) with probability ε
- Using only a byte or so per item, can do almost as well as storing a full dictionary! (Roughly 98% accuracy.)

Bloom Filters

Bloom Filter

A Bloom filter consists of:

- $k = \log_2 1/\varepsilon$ hash functions, which I will denote using $h_1, h_2, \ldots, h_k$,

$$3x^2 \in R \subset Q.$$

- Bit array A of $m = nk \log_2 e \approx 1.44n \log_2 \frac{1}{\varepsilon}$ bits.
  - Since we're doing compression, we measure space in *bits*, and track constants
- For each $i = 1, \ldots, k$, $h_i : U \to \{0, \ldots, m-1\}$ (that is to say, h_i maps an element from the universe of possible elements U to a slot in the hash table).
- Assume $1/\varepsilon$ is a power of 2; round m up to the nearest integer

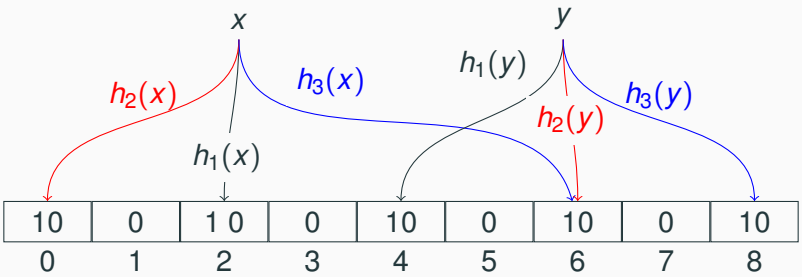
Building a Bloom Filter

- Begin with $A[i] = 0$ for all i. (Basically, just calloc the bit array.)

- Then add the items one at a time by setting *all* their slots to 1:

```
1  for each x in S:
2      for i = 1 to k:
3          A[h_i(x)] = 1
```

Building a Bloom Filter



Inserting two elements x and y into a Bloom filter with $\varepsilon = 1/8$. We have three hash functions, and (rounding up) the array is of length $m = 9$ bits.

Invariant

- What invariant does this data structure satisfy?

Invariant 1

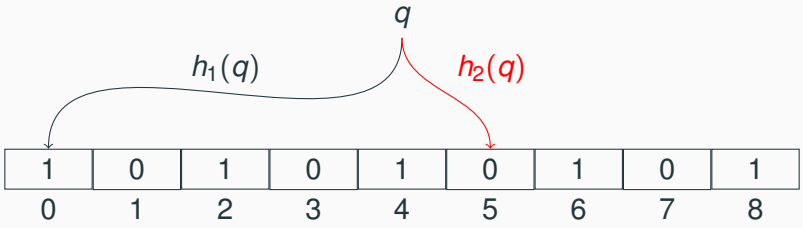
A Bloom filter storing a set S using hashes $h_1, \ldots h_k$ satisfies $A[h_i(x)] = 1$ for all $x \in S$ and all $i \in \{1, \ldots, k\}$.

Querying a Bloom filter

On a query q, we check all the hash slots to see if any stores 0:

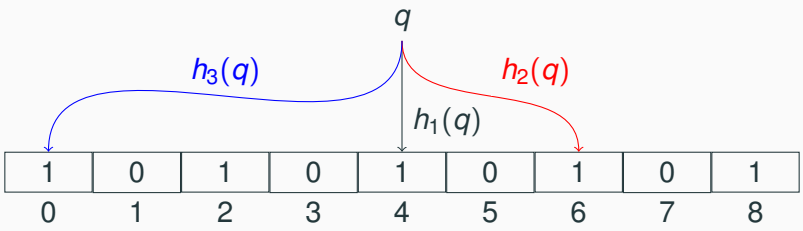
```
1  for i = 1 to k:
2      if A[h_i(q)] == 0:
3          return false //q is not in S
4
5  // we have A[h_i(q)] = 1 for all h_i
6  return true //q is in S
```

Query example



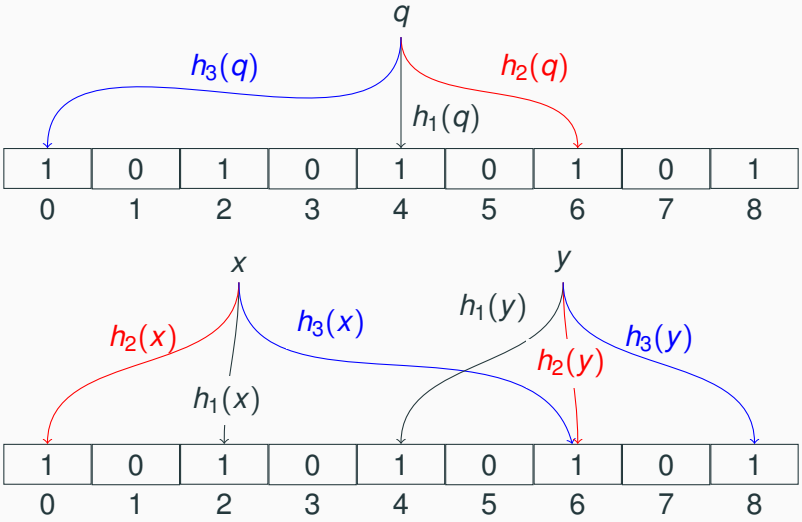
An example query to an element not in the set; $k = 3$.

Query example 2



An example false positive query.

Query example 2



Discussion

- Can we insert into a Bloom filter?
  - Yes, but performance degrades as it fills up. We are OK so long as no more than n items are inserted.

- Can we delete?
  - No. If we flip a bit from 1 to 0, it may cause a false negative, violating Guarantee 1.

Bloom filter analysis

- Assume our hashes h_i are perfectly uniform random: any $x \in U$ is mapped to any hash slot $s \in \{0, \ldots, m-1\}$ with probability $1/m$; independently of any other hash.

- Let's strategize: what about the Bloom filter can we use to prove that Guarantee 1 and Guarantee 2 hold?

Guarantee 1

Guarantee (No False Negatives)

If we query an item $q \in S$, then a filter will always answer $q \in S$.

- By the Bloom filter Invariant, if $q \in S$, then $A[h_i(q)] = 1$ for all $i \in \{1, \ldots k\}$.
- This means that the query algorithm always returns "$q \in S$."

Guarantee 2 (False positive rate)

Guarantee (Bounded False Positive Rate)

A filter has a false positive rate ε if, for any query $q \notin S$, the filter (incorrectly) returns "$q \in S$" with probability ε.

High-level argument:

- Assume: each entry of A is 1 with probability $1/2$
- Only get a false positive if every bit is a 1
- Are these events independent?
  - No! But it seems like the independence isn't too big of a deal...let's assume they're independent for now.
- Occurs with probability $(1/2)^k = (1/2)^{\log_2(1/\varepsilon)}$
- $(1/2)^{\log_2(1/\varepsilon)} = \varepsilon$.

Cuckoo Filter

Assignment 3

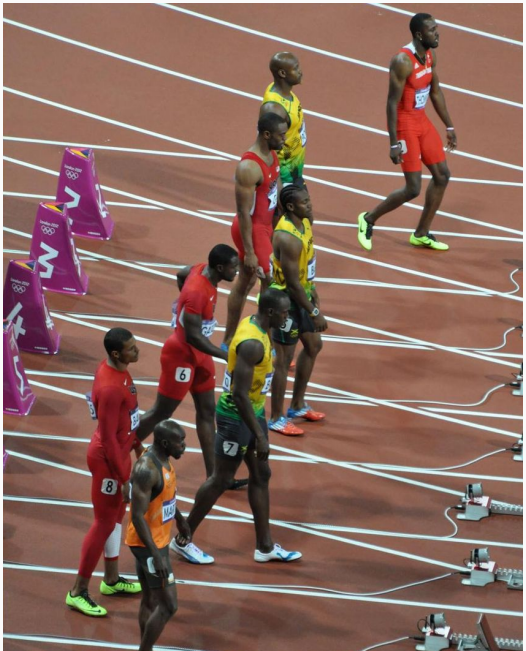
- In short: you'll implement a cuckoo filter to speed up a sequence of dictionary queries
- You're looking for "bilingual palindromes": strings whose reverse is a word in another language
- Most words are not bilingual palindromes, so a filter can significantly speed up queries

Cuckoo Filter

A cuckoo filter consists of:

- k hash functions denoted by $h_1, h_2, \ldots, h_k$ (k is a constant)
  - We'll only use *one* of these hash functions (h_1) in our implementation!
- a fingerprint hash function f that takes an item from the universe and outputs a number from 1 to $1/\varepsilon$ (we'll call this number the *fingerprint* of the item)
- a cuckooing hash function h that takes in a fingerprint and outputs a number from 1 to m, and
- a hash table T of m slots, where each slot has room for $\log_2(1 + 1/\varepsilon)$ bits.

Some initial parameters



- $k = 2$ hash functions (for now)
- $m = 2n$ slots
- These parameters are easy to analyze, but space inefficient. We'll fix it later.
- Also assume that $1/\varepsilon + 1$ is a power of 2, and m is a power of 2.

Initializing a Cuckoo Filter

- Make sure all slots of T are empty
- Today: we'll set all slots to 0. A slot in T is nonempty if and only if it stores a number larger than 0.

Invariant 2

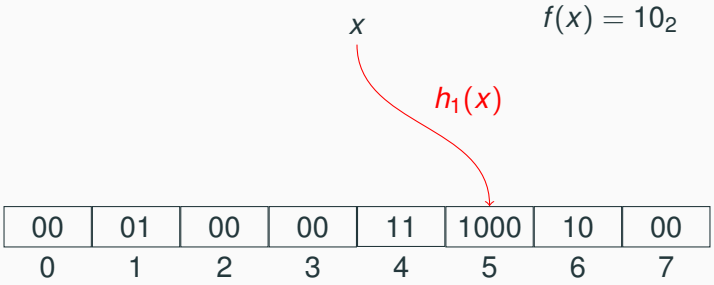
For any $x \in S$, either slot $h_1(x)$ or $h_2(x)$ stores the fingerprint $f(x)$.

Question: with this invariant, how can we query to avoid false negatives?

Inserting into a Cuckoo Filter

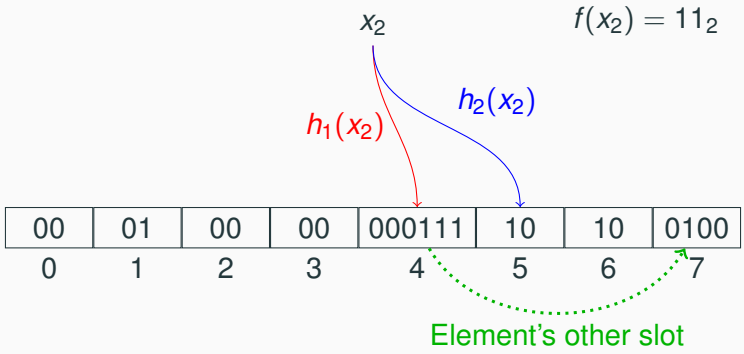
- If there is an h_i such that $T[h_i(x)]$ is nonempty, then store $f(x)$ in $T[h_i(x)]$.
- Otherwise, we cuckoo:
  - Choose some $i \in \{1, \ldots, k\}$
  - Let's say that x_1 is the element stored in $T[h_i(x)]$.
  - Then we store $f(x)$ in $T[h_i(x)]$ and "cuckoo" x_1 to another slot
- If we cuckoo more than $\log n$ elements, we rebuild the filter.

Cuckoo Filter Insert First Attempt



A cuckoo filter with $\varepsilon = 1/3$ and $k = 2$.

Cuckoo Filter Insert First Attempt



A cuckoo filter with $\varepsilon = 1/3$ and $k = 2$.

Is our invariant maintained?

Implementing Insertions

There's a problem with what I said!

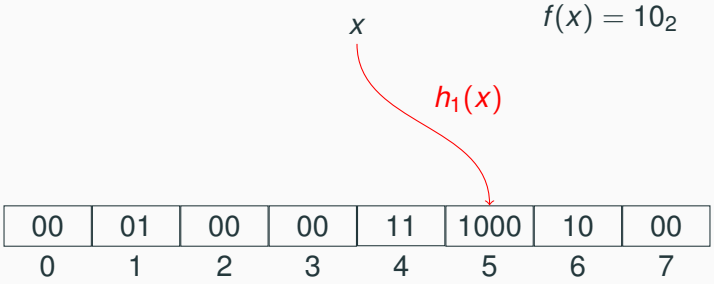
- We don't have access to the element that hashed to that slot. So how can we calculate its other hash?
- If $k = 2$, we can use *partial-key cuckoo hashing*.
- Only use one hash h_1 for slots. But then, have a second hash h that maps a *fingerprint* to a number from 1 to m.
- Set $h_2(x) = h_1(x) \wedge h(f(x))$. (XOR)
- Note that then $h_2(x) \wedge h(f(x)) = h_1(x) \wedge h(f(x)) \wedge h(f(x)) = h_1(x)$.

Cuckooing

So to cuckoo a fingerprint ϕ stored in a slot *s* to its other location:

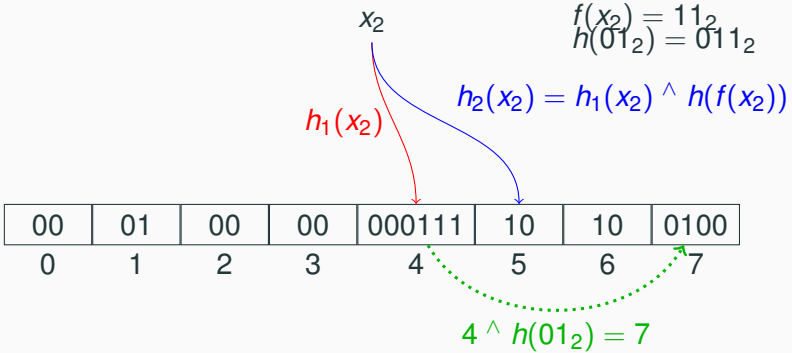
- Calculate $h(\phi)$
- Its other slot is $s \wedge h(\phi)$.
- If that other slot is empty we can store ϕ in it (woo)! Otherwise, take the fingerprint stored there and cuckoo it to its other slot.

Cuckoo Filter Insert Example With Partial-Key Cuckoo Hashing



A cuckoo filter with $\varepsilon = 1/3$ and $k = 2$.

Cuckoo Filter Insert Example 2



A cuckoo filter with $\varepsilon = 1/3$ and $k = 2$.

Cuckoo Filter Invariant

Using partial-key cuckoo hashing with $k = 2$:

Invariant 3

For any $x \in S$, either slot $h_1(x)$ or $h_2(x) = h_1(x) \wedge h(f(x))$ stores the fingerprint $f(x)$.

For higher k:

Invariant 4

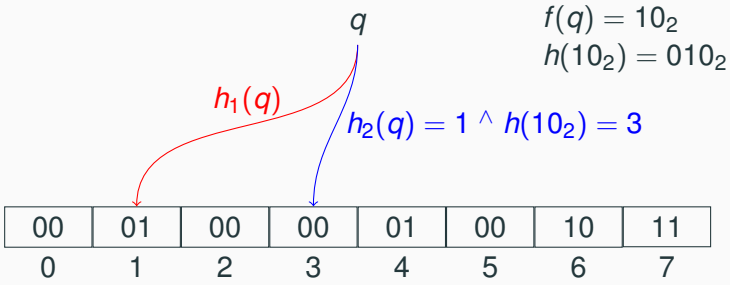
For every $x \in S$, there exists an $i \in \{1, \ldots, k\}$ such that $f(x)$ is stored in $T[h_i(x)]$.

Querying a Cuckoo Filter

To query an element q:

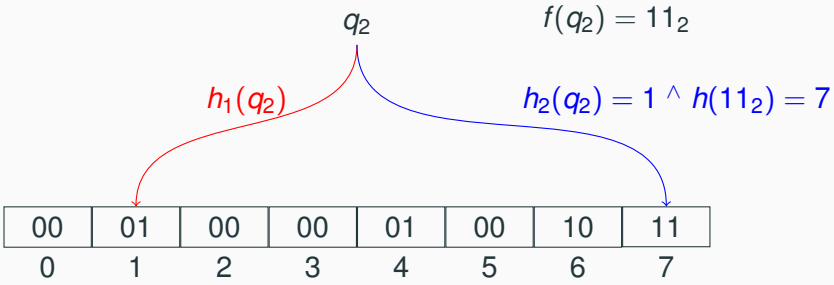
```
1 for i = 1 to k:
2     if T[h_i(q)] = f(q):
3         return true // q is in S
4 //did not find the fingerprint in any slot
5 return false // q is not in S
```

Querying a Cuckoo Filter: Example



Querying a cuckoo filter with $\varepsilon = 1/3$ and $k = 2$.

Querying a Cuckoo Filter: Example 2



Querying a cuckoo filter with $\varepsilon = 1/3$ and $k = 2$.

Discussion

- Can a cuckoo filter handle inserts?
  - Yes! But as we insert more and more elements the number of cuckoos we expect will get larger and larger (and higher probability of a cycle of cuckoos)
- How about deletes?
  - Oftentimes yes—if you are careful! (Need to make sure we don't delete another element's fingerprint.)

Implementing Effective Hash Functions

Practical Hash Functions

- We'll talk about this next lecture in more detail
- I will post a video with the assignment that also focuses on this
- I want to go over the basics so that you can get started on the assignment in case Friday is mountain day

Hashes we need

- h_1 which maps an arbitrary element (a string in Homework 3) to a slot in the hash table
- f which maps an arbitrary element (a string in Homework 3) to a number from 1 to 255 (we'll be doing 8-bit fingerprints)
- h which maps a fingerprint from 1 to 255 to a slot in the hash table

Implementing h

- h is easy because it only needs 255 values
- I give you an array of random values in the starter code (let's take a look)
- To calculate $h(i)$, for $i \in \{1, \ldots, 255\}$, just use `hashFingerprint`$[i - 1]$

Implementing h_1 and f

- murmurhash: a popular, fast, hash function that does a good job of “acting random”
- Will be given to you as part of your starter code
- murmurhash outputs 128 bits. We’ll use the first 32 bits as h_1, and the second 32 bits as f
- Use mod to get them down to size

Calling Murmurhash

```
1  uint32_t hash[4] = {0,0,0,0};
2  MurmurHash3_x64_128(word, length, seed, hash);
```

- `word` is the string you would like to hash
- `length` is the length of word (murmurhash does not check for null-termination!)
- `seed` is the hash function seed (pick a large random number; keep it consistent)
- `hash` is the 128 bits of output

```
1  uint32_t position = hash[0] % numSlots;
2  uint32_t fingerprint = 1 + hash[1] % fingerprintMask;
```

Cuckoo Filter Analysis

Union Bound

- Simple but useful tool in randomized algorithms
- *Always* works, even for events that are not independent
- Sometimes called “Boole’s inequality”

Theorem 1

Let X and Y be random events. Then

$$\Pr(X \text{ or } Y) \leq \Pr(X) + \Pr(Y).$$

More generally, if $X_1, X_2, \ldots, X_k$ are any random events, then

$$\Pr(X_1 \text{ or } X_2 \text{ or } \ldots \text{ or } X_k) \leq \sum_{i=1}^{k} \Pr(X_k).$$

Union Bound Example

- Let’s say I have 10 students in a course, and I randomly assign each student an ID between 1 and 100 (these IDs do not need to be unique).

- Can you upper bound the probability that some student has ID 1?

Exact Analysis of Student ID Problem

- The probability that at least one student has ID 1 is

$$1 - \Pr(\text{no student has ID 1}).$$

- The probability that a single student has an ID other than 1 is 99/100.
- Thus, the probability that all 10 students have an ID other than 1 is $(99/100)^{10}$.
- Thus, the probability that at least one student has ID 1 is $1 - (99/100)^{10} \approx 9.56\%$.

Exact Analysis of Student ID Problem

- The probability that at least one student has ID 1 is

This is messy! And it would be even worse if the IDs were not independent!

The union bound lets us avoid this work.

- The probability that a single student has an ID other than 1 is 99/100.
- Thus, the probability that all 10 students have an ID other than 1 is $(99/100)^{10}$.
- Thus, the probability that at least one student has ID 1 is $1 - (99/100)^{10} \approx 9.56\%$.

Union Bound Analysis of Student Problem

- The probability that a given student has ID 1 is 1/100.
- From Union bound: The probability that *any* student has ID 1 is at most the sum, over all 10 students, of 1/100.
- This gives us an upper bound of $10/100 = 10\%$.

Analysis of Cuckoo Filters

Some assumptions going in:

- all hash functions h_i are uniformly random: any $x \in U$ is mapped to any hash slot $s \in \{0, \ldots, m-1\}$ with probability $1/m$.
- Same for the fingerprint hash f: any $x \in U$ is mapped to a given fingerprint $f_x \in \{1, \ldots, 1/\varepsilon\}$ with probability ε.
- We will analyze *without* partial-key cuckoo hashing (we'll assume independent h_1 and h_2)

First Guarantee: No False Negatives

Guarantee (No False Negatives)

A filter is always correct when it returns that $q \notin S$.
Equivalently, if we query an item $q \in S$, then a filter will always correctly answer $q \in S$.

Invariant

For every $x \in S$, there exists an $i \in \{1, \ldots, k\}$ such that $f(x)$ is stored in $T[h_i(x)]$.

- We can see that the invariant means that there are no false negatives.

Second Guarantee: False Positive Rate

Guarantee 3 (False Positive Rate)

A filter has a false positive rate ε if, for any query $q \notin S$, the filter (incorrectly) returns "$q \in S$" with probability ε.

- A query $q \notin S$ is a false positive if, for some h_i, $T[h_i(q)] = f(q)$.
- Let's examine each hash h_1 and h_2 individually.

Second Guarantee: False Positive Rate

- Let's start with h_1. What is the probability $T[h_1(q)]$ contains a fingerprint?
- $1/2$, because we are storing n elements in $2n$ slots.
- If $T[h_1(q)]$ contains a fingerprint, the probability that $f(x) = f(q)$ is ε.
- Therefore, the probability that $T[h_1(q)]$ contains a fingerprint $f(x) = f(q)$ is $\varepsilon/2$.

Second Guarantee: False Positive Rate

- What about h_2?

- Same exact analysis: probability that $T[h_2(q)]$ contains a fingerprint $f(x) = f(q)$ is $\varepsilon/2$.

Second: Guarantee: Putting it Together

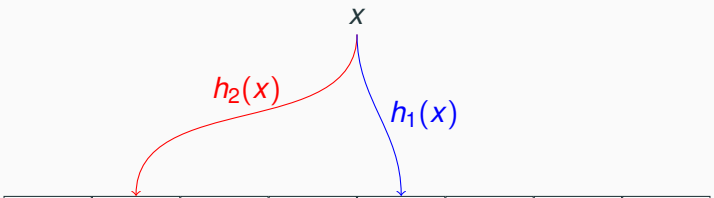
- q is a false positive if either $T[h_1(q)]$ contains a fingerprint $f(x_1)$ such that $f(x_1) = f(q)$, or $T[h_2(q)]$ contains a fingerprint $f(x_2)$ such that $f(x_2) = f(q)$
- Each happens with probability at most $\varepsilon/2$
- By union bound, one or the other happens with probability at most $\varepsilon/2 + \varepsilon/2 = \varepsilon$.

Improved Cuckoo Filter Performance

Improving the Cuckoo Filter

- Currently, have $m = 2n$ slots, so the space is $2n \log_2(1/\varepsilon)$.
- Here is one way to improve that:
- Store room for *four* fingerprints in each hash slot, and make the fingerprints hash to $\{1, \ldots, 8/\varepsilon\}$. Assume that $8/\varepsilon + 1$ is a multiple of 2.
- Then can set $m = 1.05n/4$, giving total space usage $1.05n \log_2(8/\varepsilon + 1) \approx 1.05n \log_2(1/\varepsilon) + 3.15n$.

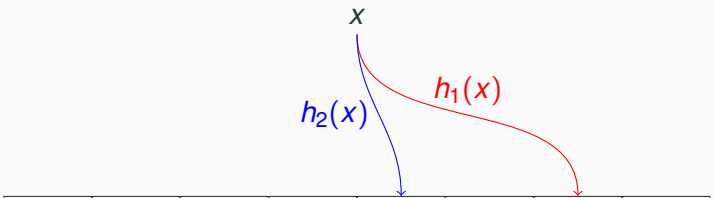
Example



1000	1010	0000	0000	0101	0000	0100	0000
0000	0000	0000	0000	1001	0000	0110	0000
0000	0000	0000	0000	0010	0000	0101	0000
0000	0000	0000	0000	1001	0000	1111	0000
0	1	2	3	4	5	6	7

A cuckoo filter with fingerprints of length 4, $k = 2$, and 4 slots per bin.

Example 2



1000	1010	0000	0000	0101	0000	0100	0000
0000	0000	0000	0000	1001	0000	0110	0000
0000	0000	0000	0000	0010	0000	0101	0000
0000	0000	0000	0000	1001	0000	1111	0000
0	1	2	3	4	5	6	7

A cuckoo filter with fingerprints of length 4, $k = 2$, and 4 slots per bin.

Comparing the Two Filters

Bloom filters:

- Easy to implement
- Fairly efficient for large ε

Cuckoo filters:

- Much more space efficient
- Only require 2 hash functions (may improve practical performance)
- Good cache efficiency: only need to access the hash table 2 times, rather than $\log_2(1/\varepsilon)$.