Randomized Algorithms

Bloom filters: checking membership with tiny memory

Alex Gittens
Department of Computer Science, Rensselaer Polytechnic Institute

Fall 2026

Learning Objectives

By the end of this lecture, you should be able to:

1. **State** one-sided approximate membership and distinguish a per-query guarantee from a simultaneous guarantee over all nonmembers.
2. **Analyze** a fingerprint scheme and explain why comparing against n separate tags costs an extra $\log_2 n$ bits per key.
3. **Execute** Bloom-filter insertion and query, and explain how sharing one bit array avoids that separate-tag cost.
4. **Derive** $p_{\text{fp}} = \mathbb{E}[(Z/m)^k]$ and justify the approximation $p_{\text{fp}} \approx (1 - e^{-kn/m})^k$ using occupancy concentration.
5. **Choose** the number of hash functions and the table size for a target error δ.
6. **Prove** that any one-sided filter needs about $n \log_2(1/\delta)$ bits, independent of $|U|$ under the per-query model.

PART 1

The problem, and a first attempt

1

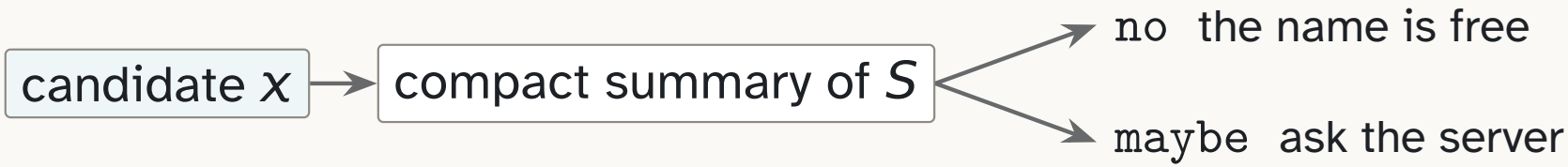
Checking usernames without the server

A service keeps the set

$$S = \{\text{usernames already taken}\} \subseteq U, \qquad |S| = n.$$

While someone types a proposed name, the browser should warn them about taken names without a round trip to the server on every keystroke.

The browser does not need profiles or passwords, and it never deletes names. It needs a fast local test, computed from a compact summary of S that the server sends once.



A full dictionary of S answers exactly, but it stores every username. How much smaller can the summary be if we allow some mistakes?

Which mistakes can we tolerate?

- Answering `no` for a taken name tells the user a taken name is free. We forbid this.
- Answering `maybe` for a free name costs one extra request to the server. That is fine if it is rare.

Truth	answer `no`	answer `maybe`
$x \in S$	**forbidden**	correct
$x \notin S$	correct	false positive

Which mistakes can we tolerate?

- Answering `no` for a taken name tells the user a taken name is free. We forbid this.
- Answering `maybe` for a free name costs one extra request to the server. That is fine if it is rare.

Truth	answer `no`	answer `maybe`
$x \in S$	**forbidden**	correct
$x \notin S$	correct	false positive

The guarantee is for one fixed nonmember

For every fixed $x \notin S$,

$$\mathbb{P}_H\{\text{query}_H(x) = \texttt{maybe}\} \le \delta.$$

Every $x \in S$ must always receive `maybe`.

Which mistakes can we tolerate?

- Answering `no` for a taken name tells the user a taken name is free. We forbid this.
- Answering `maybe` for a free name costs one extra request to the server. That is fine if it is rare.

Truth	answer `no`	answer `maybe`
$x \in S$	**forbidden**	correct
$x \notin S$	correct	false positive

The guarantee is for one fixed nonmember

For every fixed $x \notin S$,

$$\mathbb{P}_H\{\mathrm{query}_H(x) = \texttt{maybe}\} \le \delta.$$

Every $x \in S$ must always receive `maybe`.

This is not a simultaneous guarantee. We do *not* require

$$\mathbb{P}_H\{\text{every } x \notin S \text{ is answered } \texttt{no}\} \ge 1 - \delta.$$

Why might $|U|$ disappear from the space bound?

Usernames, URLs, and file paths can come from enormous universes. Suppose $|U|$ is astronomically large. Should the summary need more bits just because there are more possible nonmembers?

Recall the guarantee: we fix *one* $x \notin S$ and require

$$\mathbb{P}_H\{FP(x)\} \leq \delta.$$

Why might $|U|$ disappear from the space bound?

If instead we demanded one random summary that was simultaneously correct on every nonmember, then a union bound over $|U \setminus S|$ would naturally introduce the universe size.

$$\mathbb{P}_H\{\exists x \notin S : FP(x)\} \leq \sum_{x \notin S} \mathbb{P}_H\{FP(x)\}.$$

That is *not* our model.

Why might $|U|$ disappear from the space bound?

So it is plausible that the right space scale depends on n and δ, but not on $|U|$ or on the number of bits used to write a key.

Make a guess before we compute anything: with $n = 10^7$ usernames and $\delta = 1\%$, about how many bits per username should be necessary?

1. Try storing a short random fingerprint for every key.

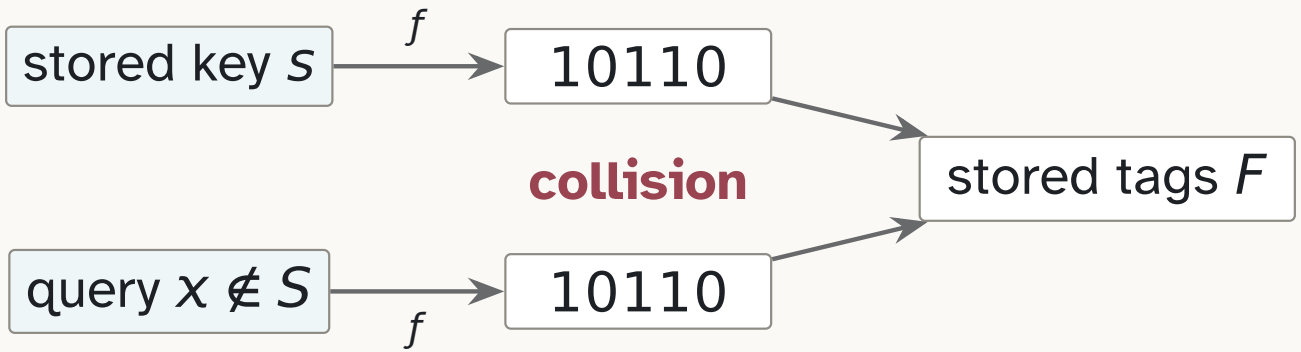
Why might $|U|$ disappear from the space bound?

First attempt: store short random tags

Choose a random hash function

$$f : U \to \{0,1\}^b$$

and store the set of tags $F = \{f(s) : s \in S\}$. Query x by checking whether $f(x) \in F$.



A stored key always finds its own tag, so there are no false negatives. A nonmember is a false positive exactly when its tag equals some stored tag.

How long must the tags be?

Fix $x \notin S$. By the union bound over the n stored keys,

$$\mathbb{P}\{f(x) \in F\} \le \sum_{s \in S} \mathbb{P}\{f(x) = f(s)\}.$$

Each term involves only one pair of keys. So f need not be fully random: it is enough that

$$\mathbb{P}\{f(x) = f(y)\} \le 2^{-b} \qquad \text{for every pair } x \neq y,$$

which is exactly what a universal hash family with range $\{0,1\}^b$ guarantees. Then

$$\mathbb{P}\{f(x) \in F\} \le \frac{n}{2^b} \le \delta \qquad \text{as soon as} \qquad b \ge \log_2 \frac{n}{\delta}.$$

Why does the fingerprint length grow with n?

For one fixed nonmember x, the union bound gave

$$\mathbb{P}\{f(x) \in F\} \leq \frac{n}{2^b}.$$

To make this at most δ,

$$b \geq \log_2 \frac{n}{\delta} = \log_2 n + \log_2 \frac{1}{\delta}.$$

Why does the fingerprint length grow with n?

What the tag length pays for	formula	example
collision with any of n stored tags	$\log_2 n$	about 23 bits

Why does the fingerprint length grow with n?

What the tag length pays for	formula	example
collision with any of n stored tags	$\log_2 n$	about 23 bits
target error for the query	$\log_2(1/\delta)$	about 7 bits

Why does the fingerprint length grow with n?

What the tag length pays for	formula	example
collision with any of n stored tags	$\log_2 n$	about 23 bits
target error for the query	$\log_2(1/\delta)$	about 7 bits
total tag length	$\log_2(n/\delta)$	30 bits

Why does the fingerprint length grow with n?

What the tag length pays for	formula	example
collision with any of n stored tags	$\log_2 n$	about 23 bits
target error for the query	$\log_2(1/\delta)$	about 7 bits
total tag length	$\log_2(n/\delta)$	30 bits

Why the $\log_2 n$ term? The query is compared with n separate stored tags. Any one collision is enough for a false positive, so each pairwise collision must have probability only about δ/n.

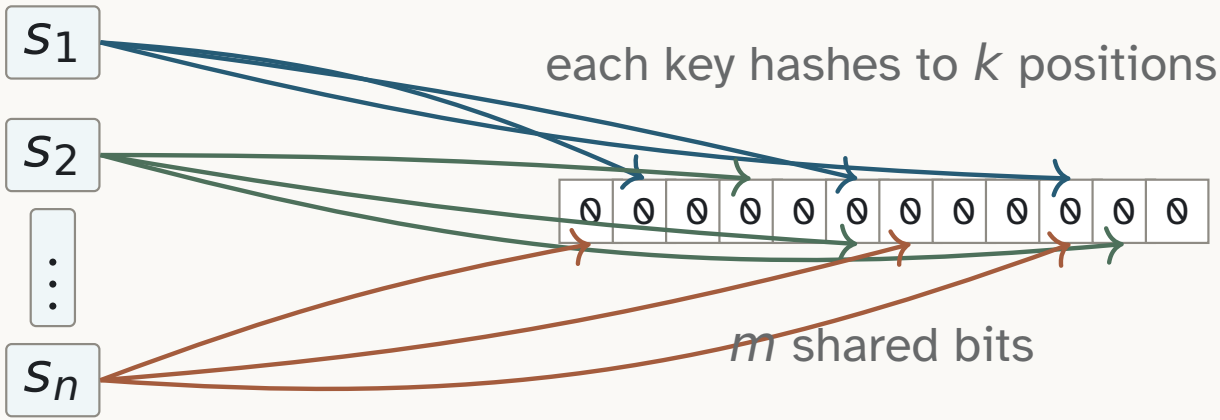
Can we stop paying separately for each stored key? Bloom filters do this by making all keys share the same bits.

PART 2

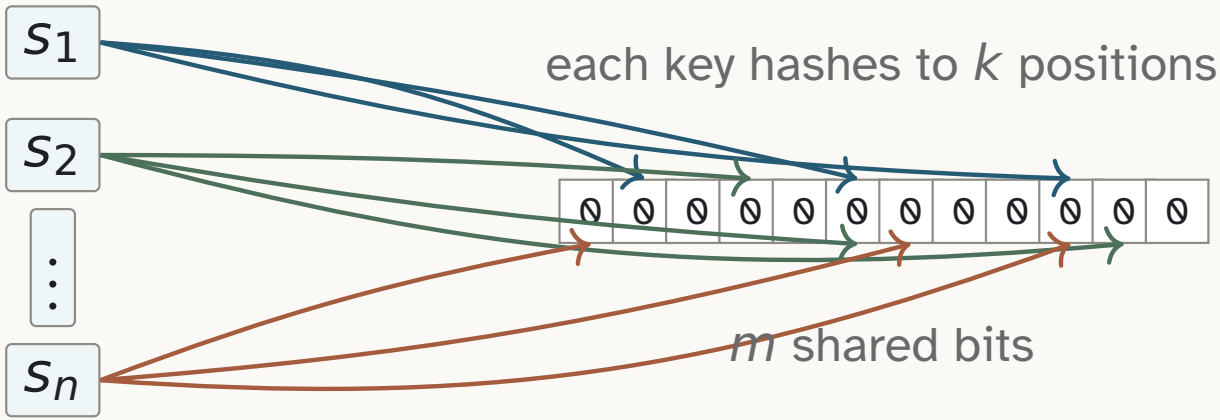
Bloom filters

2

Bloom filter: all keys share one bit array



Bloom filter: all keys share one bit array

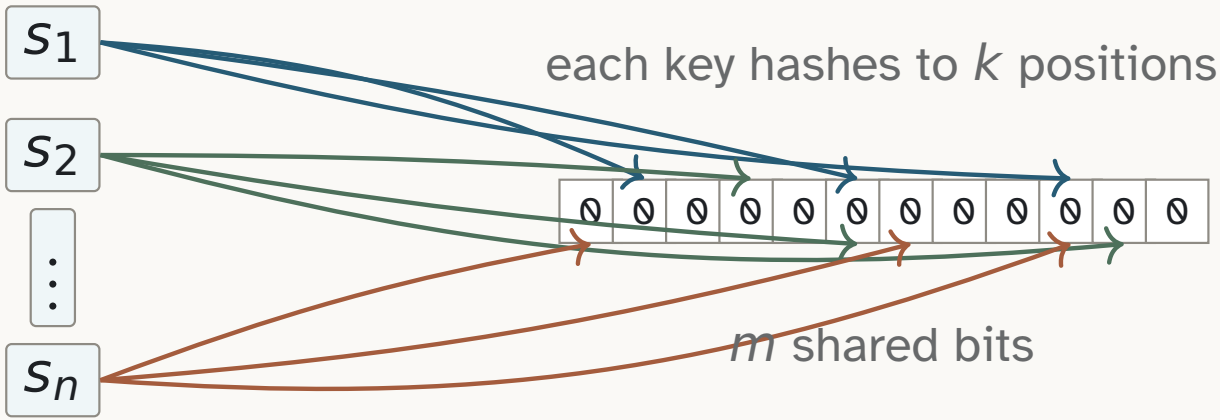


A Bloom filter has an array $B[1:m]$ and hash functions

$$h_1, \ldots, h_k : U \to [m].$$

Inserting x sets $B[h_1(x)], \ldots, B[h_k(x)]$ to 1; querying x checks those same positions.

Bloom filter: all keys share one bit array



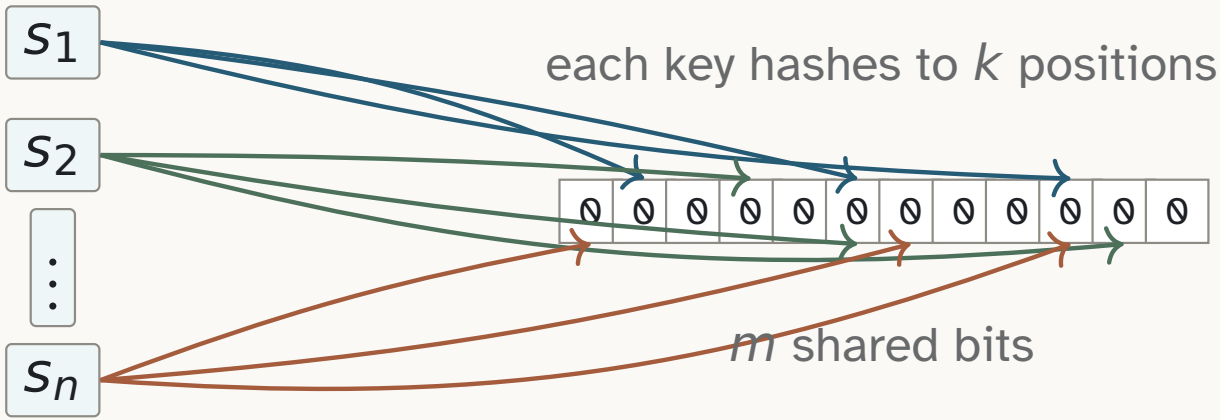
A Bloom filter has an array $B[1:m]$ and hash functions

$$h_1, \ldots, h_k : U \to [m].$$

Inserting x sets $B[h_1(x)], \ldots, B[h_k(x)]$ to 1; querying x checks those same positions.

This sharing is the structural change. The fingerprint scheme stored a separate tag for every key and paid a $\log n$ collision penalty. Here all n keys contribute to one common occupancy pattern.

Bloom filter: all keys share one bit array



A Bloom filter has an array $B[1:m]$ and hash functions

$$h_1, \ldots, h_k : U \to [m].$$

Inserting x sets $B[h_1(x)], \ldots, B[h_k(x)]$ to 1; querying x checks those same positions.

This sharing is the structural change. The fingerprint scheme stored a separate tag for every key and paid a $\log n$ collision penalty. Here all n keys contribute to one common occupancy pattern.

Model used for the probability calculation

The values $h_i(x)$ for distinct pairs (i, x) are independent and uniform on $[m]$.

Insert: turn on k bits

Insertion rule

```
insert(x):
  for i = 1,...,k:
    B[h_i(x)] ← 1
```



start with all zeros

$a \mapsto \{2, 5, 10\}, \quad b \mapsto \{4, 5, 9\}, \quad c \mapsto \{1, 8, 10\}.$

Insert: turn on k bits

Insertion rule

```
insert(x):
  for i = 1,...,k:
    B[h_i(x)] ← 1
```



insert a

$a \mapsto \{2, 5, 10\}, \quad b \mapsto \{4, 5, 9\}, \quad c \mapsto \{1, 8, 10\}.$

Insert: turn on k bits

Insertion rule

```
insert(x):
  for i = 1,...,k:
    B[h_i(x)] ← 1
```



insert b: bit 5 was already 1

$a \mapsto \{2, 5, 10\}, \quad b \mapsto \{4, 5, 9\}, \quad c \mapsto \{1, 8, 10\}.$

Insert: turn on k bits

Insertion rule

```
insert(x):
  for i = 1,...,k:
    B[h_i(x)] ← 1
```



insert c: bit 10 was already 1

$a \mapsto \{2, 5, 10\}, \quad b \mapsto \{4, 5, 9\}, \quad c \mapsto \{1, 8, 10\}.$
Several keys can set the same bit. A 1 records only that at least one key touched that position, not which key.

$$T_{\text{insert}} = O(k).$$

Query: one zero bit rejects

Query rule

```
query(x):
  for i = 1,...,k:
    if B[h_i(x)] = 0:  return no
  return maybe
```



a: all three positions are 1

$a \mapsto \{2, 5, 10\}, \quad x \mapsto \{1, 6, 9\}, \quad y \mapsto \{1, 4, 8\}.$

Query: one zero bit rejects

Query rule

```
query(x):
  for i = 1,...,k:
    if B[h_i(x)] = 0:  return no
  return maybe
```



x: position 6 is 0 $\Rightarrow$ no

$a \mapsto \{2, 5, 10\}, \quad x \mapsto \{1, 6, 9\}, \quad y \mapsto \{1, 4, 8\}.$

Query: one zero bit rejects

Query rule

```
query(x):
  for i = 1,...,k:
    if B[h_i(x)] = 0:   return no
  return maybe
```



y: all three are 1 ⇒ `maybe`

$a \mapsto \{2, 5, 10\}$, $x \mapsto \{1, 6, 9\}$, $y \mapsto \{1, 4, 8\}$.
The third query is a false positive: its three 1 bits can have been supplied by different stored keys. `maybe` means only “not ruled out.”

Stored keys are never rejected

If $x \in S$, then inserting x set every one of its bits:

$$B[h_i(x)] = 1, \qquad i = 1, \ldots, k.$$

Later insertions only change bits from 0 to 1; nothing ever clears a bit. Hence every later query of x sees k ones.

Stored keys are never rejected

If $x \in S$, then inserting x set every one of its bits:

$$B[h_i(x)] = 1, \qquad i = 1, \ldots, k.$$

Later insertions only change bits from 0 to 1; nothing ever clears a bit. Hence every later query of x sees k ones.

No false negatives

For a Bloom filter with insertions only, every stored key is answered `maybe`, with probability 1.

Stored keys are never rejected

If $x \in S$, then inserting x set every one of its bits:

$$B[h_i(x)] = 1, \qquad i = 1, \ldots, k.$$

Later insertions only change bits from 0 to 1; nothing ever clears a bit. Hence every later query of x sees k ones.

No false negatives

For a Bloom filter with insertions only, every stored key is answered `maybe`, with probability 1.

The important change from fingerprints is not merely that we used different hashes. We stopped storing n separate objects. All n keys share one occupancy pattern, so a query will no longer union-bound over n separate stored tags.

Clearing bits breaks deletion

Insert keys a and b. They share bit 5.



If we delete a, can we simply clear positions 2, 5, 8?

Clearing bits breaks deletion

Clearing a's positions also clears bit 5, which b still needs.



querying b now finds a zero

The bit stored only "someone touched me," not how many keys touched it.

Clearing bits breaks deletion

Counting Bloom filter

Replace each bit by a small counter. Insertion increments the k counters; deletion decrements them; a query rejects if any counter is zero.

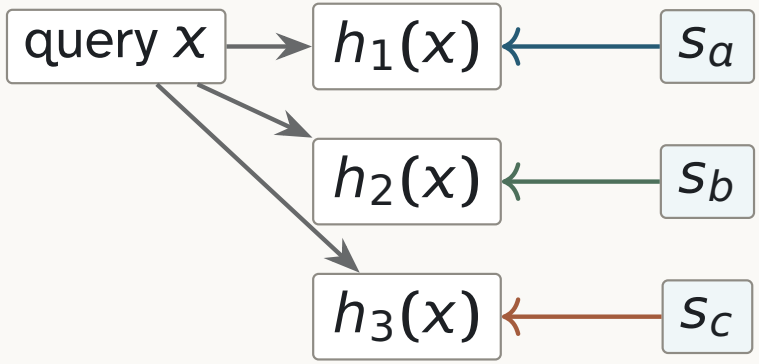
This supports deletion, but uses more memory and introduces counter-overflow considerations.

A false positive can combine bits from different k

For fingerprints, a false positive had a single witness: some stored key s with $f(s) = f(x)$.

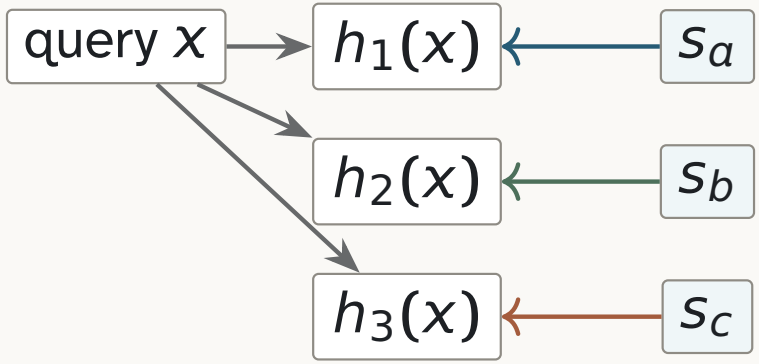
A false positive can combine bits from different k

For fingerprints, a false positive had a single witness: some stored key s with $f(s) = f(x)$.
For a Bloom filter, a nonmember x can pass because its k queried positions were set by several different stored keys.



A false positive can combine bits from different k

For fingerprints, a false positive had a single witness: some stored key s with $f(s) = f(x)$.
For a Bloom filter, a nonmember x can pass because its k queried positions were set by several different stored keys.



So the right summary statistic is not “how many stored keys could collide with x?” It is **how full the shared array is**. That leads to an occupancy problem.

Increasing k makes queries stricter but fills the t

Suppose $k = 1$. After n insertions into m bits, a random query position is occupied with probability roughly n/m when the table is sparse. To get 1% false positives we would need about

$$m \approx 100n,$$

roughly 100 bits per key.

Increasing k makes queries stricter but fills the t

With several probes, a nonmember is rejected as soon as *one* queried position is zero.



three probes: the zero at position 5 rejects

Increasing k makes queries stricter but fills the t

If a fraction ϕ of the array is occupied, then k fresh probes suggest a false-positive probability around

$$\phi^k.$$

So increasing k makes a query harder to pass.

Increasing k makes queries stricter but fills the t

But every stored key also makes k placements. Increasing k therefore increases the occupied fraction ϕ.

We need to know how full the array gets after inserting the n stored keys.

PART 3

The false-positive probability

3

After n insertions, what is the chance one bit is o

Each of the n stored keys makes k placements, so the construction throws

$$kn \text{ balls independently into } m \text{ bins.}$$

After n insertions, what is the chance one bit is o

Each of the n stored keys makes k placements, so the construction throws

$$kn \text{ balls independently into } m \text{ bins.}$$

Fix one bit j. Define

$$q_m := \mathbb{P}\{B[j] = 1\},$$

the probability that this particular bit is occupied after all n stored keys are inserted.

After n insertions, what is the chance one bit is o

Each of the n stored keys makes k placements, so the construction throws

$$kn \text{ balls independently into } m \text{ bins.}$$

Fix one bit j. Define

$$q_m := \mathbb{P}\{B[j] = 1\},$$

the probability that this particular bit is occupied after all n stored keys are inserted.
One placement misses j with probability $1 - 1/m$, so all kn placements miss it with probability

$$\mathbb{P}\{B[j] = 0\} = \left(1 - \frac{1}{m}\right)^{kn}.$$

After n insertions, what is the chance one bit is o

Each of the n stored keys makes k placements, so the construction throws

$$kn \text{ balls independently into } m \text{ bins.}$$

Fix one bit j. Define

$$q_m := \mathbb{P}\{B[j] = 1\},$$

the probability that this particular bit is occupied after all n stored keys are inserted.
One placement misses j with probability $1 - 1/m$, so all kn placements miss it with probability

$$\mathbb{P}\{B[j] = 0\} = \left(1 - \frac{1}{m}\right)^{kn}.$$

Therefore

$$\boxed{q_m = 1 - \left(1 - \frac{1}{m}\right)^{kn}} \approx 1 - e^{-kn/m}.$$

Can we just raise the occupancy probability to the

A tempting calculation is

$$p_{\text{fp}} \stackrel{?}{=} q_m^k,$$

because a nonmember passes only if all k queried positions are occupied.

But after averaging over the random filter, are those k events independent?

Can we just raise the occupancy probability to the

Take the smallest nontrivial example: $m = 2$, one stored key, and $k = 2$. The two stored placements are

$$(1, 1), (1, 2), (2, 1), (2, 2)$$

with equal probability. The occupied fraction Z/m is therefore $1/2$ with probability $1/2$ and 1 with probability $1/2$.

Can we just raise the occupancy probability to the

For a fresh nonmember,

$$p_{\mathrm{fp}} = \mathbb{E}\left[(Z/m)^2\right] = \frac{1}{2}\left(\frac{1}{2}\right)^2 + \frac{1}{2}(1)^2 = \frac{5}{8}.$$

But

$$q_m = \mathbb{E}[Z/m] = \frac{3}{4}, \qquad q_m^2 = \frac{9}{16} \neq \frac{5}{8}.$$

The random occupancy of the whole array couples the query bits.

Conditioning on the array gives the exact false-po

Let Z be the number of occupied bits after the n stored keys are inserted.

Conditioning on the array gives the exact false-po

Let Z be the number of occupied bits after the n stored keys are inserted. Condition on the completed array. If exactly $Z = z$ of the m bits are 1, then each fresh query hash lands on an occupied position with probability z/m, independently across the k query hashes.

Conditioning on the array gives the exact false-po

Let Z be the number of occupied bits after the n stored keys are inserted. Condition on the completed array. If exactly $Z = z$ of the m bits are 1, then each fresh query hash lands on an occupied position with probability z/m, independently across the k query hashes.
Hence

$$\mathbb{P}\{FP(x) \mid Z = z\} = \left(\frac{z}{m}\right)^k.$$

Averaging over the random filter gives the exact identity

$$\boxed{p_{\mathrm{fp}} = \mathbb{E}\left[\left(\frac{Z}{m}\right)^k\right].}$$

Conditioning on the array gives the exact false-po

Let Z be the number of occupied bits after the n stored keys are inserted. Condition on the completed array. If exactly $Z = z$ of the m bits are 1, then each fresh query hash lands on an occupied position with probability z/m, independently across the k query hashes.
Hence

$$\mathbb{P}\{FP(x) \mid Z = z\} = \left(\frac{z}{m}\right)^k.$$

Averaging over the random filter gives the exact identity

$$\boxed{p_{\mathrm{fp}} = \mathbb{E}\left[\left(\frac{Z}{m}\right)^k\right].}$$

Also $\mathbb{E}[Z/m] = q_m$. So the question is now precise: **when may we replace the random quantity Z/m by its mean q_m?**

Can we replace the random occupied fraction by it

If Z/m were nearly deterministic at q_m, then

$$p_{\mathrm{fp}} = \mathbb{E}[(Z/m)^k] \approx q_m^k \approx \left(1 - e^{-kn/m}\right)^k .$$

This is the formula we would like to justify.

Can we replace the random occupied fraction by it

If Z/m were nearly deterministic at q_m, then

$$p_{\mathrm{fp}} = \mathbb{E}[(Z/m)^k] \approx q_m^k \approx \left(1 - e^{-kn/m}\right)^k .$$

This is the formula we would like to justify.
Convexity already gives one side:

$$\mathbb{E}[(Z/m)^k] \geq \left(\mathbb{E}[Z/m]\right)^k = q_m^k.$$

For an upper bound of the same order, we must rule out the event that Z/m is noticeably larger than q_m.

Can we replace the random occupied fraction by it

If Z/m were nearly deterministic at q_m, then

$$p_{\mathrm{fp}} = \mathbb{E}[(Z/m)^k] \approx q_m^k \approx \left(1 - e^{-kn/m}\right)^k.$$

This is the formula we would like to justify.
Convexity already gives one side:

$$\mathbb{E}[(Z/m)^k] \geq \left(\mathbb{E}[Z/m]\right)^k = q_m^k.$$

For an upper bound of the same order, we must rule out the event that Z/m is noticeably larger than q_m.
Why do we need a *relative* concentration bound? If

$$\frac{Z}{m} \leq (1+t)q_m,$$

then

$$\left(\frac{Z}{m}\right)^k \leq q_m^k(1+t)^k \leq q_m^k e^{kt}.$$

Thus controlling the occupied fraction to relative error $t \approx \varepsilon/k$ changes the target false-positive probability by at most a factor e^{ε}.

Poissonization makes the occupancy indicators in

To prove sharp concentration for Z, we would like to apply a Chernoff bound. The difficulty is specific: with exactly kn placements, the indicators

$$\mathbf{1}\{B[1] = 1\}, \ldots, \mathbf{1}\{B[m] = 1\}$$

are not independent.

Poissonization makes the occupancy indicators in

To prove sharp concentration for Z, we would like to apply a Chernoff bound. The difficulty is specific: with exactly kn placements, the indicators

$$\mathbf{1}\{B[1] = 1\}, \ldots, \mathbf{1}\{B[m] = 1\}$$

are not independent.
Instead let the total number of placements be

$$R \sim \mathrm{Poi}(\lambda).$$

By Poisson splitting, the number N_j of placements landing in bit j are independent and

$$N_j \sim \mathrm{Poi}(\lambda/m).$$

Poissonization makes the occupancy indicators in

To prove sharp concentration for Z, we would like to apply a Chernoff bound. The difficulty is specific: with exactly kn placements, the indicators

$$\mathbf{1}\{B[1] = 1\}, \ldots, \mathbf{1}\{B[m] = 1\}$$

are not independent.
Instead let the total number of placements be

$$R \sim \mathrm{Poi}(\lambda).$$

By Poisson splitting, the number N_j of placements landing in bit j are independent and

$$N_j \sim \mathrm{Poi}(\lambda/m).$$

Using the Poisson pmf,

$$\mathbb{P}\{N_j = 0\} = e^{-\lambda/m} \frac{(\lambda/m)^0}{0!} = e^{-\lambda/m}.$$

Therefore the occupied indicators are independent Bernoulli variables with

$$\tilde{q} = 1 - e^{-\lambda/m}, \qquad \tilde{Z} \sim \mathrm{Bin}(m, \tilde{q}).$$

Now an ordinary Chernoff bound applies to $\tilde{Z}$.

Choose a slightly larger Poisson process so it con

The real filter uses exactly kn placements. We deliberately choose

$$\lambda = (1 + \eta)kn$$

for the Poissonized process, with $\eta > 0$ small.

Choose a slightly larger Poisson process so it con

The real filter uses exactly kn placements. We deliberately choose

$$\lambda = (1 + \eta)kn$$

for the Poissonized process, with $\eta > 0$ small.
Why make the mean larger? Because then

$$R \sim \mathrm{Poi}((1 + \eta)kn)$$

is very unlikely to fall below kn.

Choose a slightly larger Poisson process so it con

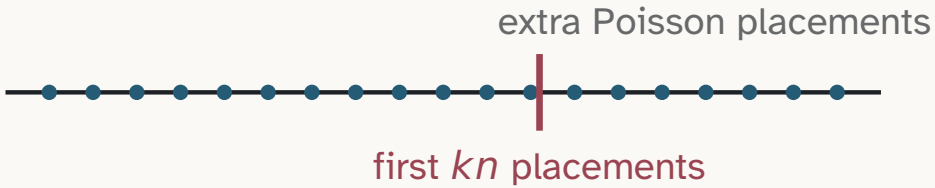
The real filter uses exactly kn placements. We deliberately choose

$$\lambda = (1 + \eta)kn$$

for the Poissonized process, with $\eta > 0$ small.
Why make the mean larger? Because then

$$R \sim \mathrm{Poi}((1 + \eta)kn)$$

is very unlikely to fall below kn.
Couple both processes using one infinite sequence of independent uniform bit locations. The real filter uses the first kn locations; the Poissonized filter uses the first R.



Choose a slightly larger Poisson process so it con

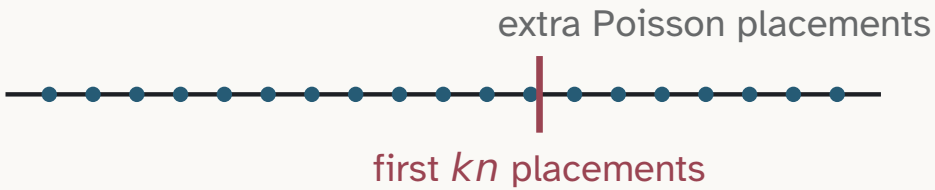
The real filter uses exactly kn placements. We deliberately choose

$$\lambda = (1 + \eta)kn$$

for the Poissonized process, with $\eta > 0$ small.
Why make the mean larger? Because then

$$R \sim \mathrm{Poi}((1 + \eta)kn)$$

is very unlikely to fall below kn.
Couple both processes using one infinite sequence of independent uniform bit locations. The real filter uses the first kn locations; the Poissonized filter uses the first R.



On the event $R \geq kn$, adding the extra placements can only increase the number of occupied bits. Thus an upper-tail bound for $\tilde{Z}$, plus the small probability $\mathbb{P}\{R < kn\}$, yields the needed upper-tail bound for Z.

How m, n, and k determine false positives

We have the exact identity

$$p_{\mathrm{fp}} = \mathbb{E}\left[\left(\frac{Z}{m}\right)^k\right],$$

and the concentration argument shows that for fixed load n/m and fixed k,

$$p_{\mathrm{fp}} = (1 + o(1))\left(1 - e^{-kn/m}\right)^k.$$

A finite bound with explicit exponentially small correction terms is in Appendix A.3.

How m, n, and k determine false positives

We have the exact identity

$$p_{\mathrm{fp}} = \mathbb{E}\left[\left(\frac{Z}{m}\right)^k\right],$$

and the concentration argument shows that for fixed load n/m and fixed k,

$$p_{\mathrm{fp}} = (1 + o(1))\left(1 - e^{-kn/m}\right)^k.$$

A finite bound with explicit exponentially small correction terms is in Appendix A.3.

For example, if $m = 10^8$, $n = 10^7$, and $k = 7$, the target approximation is about 8.2×10^{-3}, while the concentration correction from Appendix A.3 is astronomically smaller than 10^{-6} for even a modest relative tolerance. At this scale it does not affect the engineering choice of m or k.

How m, n, and k determine false positives

We have the exact identity

$$p_{\mathrm{fp}} = \mathbb{E}\left[\left(\frac{Z}{m}\right)^k\right],$$

and the concentration argument shows that for fixed load n/m and fixed k,

$$p_{\mathrm{fp}} = (1 + o(1))\left(1 - e^{-kn/m}\right)^k.$$

A finite bound with explicit exponentially small correction terms is in Appendix A.3.

For example, if $m = 10^8$, $n = 10^7$, and $k = 7$, the target approximation is about 8.2×10^{-3}, while the concentration correction from Appendix A.3 is astronomically smaller than 10^{-6} for even a modest relative tolerance. At this scale it does not affect the engineering choice of m or k.

So for design we use

$$\boxed{\rho(k) := \left(1 - e^{-kn/m}\right)^k.}$$

With m and n fixed, how should we choose k?

PART 4

Choosing k and m

4

Increasing k helps queries but also fills the array

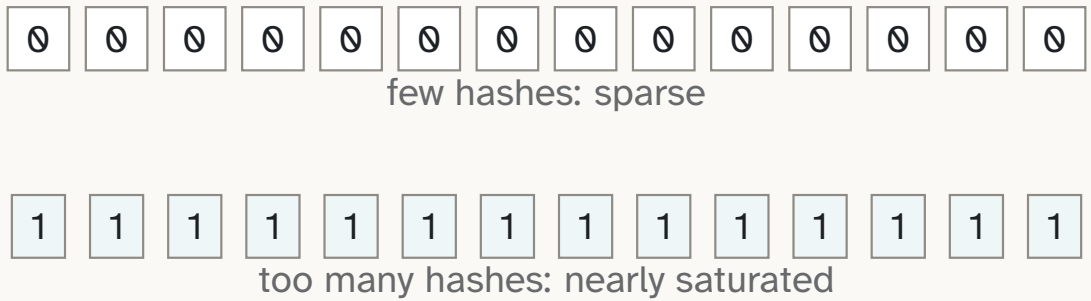
For fixed m and n,

$$\rho(k) = \left(1 - e^{-kn/m}\right)^k.$$

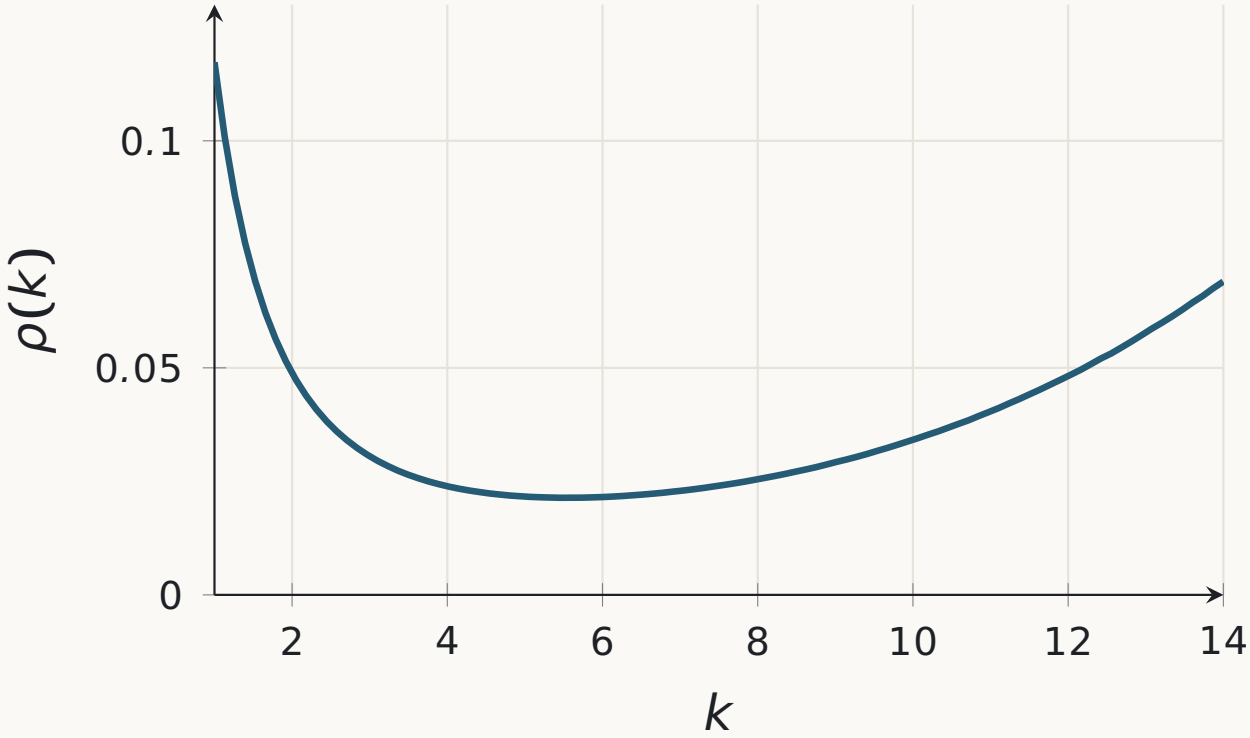
Few hashes leave many zeros, but give a nonmember few chances to encounter one.

Increasing k helps queries but also fills the array

Many hashes give a query more tests, but each stored key also sets more bits, so the array approaches all ones.



Increasing k helps queries but also fills the array



For $m/n = 8$ bits per key, the curve has an interior minimum. What property does the array have there?

Optimizing the number of hash functions

We minimize

$$\rho(k) = \left(1 - e^{-kn/m}\right)^k.$$

Optimizing the number of hash functions

We minimize

$$\rho(k) = \left(1 - e^{-kn/m}\right)^k.$$

Taking logs preserves the minimizer:

$$\ell(k) = \log \rho(k) = k \log\left(1 - e^{-kn/m}\right).$$

Optimizing the number of hash functions

We minimize

$$\rho(k) = \left(1 - e^{-kn/m}\right)^k.$$

Taking logs preserves the minimizer:

$$\ell(k) = \log \rho(k) = k \log\left(1 - e^{-kn/m}\right).$$

Let $a = n/m$. Differentiating,

$$\ell'(k) = \log(1 - e^{-ak}) + \frac{ake^{-ak}}{1 - e^{-ak}}.$$

At a critical point, set $x = e^{-ak}$; then $ak = -\log x$, and

$$0 = \log(1-x) - \frac{x \log x}{1-x} \iff (1-x)\log(1-x) = x \log x.$$

Optimizing the number of hash functions

We minimize

$$\rho(k) = \left(1 - e^{-kn/m}\right)^k.$$

Taking logs preserves the minimizer:

$$\ell(k) = \log \rho(k) = k \log\left(1 - e^{-kn/m}\right).$$

Let $a = n/m$. Differentiating,

$$\ell'(k) = \log(1 - e^{-ak}) + \frac{ake^{-ak}}{1 - e^{-ak}}.$$

At a critical point, set $x = e^{-ak}$; then $ak = -\log x$, and

$$0 = \log(1-x) - \frac{x \log x}{1-x} \iff (1-x)\log(1-x) = x \log x.$$

The equation is solved by $x = 1/2$; Appendix A.5 verifies this is the unique minimizer. Therefore

$$e^{-k^\star n/m} = \frac{1}{2}, \qquad \boxed{k^\star = \frac{m}{n} \ln 2.}$$

Optimizing the number of hash functions

We minimize

$$\rho(k) = \left(1 - e^{-kn/m}\right)^k.$$

Taking logs preserves the minimizer:

$$\ell(k) = \log \rho(k) = k \log\left(1 - e^{-kn/m}\right).$$

Let $a = n/m$. Differentiating,

$$\ell'(k) = \log(1 - e^{-ak}) + \frac{ake^{-ak}}{1 - e^{-ak}}.$$

At a critical point, set $x = e^{-ak}$; then $ak = -\log x$, and

$$0 = \log(1-x) - \frac{x \log x}{1-x} \iff (1-x)\log(1-x) = x \log x.$$

The equation is solved by $x = 1/2$; Appendix A.5 verifies this is the unique minimizer. Therefore

$$e^{-k^\star n/m} = \frac{1}{2}, \qquad \boxed{k^\star = \frac{m}{n} \ln 2.}$$

So at the optimum, about half the bits are zero and half are one.

How many bits per key give error δ?

At the optimum,

$$e^{-k^\star n/m} = \frac{1}{2}.$$

How many bits per key give error δ?

At the optimum,

$$e^{-k^\star n/m} = \frac{1}{2}.$$

Hence

$$p_{\mathrm{fp}} \approx \rho(k^\star) = \left(\frac{1}{2}\right)^{k^\star} = 2^{-k^\star}.$$

How many bits per key give error δ?

At the optimum,

$$e^{-k^\star n/m} = \frac{1}{2}.$$

Hence

$$p_{\text{fp}} \approx \rho(k^\star) = \left(\frac{1}{2}\right)^{k^\star} = 2^{-k^\star}.$$

To require $p_{\text{fp}} \le \delta$, it is enough that

$$2^{-k^\star} \le \delta \qquad \Longrightarrow \qquad k^\star \ge \log_2 \frac{1}{\delta}.$$

How many bits per key give error δ?

At the optimum,

$$e^{-k^\star n/m} = \frac{1}{2}.$$

Hence

$$p_{\text{fp}} \approx \rho(k^\star) = \left(\frac{1}{2}\right)^{k^\star} = 2^{-k^\star}.$$

To require $p_{\text{fp}} \le \delta$, it is enough that

$$2^{-k^\star} \le \delta \qquad \Longrightarrow \qquad k^\star \ge \log_2 \frac{1}{\delta}.$$

Combining this with $k^\star = (m/n)\ln 2$ gives

$$\boxed{\frac{m}{n} \ge \frac{1}{\ln 2} \log_2 \frac{1}{\delta}} \approx 1.44 \log_2 \frac{1}{\delta}.$$

Thus a tuned Bloom filter uses about $\log_2(1/\delta)$ hashes and $1.44 \log_2(1/\delta)$ bits per stored key.

Sizing the username filter

Take $n = 10^7$ usernames and $\delta = 0.01$:

$$\frac{m}{n} \geq \frac{\ln 100}{(\ln 2)^2} \approx 9.6 \text{ bits per key}, \qquad m \approx 9.6 \times 10^7 \text{ bits} \approx 12 \text{ MB},$$

and $k^\star \approx \log_2 100 \approx 6.6$, so use $k = 7$.

Summary of S	bits per key	total
exact, 16-byte avg.	≈ 128	160 MB
random tags	30	37.5 MB
Bloom filter, $k = 7$	9.6	12 MB

Compare with your guess from the start of the lecture. Is 9.6 bits per key close to the best possible, or could a cleverer structure use much less?

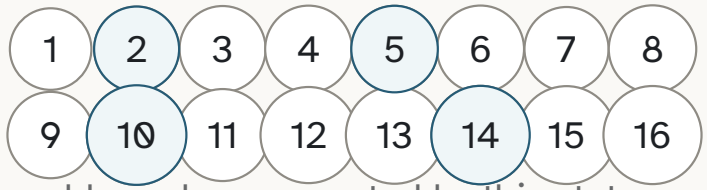
PART 5

Can any structure use less space?

5

How many sets can one memory state represent?

Take a universe of $N = 16$ keys and store sets of $n = 2$ keys. Consider one memory state that answers `maybe` on exactly four universe elements.



blue = keys accepted by this state

How many sets can one memory state represent?

Because false negatives are forbidden, any stored set represented by this state must be a 2-element subset of those four accepted keys.

$$\text{sets represented by this state} \leq \binom{4}{2} = 6.$$

How many sets can one memory state represent?

But the universe contains

$$\binom{16}{2} = 120$$

possible stored sets. Therefore we need at least

$$120/6 = 20$$

different memory states.

How many sets can one memory state represent?

With b bits there are at most 2^b states, so

$$2^b \geq 20 \qquad \Longrightarrow \qquad b \geq 5.$$

The general lower bound repeats exactly these three counts: accepted keys per state, sets per state, and total possible sets.

The same counting argument for general N and n

For the main counting idea, first assume a stronger deterministic condition: every reachable state accepts at most a δ fraction of the nonmembers. Appendix A.4 converts the original randomized per-query guarantee to this form up to an $O(n)$ loss.

The same counting argument for general N and n

For a state σ, define its accepted set

$$A_\sigma = \{x \in U : \mathrm{query}_\sigma(x) = \texttt{maybe}\}.$$

If σ represents S, then no false negatives imply

$$S \subseteq A_\sigma.$$

The same counting argument for general N and n

The deterministic false-positive condition gives

$$|A_\sigma| \le a := n + \delta(N - n).$$

So one memory state can represent at most

$$\binom{a}{n}$$

different n-element sets.

The same counting argument for general N and n

There are $\binom{N}{n}$ possible stored sets and at most 2^b memory states. Therefore

$$\boxed{2^b \binom{a}{n} \geq \binom{N}{n}.}$$

This is the general version of $20 \times 6 \geq 120$ from the example.

Turning the count into a space lower bound

From the counting inequality,

$$2^b \geq \frac{\binom{N}{n}}{\binom{a}{n}}.$$

Turning the count into a space lower bound

Expand the ratio:

$$\frac{\binom{N}{n}}{\binom{a}{n}} = \prod_{j=0}^{n-1} \frac{N-j}{a-j}.$$

For every $j < n$,

$$\frac{N-j}{a-j} \geq \frac{N}{a},$$

because $a(N-j) \geq N(a-j)$ is equivalent to $a \leq N$.

Turning the count into a space lower bound

Hence

$$2^b \geq \left(\frac{N}{a}\right)^n.$$

If the universe is large enough that $\delta(N-n) \geq n$, then

$$a = n + \delta(N-n) \leq 2\delta(N-n) \leq 2\delta N,$$

so $N/a \geq 1/(2\delta)$.

Turning the count into a space lower bound

Therefore

$$2^b \geq \left(\frac{1}{2\delta}\right)^n \qquad \Longrightarrow \qquad \boxed{b \geq n \log_2 \frac{1}{\delta} - n.}$$

Notice that $N = |U|$ disappears from the leading term. That is possible because our guarantee is for each fixed nonmember, not simultaneous correctness over all $N - n$ nonmembers.

How close is the Bloom filter to optimal?

	bits per key	username example
lower bound	$\log_2(1/\delta) - 1$	≥ 7.0 MB
Bloom, tuned	$1.44 \log_2(1/\delta)$	12 MB
random tags	$\log_2 n + \log_2(1/\delta)$	37.5 MB

As $\delta \to 0$, the lower bound is $n \log_2(1/\delta) - O(n)$ bits, while a tuned Bloom filter uses $1.44\, n \log_2(1/\delta)$ bits. So Bloom filters have the right scale, with a leading constant $1/\ln 2 \approx 1.44$ times the information-theoretic optimum.

That factor is not forced. Storing the random tags compactly, so that a tag's position in the table encodes its first $\log_2 n$ bits, uses at most $n \log_2(1/\delta) + 3n$ bits (Appendix A.6). Quotient filters and cuckoo filters are practical versions of this idea; a cuckoo filter stores tags in the cuckoo hash table from Lecture 8.

PART 6

Implementation notes and uses

6

Do we really need k independent hash functions?

The analysis assumed that all kn stored positions are independent uniform samples from $[m]$. That is clean, but evaluating k unrelated hash functions per query is expensive.

A standard practical trick is **double hashing**. Compute only two hash values $g_1(x)$ and $g_2(x)$, and use

$$h_i(x) = g_1(x) + i\,g_2(x) \pmod m, \qquad i = 1, \ldots, k.$$

Kirsch–Mitzenmacher: this has the same limiting false-positive probability as k independent hashes.

The tag scheme and the Bloom filter therefore use different randomness assumptions in the proof:

- tags only needed a pairwise collision bound;
- the clean Bloom analysis used full independence, but practice can use less randomness without changing the limiting rate.

Where Bloom filters are used

- **Databases on disk.** Storage engines such as RocksDB and Cassandra keep data in many sorted files. Each file has a small Bloom filter in memory, and a lookup skips any file whose filter answers `no`. A false positive costs one wasted disk read.
- **Web caches.** Many pages are requested only once. A cache can record requests in a Bloom filter and store a page only on its second request, so pages seen once never take cache space.
- **Networks.** A machine can send a Bloom filter summarizing objects it already has. Its peer avoids sending objects the filter answers `maybe` for; false positives may skip a useful transfer, so exact synchronization needs a later repair step.

In each case the cost structure matches the username example: `no` must be reliable, and an occasional unnecessary `maybe` costs a little extra work.

Summary

1. Approximate membership saves space because only one kind of error is allowed: a free key may produce `maybe`, but a stored key is never rejected.
2. Plain random tags work, but they pay $\log_2 n$ extra bits per key because a query can collide with any one of n stored tags.
3. A Bloom filter pools the evidence in one bit array. False positives are governed by the occupied fraction of that array.
4. Conditioning on the array gives the exact identity

$$p_{\mathrm{fp}} = \mathbb{E}\left[\left(\frac{Z}{m}\right)^k\right],$$

   and occupancy concentration justifies the design approximation

$$p_{\mathrm{fp}} \approx (1 - e^{-kn/m})^k.$$

5. The optimal load is about half full, giving $k \approx \log_2(1/\delta)$ and $m \approx 1.44\, n \log_2(1/\delta)$ bits.
6. Counting shows that the best possible leading scale is $n \log_2(1/\delta)$ bits.

Questions?

APPENDIX A

Supporting arguments

A

A.1 Comparing q_m with q_∞

Setting. kn independent uniform placements into m bits. Each bit is 1 w.p.

$$q_m = 1 - \Big(1 - \frac{1}{m}\Big)^{kn}, \qquad \text{and we compare with} \qquad q_\infty = 1 - e^{-kn/m}.$$

Upper bound. $1 - \frac{1}{m} \le e^{-1/m}$, so $(1 - \frac{1}{m})^{kn} \le e^{-kn/m}$, that is, $q_m \le q_\infty$.

Lower bound. For $0 \le x \le \frac{1}{2}$,

$$-\ln(1-x) = x + x^2 \sum_{i \ge 2} \frac{x^{i-2}}{i} \le x + x^2\big(\tfrac{1}{2} + \tfrac{1}{3} \cdot \tfrac{x}{1-x}\big) \le x + x^2,$$

so $1 - x \ge e^{-x-x^2}$. With $x = 1/m$ (and $m \ge 2$),

$$\Big(1 - \frac{1}{m}\Big)^{kn} \ge e^{-kn/m}\, e^{-kn/m^2} \ge e^{-kn/m}\Big(1 - \frac{kn}{m^2}\Big),$$

hence $0 \le q_\infty - q_m \le e^{-kn/m}\, kn/m^2 \le kn/m^2$.

For a tuned filter ($kn/m \approx \ln 2$, $q_\infty \approx \frac{1}{2}$), $q_m^k \ge q_\infty^k(1 - 1.4k/m)$.

A.2 A variance bound for the number of ones

Setting. $I_j = \mathbf{1}\{B[j] = 1\}$, $Z = \sum_j I_j$, and $\mathbb{E}I_j = q_m$.

The bits are negatively correlated. For $i \neq j$, both bits stay 0 exactly when all kn placements miss both positions:

$$\mathbb{P}\{I_i = 0, I_j = 0\} = \left(1 - \tfrac{2}{m}\right)^{kn} \leq \left(1 - \tfrac{1}{m}\right)^{2kn} = \mathbb{P}\{I_i = 0\}\mathbb{P}\{I_j = 0\},$$

using $1 - \frac{2}{m} \leq (1 - \frac{1}{m})^2$. Replacing both variables by one minus themselves does not change their covariance, so

$$\begin{aligned}\operatorname{Cov}(I_i, I_j) &= \operatorname{Cov}(1 - I_i, 1 - I_j) \\ &= \mathbb{P}\{I_i = 0, I_j = 0\} - \mathbb{P}\{I_i = 0\}\mathbb{P}\{I_j = 0\} \leq 0.\end{aligned}$$

Variance. Dropping the nonpositive covariance terms,

$$\operatorname{Var}(Z) = \sum_j \operatorname{Var}(I_j) + 2\sum_{i<j} \operatorname{Cov}(I_i, I_j) \leq \sum_j q_m(1 - q_m) \leq \frac{m}{4}.$$

A.2 From variance to an additive error

By the mean value theorem, for $a, b \in [0, 1]$,

$$|a^k - b^k| \le \Big(\max_{t \in [0,1]} k t^{k-1} \Big) |a - b| = k|a - b|.$$

By Cauchy–Schwarz and the variance bound,

$$\mathbb{E}\Big|\frac{Z}{m} - q_m\Big| \le \sqrt{\operatorname{Var}\Big(\frac{Z}{m}\Big)} \le \frac{1}{2\sqrt{m}}.$$

Therefore

$$\begin{aligned} \big|p_{\mathrm{fp}} - q_m^k\big| &= \Big|\mathbb{E}\Big[\Big(\frac{Z}{m}\Big)^k - q_m^k\Big]\Big| \\ &\le k\,\mathbb{E}\Big|\frac{Z}{m} - q_m\Big| \le \frac{k}{2\sqrt{m}}. \end{aligned}$$

This is correct but additive. When p_{fp} is smaller than $k/\sqrt{m}$ it says nothing useful, which is why the main lecture uses Poissonization instead.

A.3 Poissonization bookkeeping

Let $U_1, U_2, \ldots$ be independent uniform placements in $[m]$, and let Z_j be the number of distinct positions among $U_1, \ldots, U_j$. The real filter has $Z = Z_{kn}$.

Let $R \sim \mathrm{Poi}(\lambda)$ be independent of the U_i, with $\lambda = (1+\eta)kn$ and $0 < \eta \le 1$. If $R \ge kn$, then $Z_{kn} \le Z_R$. Therefore

$$\begin{aligned} p_{\mathrm{fp}} &= \mathbb{E}\Big[\big(\tfrac{Z_{kn}}{m}\big)^k \mathbf{1}\{R \ge kn\}\Big] + \mathbb{E}\Big[\big(\tfrac{Z_{kn}}{m}\big)^k \mathbf{1}\{R < kn\}\Big] \\ &\le \mathbb{E}\Big[\big(\tfrac{Z_R}{m}\big)^k\Big] + \mathbb{P}\{R < kn\}. \end{aligned}$$

Since $kn = (1-\theta)\lambda$ with $\theta = \eta/(1+\eta)$, the Poisson Chernoff lower tail gives

$$\mathbb{P}\{R < kn\} \le e^{-\eta^2 kn/4}.$$

A.3 Bounding the Poissonized filter

By Poisson splitting, $Z_R \sim \mathrm{Bin}(m, \tilde{q})$, where

$$\tilde{q} = 1 - e^{-\lambda/m}.$$

For $0 < \varepsilon' \le 1$, Chernoff gives

$$\mathbb{P}\{Z_R \ge (1+\varepsilon')m\tilde{q}\} \le e^{-\varepsilon'^2 m\tilde{q}/3}.$$

On the complement of this event,

$$\left(\frac{Z_R}{m}\right)^k \le ((1+\varepsilon')\tilde{q})^k,$$

and the expression is always at most 1. Hence

$$\mathbb{E}\left[\left(\tfrac{Z_R}{m}\right)^k\right] \le ((1+\varepsilon')\tilde{q})^k + e^{-\varepsilon'^2 m\tilde{q}/3}.$$

Finally compare $\tilde{q}$ with $q_\infty = 1 - e^{-kn/m}$. Concavity of $1 - e^{-x}$ gives

$$q_\infty \le \tilde{q} \le (1+\eta)q_\infty.$$

A.3 A finite relative-error statement

Combining the previous two slides,

$$p_{\mathrm{fp}} \le ((1+\varepsilon')(1+\eta)q_\infty)^k + e^{-\varepsilon'^2 m q_\infty/3} + e^{-\eta^2 k n/4}.$$

Choose $\varepsilon' = \eta = \varepsilon/(2k)$ and use $(1+\varepsilon/(2k))^{2k} \le e^{\varepsilon}$. Then, for $0 < \varepsilon \le 1$,

$$q_m^k \le p_{\mathrm{fp}} \le e^{\varepsilon} q_\infty^k + e^{-\varepsilon^2 m q_\infty/(12k^2)} + e^{-\varepsilon^2 n/(16k)}.$$

For fixed k and fixed load n/m, the two exponential tails vanish as $m, n \to \infty$, and $q_m, q_\infty \to 1 - e^{-kn/m}$. Thus

$$p_{\mathrm{fp}} = (1+o(1))(1-e^{-kn/m})^k.$$

A.4 Lower bound with free hash functions

Setting. The structure draws hash functions H (and any other randomness it uses) independently of S. These are not counted in the b bits. Given H and S it stores a b-bit state, and a query reads H and the state. The only guarantee is per query: for every S and every $x \notin S$, $\mathbb{P}_H\{x \text{ is accepted}\} \le \delta$.

For fixed H and S, let $X_H(S)$ be the fraction of the $N - n$ nonmembers that are accepted.

Step 1: one good choice of hash functions. For each fixed S, $\mathbb{E}_H[X_H(S)]$ is the average over $x \notin S$ of $\mathbb{P}_H\{x \text{ is accepted}\}$, so it is at most δ. Let S be a uniformly random n-set, independent of H. Then $\mathbb{E}_H\mathbb{E}_S[X_H(S)] = \mathbb{E}_S\mathbb{E}_H[X_H(S)] \le \delta$, so some fixed H^* has $\mathbb{E}_S[X_{H^*}(S)] \le \delta$.

Step 2: most sets are good for H^*. By Markov's inequality over S,

$$\mathbb{P}_S\{X_{H^*}(S) > 2\delta\} \le \tfrac{1}{2},$$

so at least $\frac{1}{2}\binom{N}{n}$ sets have $X_{H^*}(S) \le 2\delta$. Call these sets good.

A.4 Free hash functions, continued

Step 3: count. With H^* fixed, each set gets one state. The state of a good set S accepts all of S and at most $a' = n + 2\delta(N-n)$ keys in total. So one state represents at most $\binom{a'}{n}$ good sets, and

$$2^b \binom{a'}{n} \geq \frac{1}{2}\binom{N}{n}.$$

Step 4: solve. As in the main lecture, each factor $(N-j)/(a'-j)$ is at least N/a'. If $2\delta(N-n) \geq n$, then $a' \leq 4\delta N$, so

$$2^b \geq \frac{1}{2}\Big(\frac{1}{4\delta}\Big)^n, \qquad b \geq n \log_2 \frac{1}{\delta} - 2n - 1.$$

Why Step 1 is needed. For each S separately, a typical H works. But a state means something only together with the hash functions used to read it, so the count needs one H that works for many sets at once.

A.5 The optimal k is unique

Let $h(p) = \ln p \cdot \ln(1-p)$ for $p \in (0,1)$. Then

$$h'(p) = \frac{\ln(1-p)}{p} - \frac{\ln p}{1-p} = \frac{D(p)}{p(1-p)},$$

where $D(p) = (1-p)\ln(1-p) - p\ln p$. We have $D(0) = D(\frac{1}{2}) = 0$ (using $0 \ln 0 = 0$) and $D(1-p) = -D(p)$. Differentiating twice,

$$D'(p) = -\ln\big(p(1-p)\big) - 2, \qquad D''(p) = -\frac{1-2p}{p(1-p)},$$

and $D'' < 0$ on $(0, \frac{1}{2})$. So D is concave on $[0, \frac{1}{2}]$ and zero at both ends, hence $D > 0$ on $(0, \frac{1}{2})$. By the symmetry, $D < 0$ on $(\frac{1}{2}, 1)$.

Therefore h increases on $(0, \frac{1}{2}]$ and decreases on $[\frac{1}{2}, 1)$, and $p = \frac{1}{2}$ is its only critical point.

Since $p = e^{-kn/m}$ decreases in k and $\ln \rho = -\frac{m}{n} h(p)$, the function $\rho(k)$ decreases for $k < k^\star$ and increases for $k > k^\star$. That is why the best integer k is $\lfloor k^\star \rfloor$ or $\lceil k^\star \rceil$.

A.6 Storing tags without paying for $\log_2 n$

Assume n is a power of 2 (otherwise round up, at a cost of $O(n)$ bits). Use tags of $b = \log_2 n + r$ bits with $r = \lceil \log_2(1/\delta) \rceil$, so the false-positive bound is $n/2^b = 2^{-r} \le \delta$, as before.

Split each tag into a *quotient* (its first $\log_2 n$ bits, naming one of n buckets) and a *remainder* (its last r bits). Store:

1. for each bucket $q = 1, \ldots, n$ in order, the number c_q of stored tags with quotient q, written in unary as c_q ones followed by a zero: $\sum_q (c_q + 1) = 2n$ bits;
2. the remainders, grouped by bucket in the same order: nr bits.

To query x, compute the quotient and remainder of $f(x)$. The unary counts say where bucket q's remainders start and how many there are; answer `maybe` if the remainder appears there. This tests exactly whether $f(x) \in F$, so the error is the same as for plain tags.

Total: $nr + 2n \le n \log_2(1/\delta) + 3n$ bits. The quotient is never written down; a remainder's position encodes it, which saves $\log_2 n$ bits per key. Queries here scan the counts. Quotient filters and cuckoo filters spend a few more bits per key to make queries fast.

APPENDIX B

Chernoff without Poissonization

B

B.1 What the Chernoff proof needs

Setting. Placements $U_1, \ldots, U_{kn}$ are independent and uniform in $[m]$. Bit j is occupied ($I_j = 1$) if $U_i = j$ for some i. Let $Z = \sum_j I_j$, $q_m = \mathbb{P}\{I_j = 1\}$, and $\mu = \mathbb{E}Z = mq_m$.

This appendix applies Chernoff directly to Z, even though the I_j are dependent.

For a sum $X = \sum_j X_j$ of independent indicators with $\mathbb{P}\{X_j = 1\} = p_j$, the Chernoff upper tail comes from $\mathbb{E}e^{sX} = \prod_j \mathbb{E}e^{sX_j} = \prod_j \big(1 + p_j(e^s - 1)\big) \le e^{\mu(e^s - 1)}$ for $s > 0$, and Markov's inequality applied to e^{sX}. Independence is used only in the first equality, and an inequality would do. So it is enough to show that for every $s > 0$,

$$\mathbb{E}e^{sZ} \le \prod_{j=1}^{m} \mathbb{E}e^{sI_j} = \big(1 + q_m(e^s - 1)\big)^m.$$

The bits compete for a fixed number of placements, so we expect them to be 1 together less often than independent bits.

B.2 Conditioning on an empty bit

For $J \subseteq [m]$, let O_J be the event that every bit in J is occupied.

Lemma 1

If $j \notin J$, then $\mathbb{P}\{O_J \mid I_j = 0\} \geq \mathbb{P}\{O_J\}$.

Proof by re-throwing. Let $W_1, \ldots, W_{kn}$ be independent and uniform on $[m] \setminus \{j\}$, independent of the U_i. Set $V_i = U_i$ if $U_i \neq j$, and $V_i = W_i$ if $U_i = j$.

The V_i are independent, and for $l \neq j$, $\mathbb{P}\{V_i = l\} = \frac{1}{m} + \frac{1}{m} \cdot \frac{1}{m-1} = \frac{1}{m-1}$. Conditioned on $I_j = 0$, the U_i are also independent and uniform on $[m] \setminus \{j\}$. So (V_i) has the conditional law of the placements given $I_j = 0$.

Re-throwing only moves placements out of bit j: any bit $l \neq j$ hit by U_i is hit by V_i. As $j \notin J$, if the U's occupy J, so do the V's. Hence

$$\mathbb{P}\{O_J\} \leq \mathbb{P}\{\text{the } V_i \text{ occupy } J\} = \mathbb{P}\{O_J \mid I_j = 0\}. \qquad \square$$

B.3 Occupied bits are rarely all 1 together

Lemma 2

For every $J \subseteq [m]$, $\mathbb{P}\{O_J\} \le q_m^{|J|}$.

Proof. Induction on $|J|$; the case $J = \varnothing$ is trivial. For $j \notin J$, Lemma 1 gives

$$\begin{aligned}\mathbb{P}\{O_{J\cup\{j\}}\} &= \mathbb{P}\{O_J\} - \mathbb{P}\{I_j = 0\}\,\mathbb{P}\{O_J \mid I_j = 0\} \\ &\le \mathbb{P}\{O_J\} - (1 - q_m)\mathbb{P}\{O_J\} \\ &= q_m \mathbb{P}\{O_J\} \le q_m^{|J|+1}. \qquad \square\end{aligned}$$

Compare: if the bits were independent, $\mathbb{P}\{O_J\}$ would equal $q_m^{|J|}$ exactly.

B.3 The moment generating function bound

Lemma 3

For every $s > 0$, $\mathbb{E}e^{sZ} \le \left(1 + q_m(e^s - 1)\right)^m \le e^{\mu(e^s-1)}$.

Proof. Let $a = e^s - 1 > 0$. Since $I_j \in \{0, 1\}$, $e^{sI_j} = 1 + aI_j$. Expand the product over all subsets J of bits:

$$\prod_{j=1}^{m}(1 + aI_j) = \sum_{J\subseteq[m]} a^{|J|} \prod_{j\in J} I_j = \sum_{J\subseteq[m]} a^{|J|}\, \mathbf{1}_{O_J}.$$

Take expectations and apply Lemma 2 term by term. Every coefficient $a^{|J|}$ is positive, which is where $s > 0$ is used:

$$\mathbb{E}e^{sZ} = \sum_{J\subseteq[m]} a^{|J|}\, \mathbb{P}\{O_J\} \le \sum_{J\subseteq[m]} (aq_m)^{|J|} = (1 + aq_m)^m.$$

Finally $1 + x \le e^x$ gives $(1 + aq_m)^m \le e^{maq_m} = e^{\mu(e^s-1)}$. □

B.4 The Chernoff upper tail for Z

Proposition

For $0 < \varepsilon \le 1$, $\mathbb{P}\{Z \ge (1+\varepsilon)\mu\} \le e^{-\varepsilon^2\mu/3}$.

Proof. For $s > 0$, Markov's inequality applied to e^{sZ} and Lemma 3 give

$$\mathbb{P}\{Z \ge (1+\varepsilon)\mu\} \le \frac{\mathbb{E}e^{sZ}}{e^{s(1+\varepsilon)\mu}} \le \exp\big(\mu(e^s-1) - s(1+\varepsilon)\mu\big).$$

Take $s = \ln(1+\varepsilon)$: the bound becomes $\exp\big(-\mu[(1+\varepsilon)\ln(1+\varepsilon) - \varepsilon]\big)$.

It remains to show $g(\varepsilon) = (1+\varepsilon)\ln(1+\varepsilon) - \varepsilon - \varepsilon^2/3 \ge 0$ on $[0,1]$. We have

$$g(0) = 0, \qquad g'(\varepsilon) = \ln(1+\varepsilon) - \tfrac{2\varepsilon}{3}, \qquad g''(\varepsilon) = \tfrac{1}{1+\varepsilon} - \tfrac{2}{3}.$$

So g' increases on $[0, \frac{1}{2}]$ and decreases on $[\frac{1}{2}, 1]$. Since $g'(0) = 0$ and $g'(1) = \ln 2 - \frac{2}{3} > 0$, $g' \ge 0$ on $[0,1]$, so $g \ge 0$. □

B.5 The resulting bound on p_{fp}

Split on the event $\{Z \le (1 + \frac{\varepsilon}{k})\mu\}$, as in the main lecture, and use the Proposition with ε/k in place of ε:

$$q_m^k \le p_{\mathrm{fp}} \le \left((1 + \tfrac{\varepsilon}{k})q_m\right)^k + e^{-\varepsilon^2 m q_m/(3k^2)} \le e^{\varepsilon} q_m^k + e^{-\varepsilon^2 m q_m/(3k^2)}.$$

Compared with the Poissonization proof, this bound has base q_m instead of q_∞, one tail instead of two, and a better constant in the exponent. The price is the coupling argument in Lemma 1.

Terminology. Lemma 2 says the occupied bits are less likely to all be 1 than independent bits with the same marginals would be. This is one form of **negative dependence**. The occupancy indicators in fact satisfy a stronger property, **negative association** (Dubhashi–Ranjan, 1998): for disjoint sets of bits and nondecreasing functions f, g of those bits, $\mathbb{E}[fg] \le \mathbb{E}f\,\mathbb{E}g$. That property implies Lemma 3, but we did not need it: the Chernoff proof uses only the bound on $\mathbb{E}e^{sZ}$.