

Download this presentation here:

# Intro to probabilistic data structures



# About Me

- Software Engineer
- Interest in Algorithms & DS and AI

# Agenda

**Probabilistic Data Structures (PDS) for this classes of problems:**

- membership
- cardinality
- frequency

**PDS will return approximate values**

# Membership



# Membership use cases

In general when wrong answer do not involve correctness but more work

- Taken username
- Cassandra avoids checking SSTable data files when merging data on disk
- Ad placement: has the user already seen this ad?
- Fraud detection (has the user paid from this location before?)

# Membership

## Checking if an item belongs to a set

- Bloom Filter
- Counting Bloom Filter
- Cuckoo Filter

TYPE I ERROR: FALSE POSITIVE

TYPE II ERROR: FALSE NEGATIVE

TYPE III ERROR: TRUE POSITIVE FOR  
INCORRECT REASONS

TYPE IV ERROR: TRUE NEGATIVE FOR  
INCORRECT REASONS

TYPE V ERROR: INCORRECT RESULT WHICH  
LEADS YOU TO A CORRECT  
CONCLUSION DUE TO  
UNRELATED ERRORS

TYPE VI ERROR: CORRECT RESULT WHICH  
YOU INTERPRET WRONG

TYPE VII ERROR: INCORRECT RESULT WHICH  
PRODUCES A COOL GRAPH

TYPE VIII ERROR: INCORRECT RESULT WHICH  
SPARKS FURTHER RESEARCH  
AND THE DEVELOPMENT OF  
NEW TOOLS WHICH REVEAL  
THE FLAW IN THE ORIGINAL  
RESULT WHILE PRODUCING  
NOVEL CORRECT RESULTS

TYPE IX ERROR: THE RISE OF SKYWALKER

(source: <https://xkcd.com/2303>)

# Bloom Filter

Invented by Burton H. Bloom in 1970

Consists of :

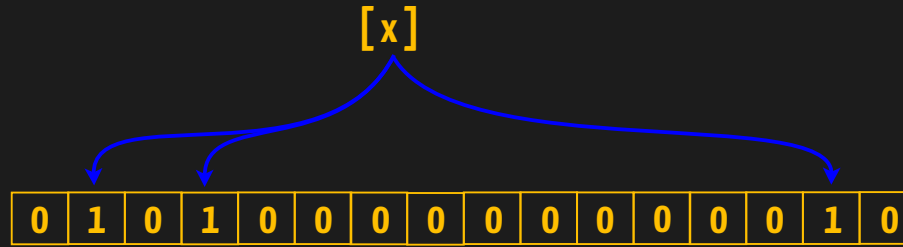
- a bit array of size  $m$
- $k$  different hash functions

Membership condition :

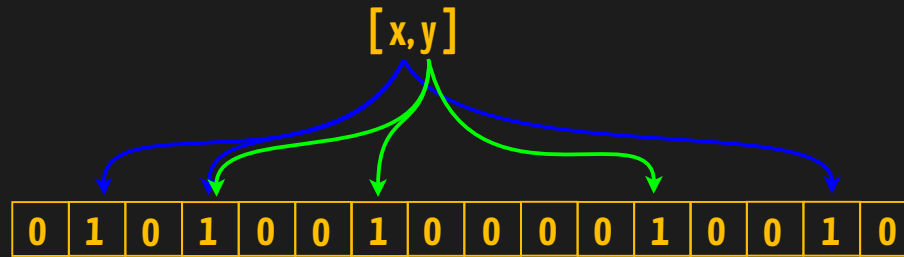
- if all bits are set

# Bloom Filter with $m=16, k=3$

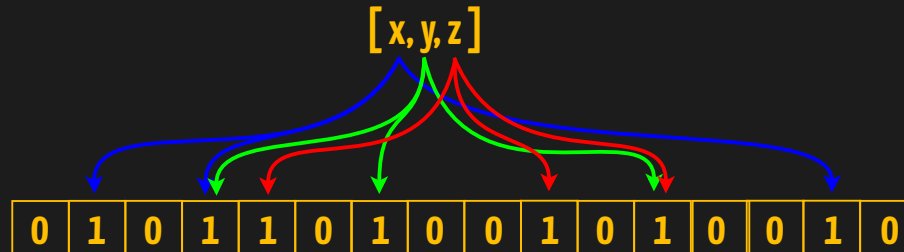
Insert  $x$



Insert  $y$

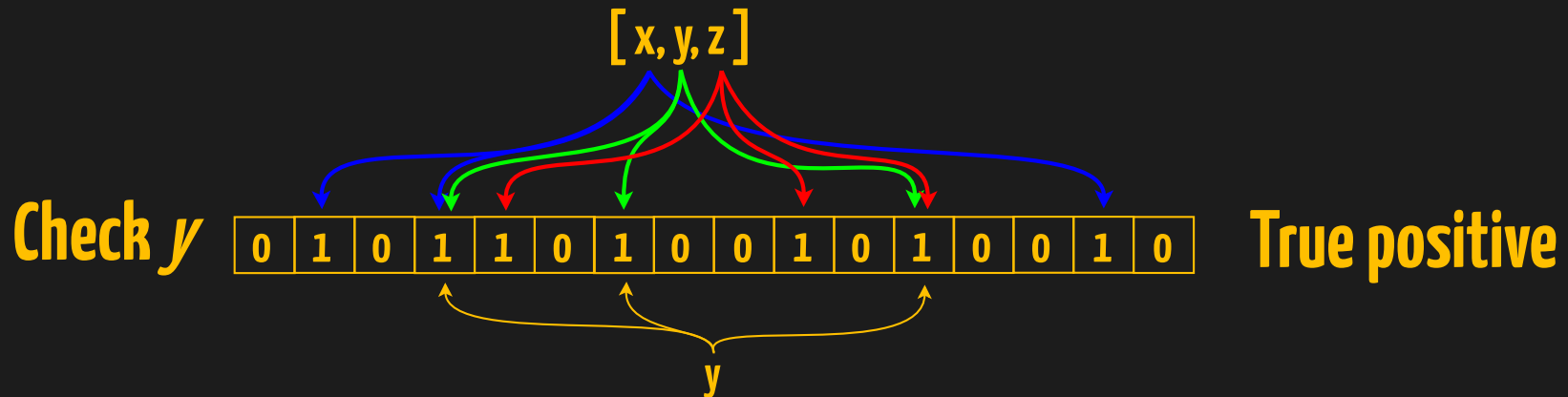
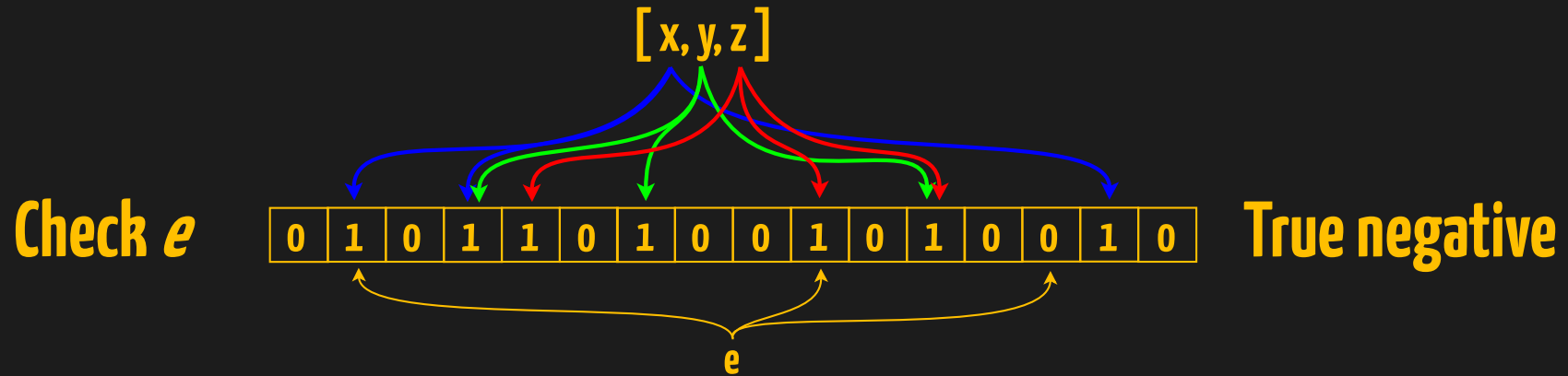


Insert  $z$

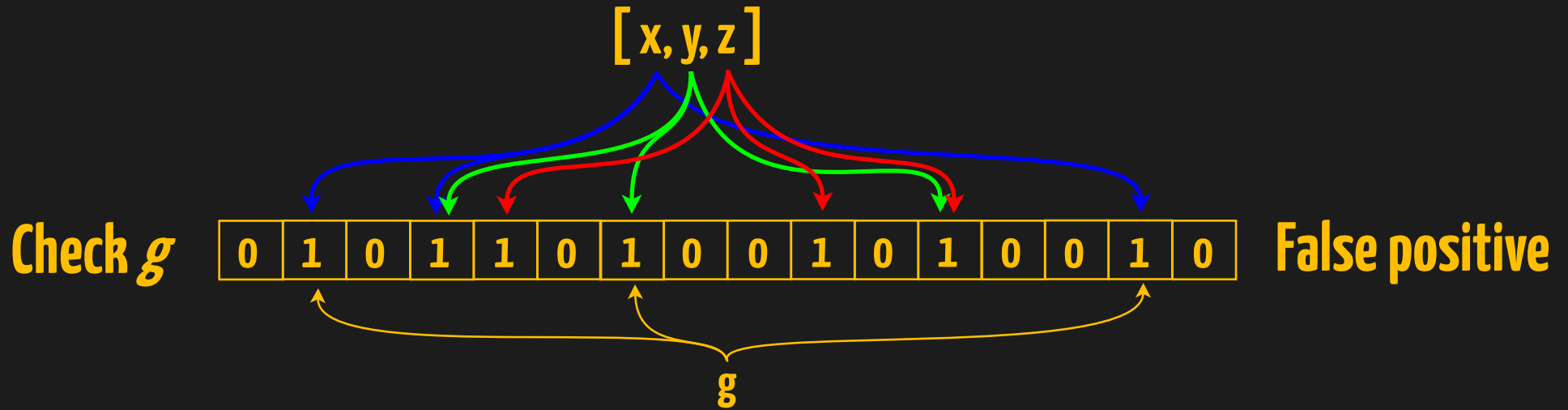




# Bloom Filter with $m=16, k=3$



# Bloom Filter with $m=16, k=3$



# Bloom Filter

## Approximate optimal size

Given:

- $n$ : the number of expected items
- $\varepsilon$ : the wanted false positive rate, where  $\varepsilon \in [0, 1]$

Then:

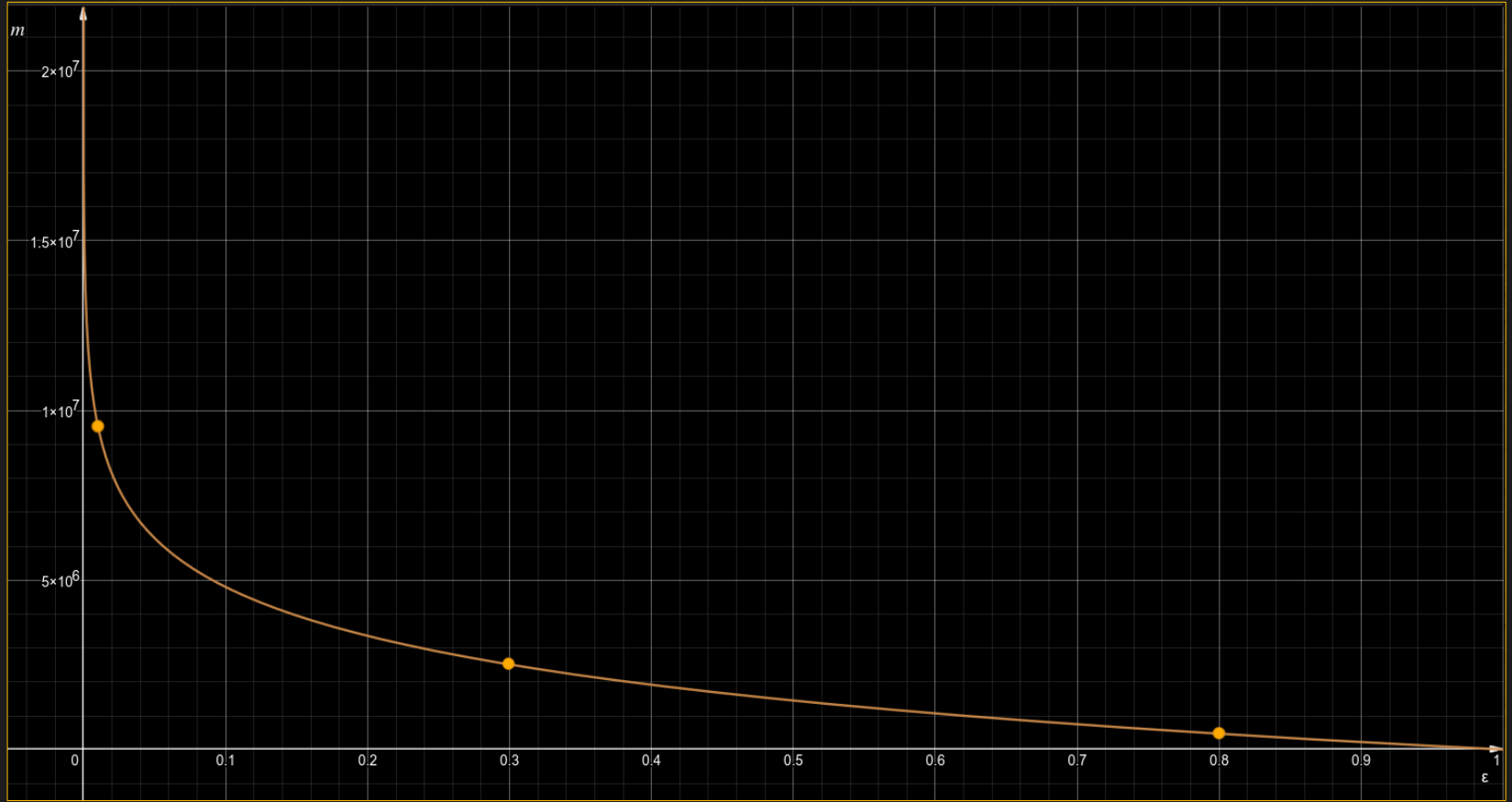
- $m = -\frac{n \cdot \ln(\varepsilon)}{\ln(2)^2}$
- $k = -\log_2(\varepsilon)$

To check 1M items, with 1% error rate:

- $m = -\frac{1,000,000 \cdot -4.6}{0.48} \simeq 9.5 \cdot 10^6 \simeq 1.2 \text{ MB}$
  - $k \simeq 6.6 = 7$
- We have a capped size!

# Bloom Filter

Given  $n = 1\text{M}$



$$\epsilon = 0.01 \Rightarrow m = 9.5\text{M}$$

$$\epsilon = 0.3 \Rightarrow m = 2.5\text{M}$$

$$\epsilon = 0.8 \Rightarrow m = 460\text{K}$$

# Bloom Filter

Fruit flies olfactory neural circuit evolved a variant of a Bloom filter to assess the novelty of odors!

It adds two additional features:

- based on similarity to previously experienced odors
- time elapsed since the odor was last experienced

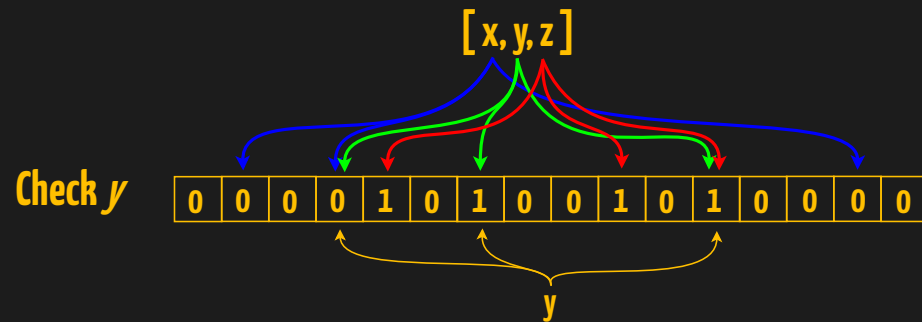
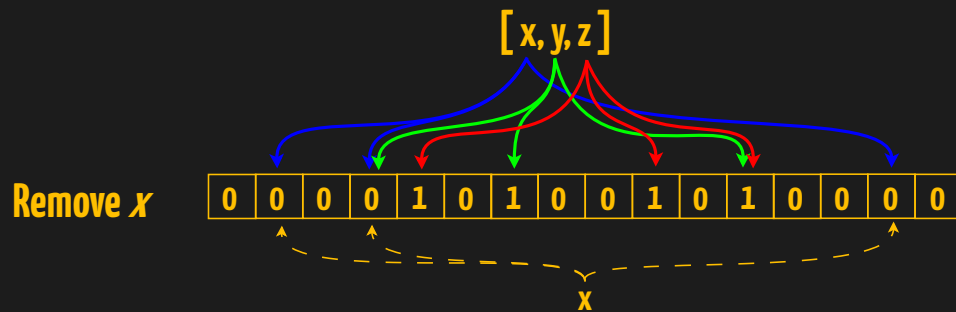
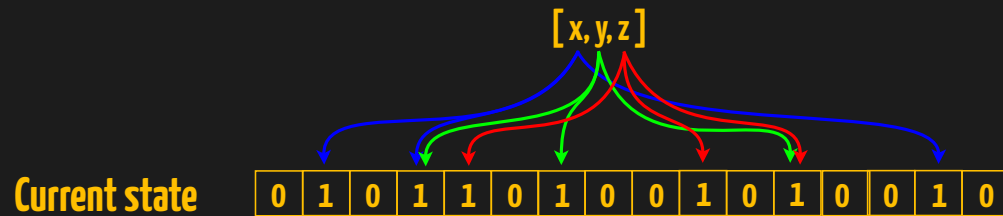


(source: <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6304992/>)

# Bloom Filter Code

```
01 class StringBloomFilter(expectedSize: Int, errorRate: Double) {
02
03     private val m: Int = -(expectedSize * ln(errorRate) / ln2squared).toInt() // bitset size
04     private val k = ceil(-log2(errorRate)).toInt() // number of hash functions
05     private val bitSet = BitSet(m)
06     private val hashers = IntStream
07         .rangeClosed(1, k)
08         .mapToObj { n → Hasher(primes[n + 4], primes[3 * n + 3]) }
09         .toList()
10
11     fun contains(item: String)=hashers.all { hasher→bitSet.get(abs(hasher.hashCode(item)) % m) }
12
13     fun add(item: String)=hashers.forEach { hasher→bitSet.set(abs(hasher.hashCode(item)) % m, true)}
14
15     private class Hasher(private val base: Int, private val multiplier: Int) {
16         fun hashCode(value: String): Int = value
17             .map { it.code }
18             .fold(base) { acc, curr → acc * multiplier + curr }
19     }
20 }
```

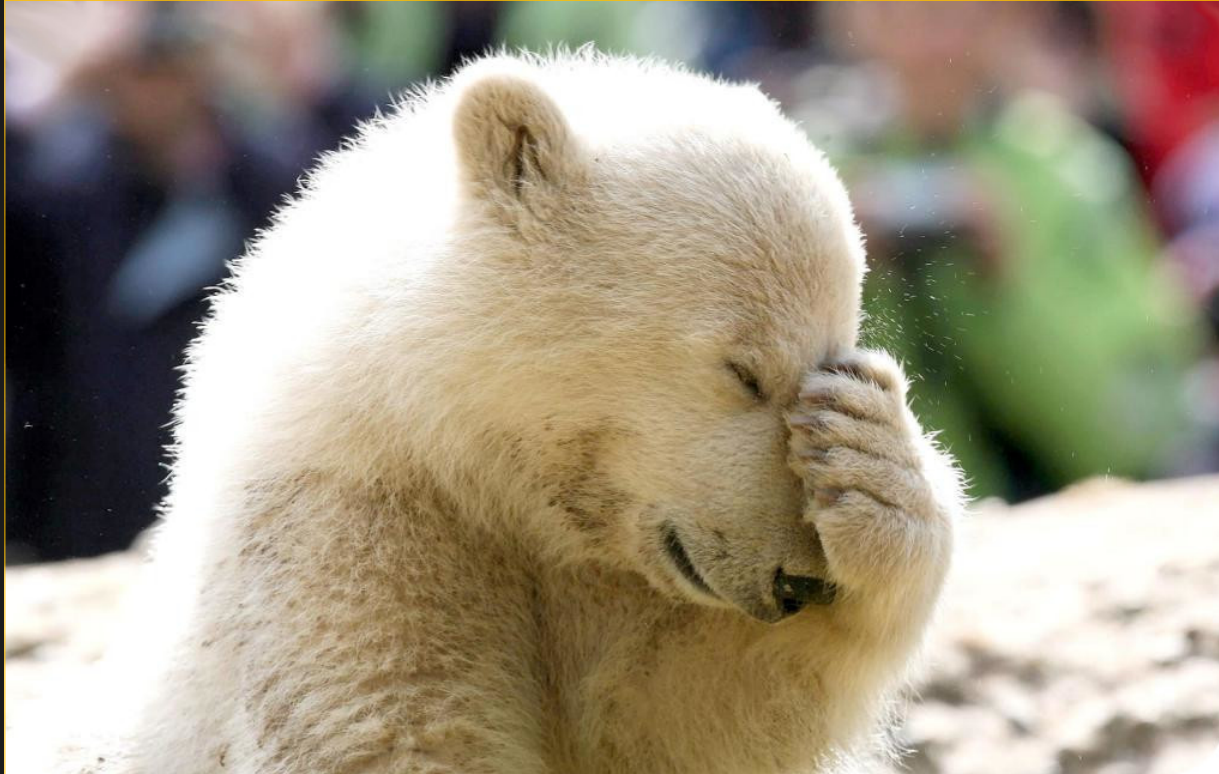
# Bloom Filter with $M=16, k=3$



False negative!

# Bloom Filter

**If we remove an item from the set, we have false negatives!**





# Counting Bloom Filter

Consists of:

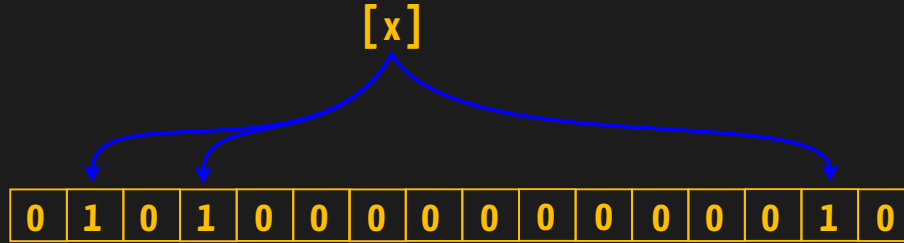
- a count array of size  $m$
- $k$  different hash functions

Membership condition:

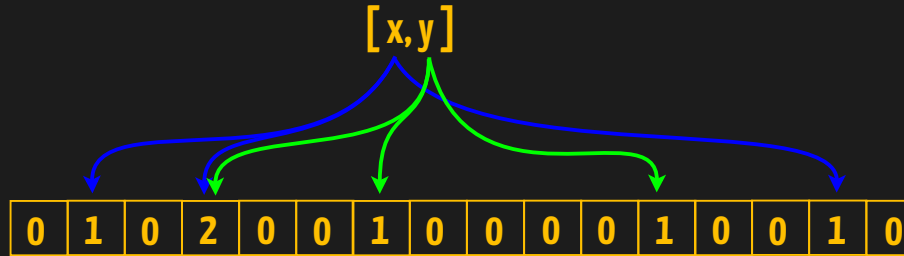
- if all counts are  $> 0$

# Counting Bloom Filter

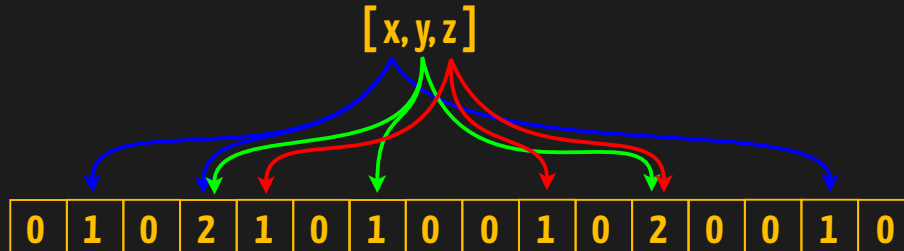
Insert  $x$



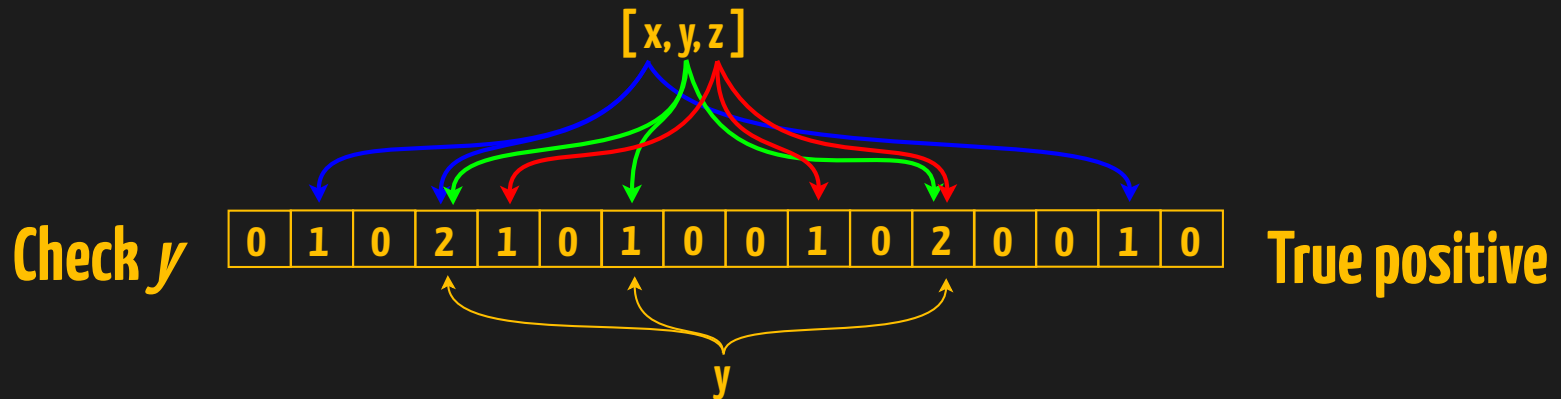
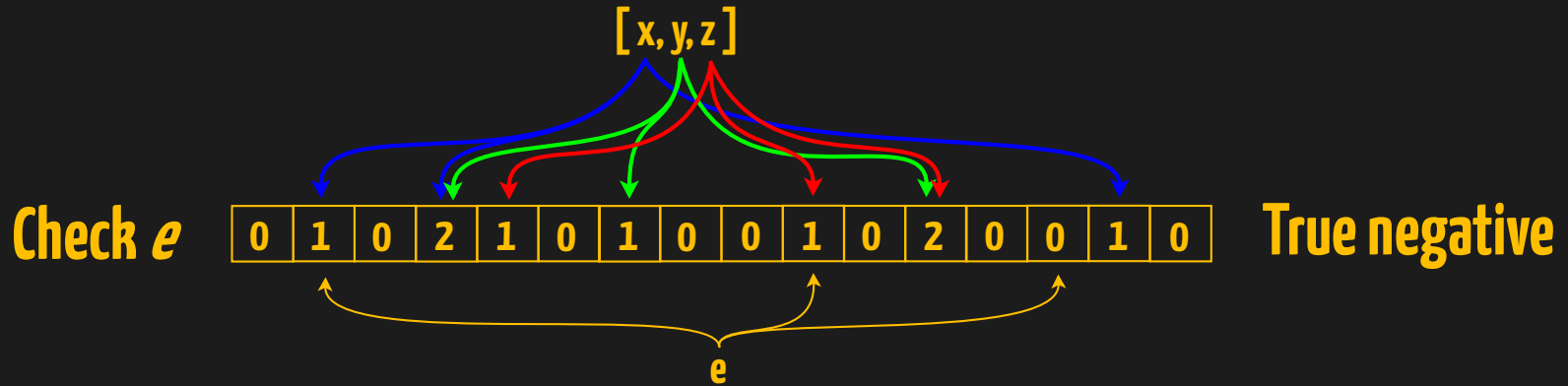
Insert  $y$



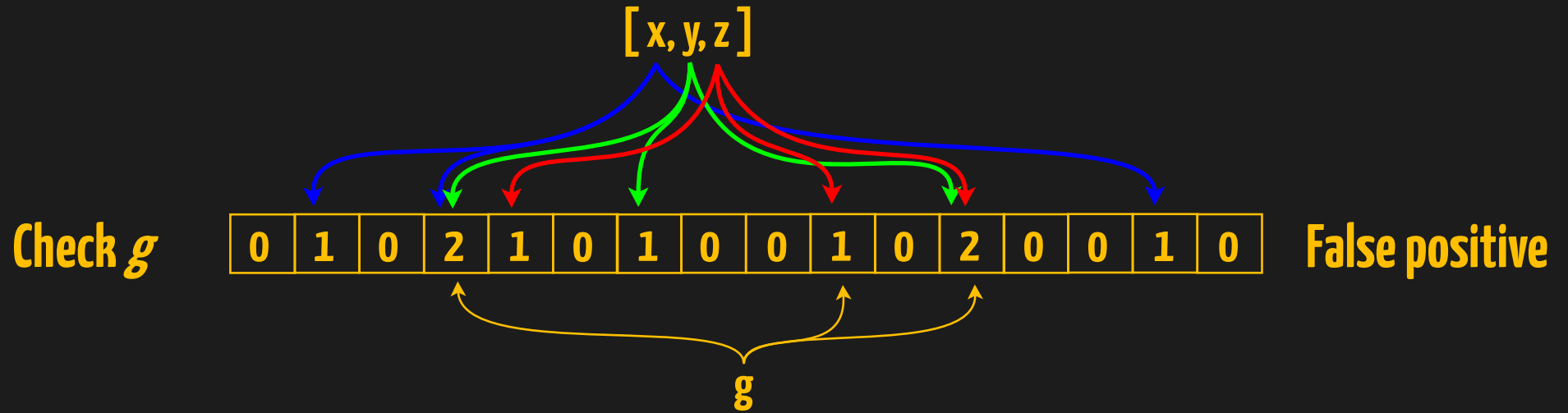
Insert  $z$



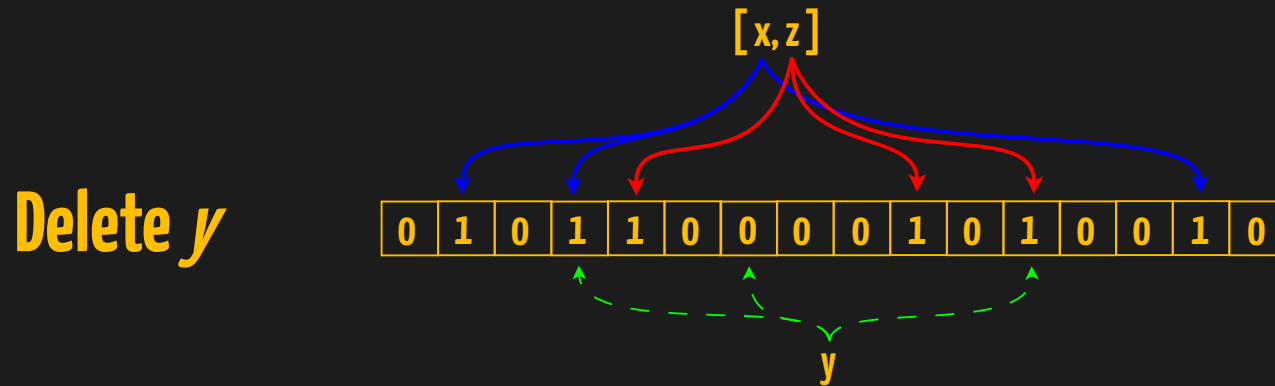
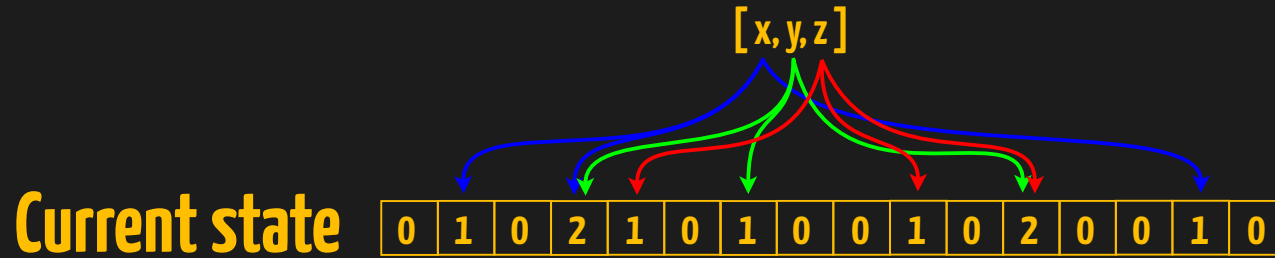
# Counting Bloom Filter



# Counting Bloom Filter



# Counting Bloom Filter



```

01 class StringCountingBloomFilter(expectedSize: Int, errorRate: Double) {
02     private val m: Int = -(expectedSize * ln(errorRate) / ln2squared).toInt()
03     private val k = ceil(-log2(errorRate)).toInt()
04     private val counters = ByteArray(m) { Byte.MIN_VALUE }
05     private val hashers = IntStream
06         .rangeClosed(1, k)
07         .mapToObj { n → Hasher(primes[n + 4], primes[3 * n + 3]) }
08         .toList()
09
10     fun add(item: String)=hashers.forEach { counters[abs(it.hashCode(item)) % m]++ }
11     fun delete(item: String)=hashers.forEach { counters[abs(it.hashCode(item)) % m]-- }
12     fun contains(item: String)=hashers.all { counters[abs(it.hashCode(item)) % m] > MIN_VALUE }
13
14     private class Hasher(private val base: Int, private val mult: Int) {
15         fun hashCode(value: String) = value.map { it.code }
16             .fold(base) { acc, curr → acc * mult + curr }
17     }
18 }
19 }

```

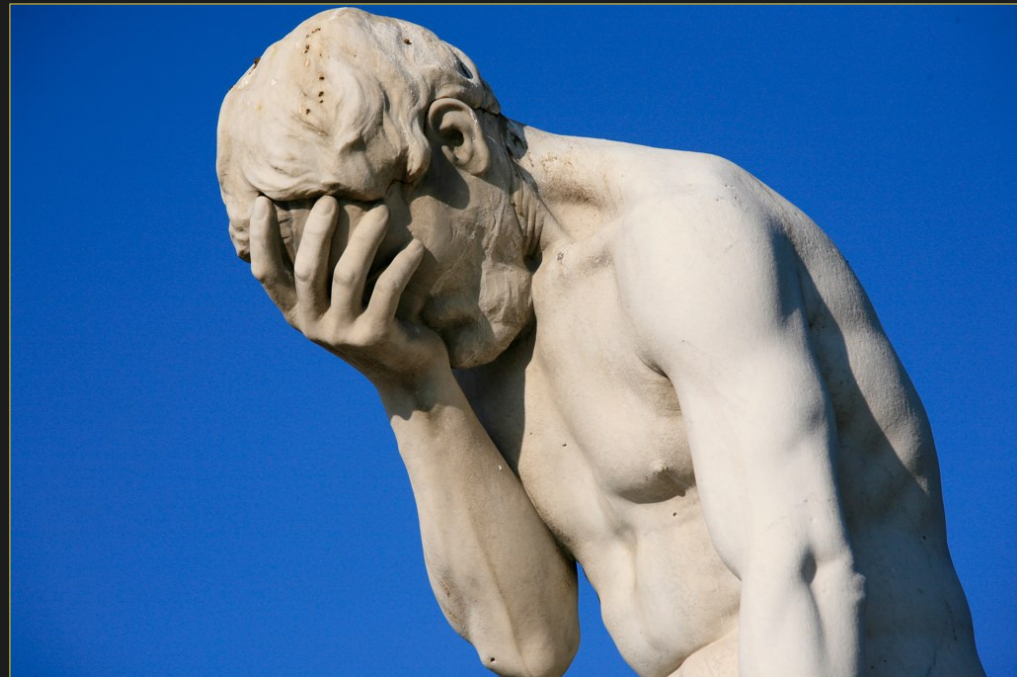
## Counting Bloom Filter Code

# Counting Bloom Filter

It uses a lot more memory than Bloom Filter:

- 8x with byte
- 16x with short
- 32x with int

It also might overflow



# Cuckoo Filter

Described by Fan, Andersen, Kaminsky, and Mitzenmacher in 2014

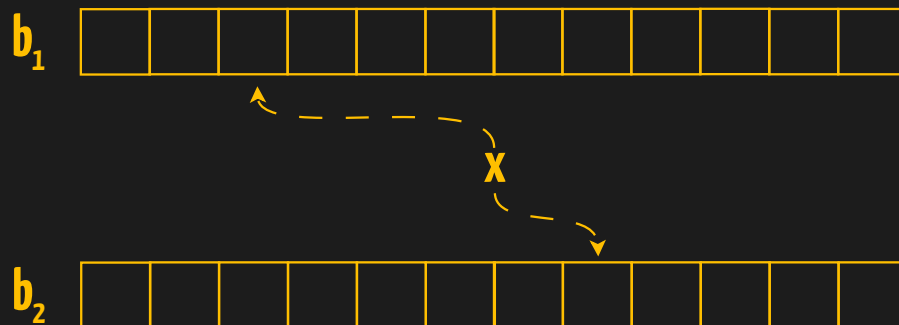
## Cuckoo Hashing:

- 1) Each item to insert has two possible locations given by two hashing functions
- 2<sup>a</sup>) If one or both of the two locations are empty, just insert
- 2<sup>b</sup>) Else randomly select one of the two locations, insert and kick out the old item
- 3) Insert the old item into the other location: if not empty, repeat this process

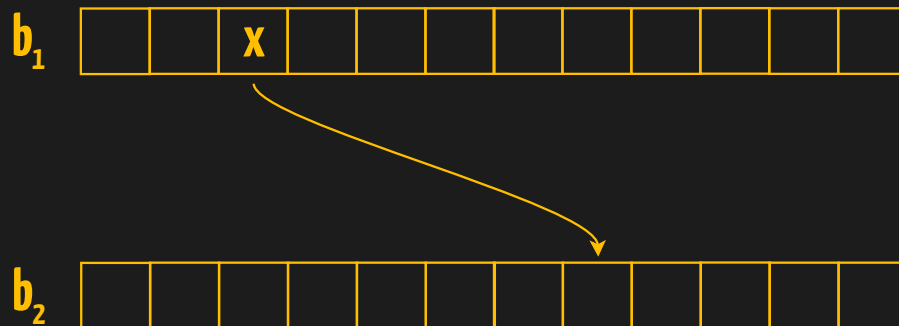


# Cuckoo Hashing

Choose location for  $x$

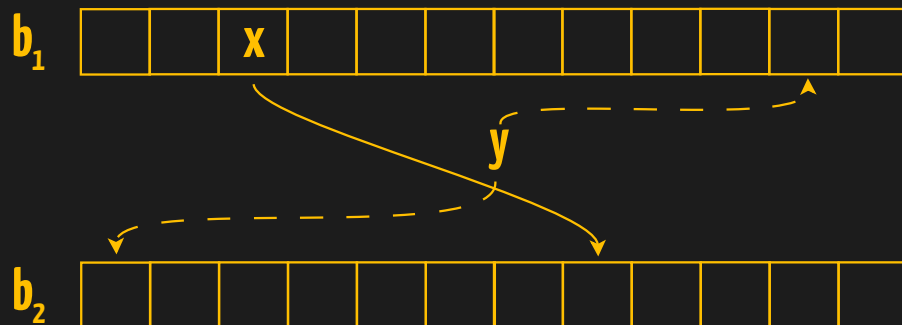


Insert and keep track

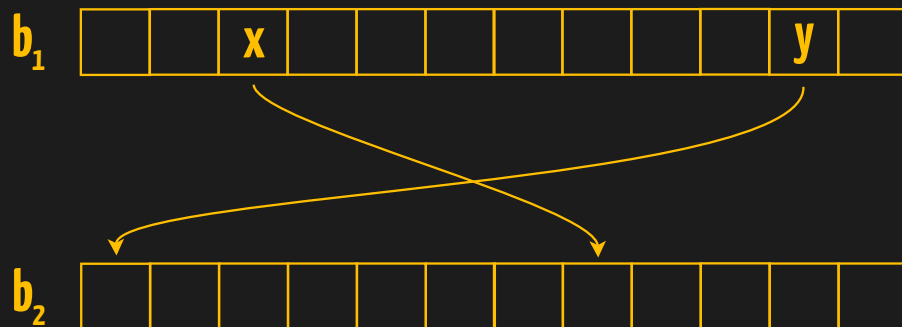


# Cuckoo Hashing

Choose location for  $y$

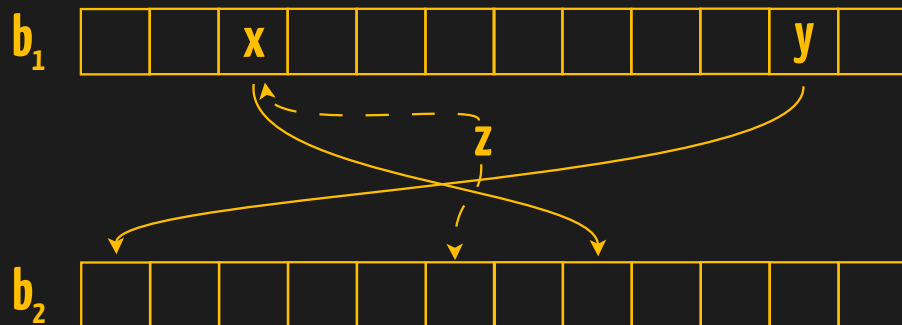


Insert and keep track

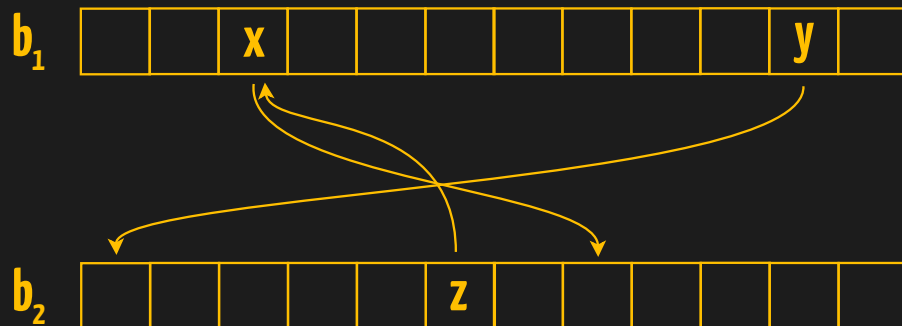


# Cuckoo Hashing

Choose location for  $z$

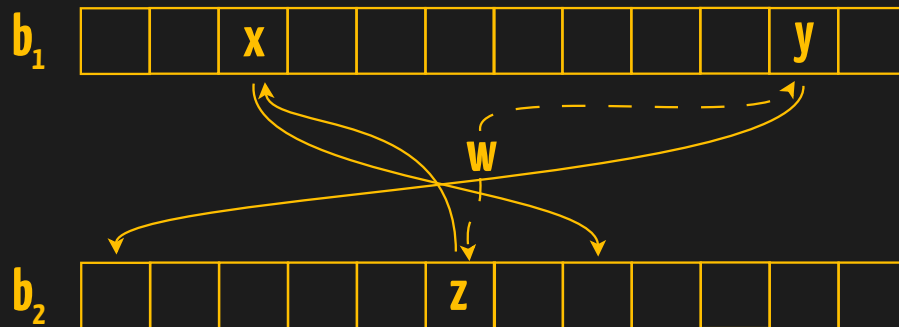


Insert and keep track

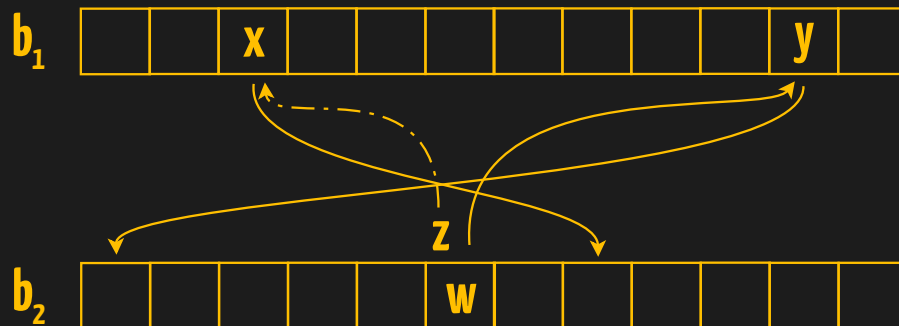


# Cuckoo Hashing

Choose location for  $w$

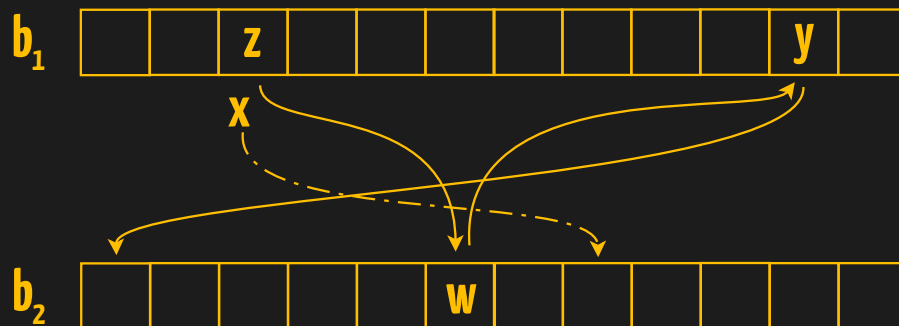


Randomly choose bucket #2  
and kick out  $z$ !

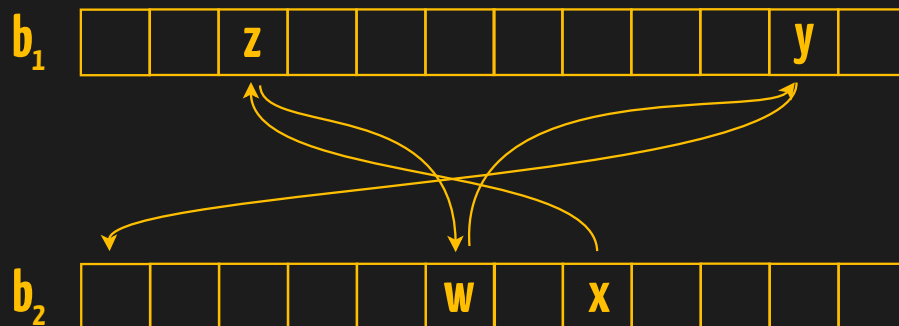


# Cuckoo Hashing

z kicks out x



Insert x into its  
alternative location



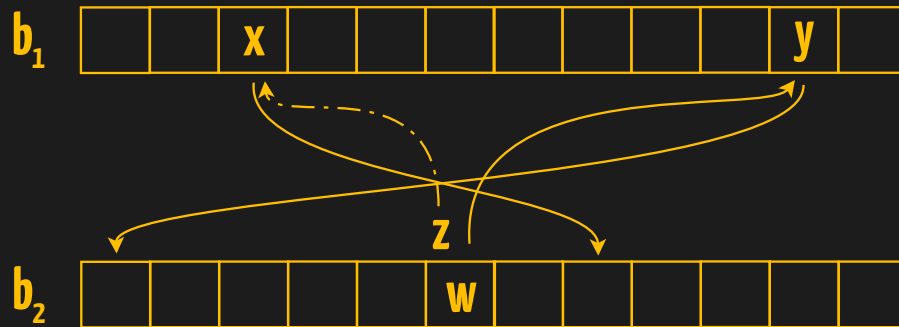
# Cuckoo Hashing

- $O(1)$  lookup runtime complexity: each item will be in one of the two locations
- Insertion can fail if there's a loop in items to be kicked out
- Incredibly enough, amortized runtime complexity of insertion is  $O(1)$

# Cuckoo Filter

## Based on Cuckoo Hashing

- to save memory instead of storing the whole item we store a fingerprint ( $f$  bit)
- how can we get the alternate location if we don't have the kicked out item?



- create a hash that keeps track of fingerprint

# Cuckoo Filter

- a fingerprint function returning a  $f$ -bit value

- two hash functions 
$$\begin{cases} h_1(x) = \text{hash}(x) \\ h_2(x) = h_1(x) \oplus \text{hash}(\text{fingerprint}(x)) \end{cases}$$
$$\Downarrow$$
$$h_1(x) = h_2(x) \oplus \text{hash}(\text{fingerprint}(x))$$



# Cuckoo Filter

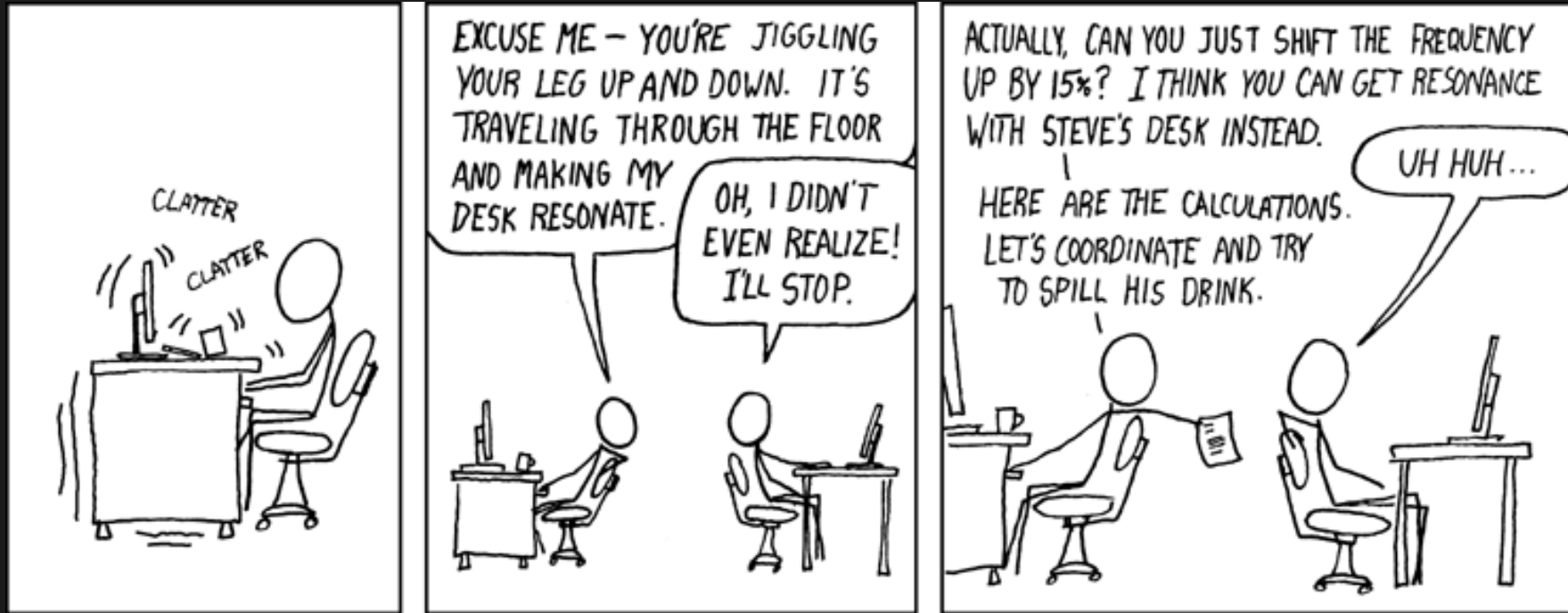
Cuckoo filter requires  $\frac{\log_2 \left( \frac{1}{\epsilon} \right) + 1 + \log_2 (b)}{\alpha}$  bits per key

where  $\alpha$  is the load factor of the cuckoo hash table

# Cuckoo Filter Code

```
01 fun insert(item: String): Boolean {
02     var f = fingerprint(item)
03     val index1 = hash(item)
04     if (bucket[index1] == EMPTY) {
05         bucket[index1] = f
06         return true
07     }
08     val index2 = index1 xor hash(f)
09     if (bucket[index2] == EMPTY) {
10         bucket[index2] = f
11         return true
12     }
13
14     // relocate existing items
15     var i = if (Random.nextBoolean()) index1 else index2
16     for (n in 0..  
MAX_KICKS) {
17         f = bucket[i].also { bucket[i] = f }
18         i = i xor hash(f)
19         if (bucket[i] == EMPTY) {
20             bucket[i] = f
21         }
22     }
23     return false // Max displacement attempts reached
24 }
```

# Frequency



(source: <https://xkcd.com/228/>)

# Frequency

Counting the number of times an item appears in a stream

- Count-Min Sketch

# Frequency Use Cases

- Unique users of a service
- Top k items (e.g. top players in online gaming)
- Network traffic analysis

# Count-Min Sketch

Invented by G. Cormode et al in 2005

- $d$  hash functions (pairwise independent)
- an array *count* of size  $d \times w$  for counting the items, where:

$$d = \ln\left(\frac{1}{\delta}\right)$$

$\varepsilon$  is the error rate and  $\delta$  its probability

$$w = \left(\frac{e}{\varepsilon}\right)$$

If we want an error of at most 0.1% (of the sum of all frequencies) with 99.9% certainty, then:

$$w = \left(\frac{e}{0.001}\right) \simeq 2,718 \quad d = \ln\left(\frac{1}{0.001}\right) \simeq 6.9 = 7$$

Using 32 bit counters,  $\text{sizeof}(\textit{count}) = w \cdot d \cdot 4 \simeq 76 \text{ KB}$

# Count-Min Sketch

$d = 4, w = 16$

Initial state

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Hash Function #1

Hash Function #2

Hash Function #3

Hash Function #4

# Count-Min Sketch

$d = 4, w = 16$

Insert  $x$

$$h_1(x) = 2$$

$$h_2(x) = 13$$

$$h_3(x) = 12$$

$$h_4(x) = 7$$

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

Hash Function #1

Hash Function #2

Hash Function #3

Hash Function #4



# Count-Min Sketch

$d = 4, w = 16$

Insert  $y$

$$h_1(y) = 6$$

$$h_2(y) = 13$$

$$h_3(y) = 8$$

$$h_4(y) = 2$$

0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0
0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0

Hash Function #1

Hash Function #2

Hash Function #3

Hash Function #4

# Count-Min Sketch

$d = 4, w = 16$

Insert  $x$

$$h_1(x) = 2$$

$$h_2(x) = 13$$

$$h_3(x) = 12$$

$$h_4(x) = 7$$

0	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0
0	0	0	0	0	0	0	0	1	0	0	0	2	0	0	0
0	0	1	0	0	0	0	2	0	0	0	0	0	0	0	0

Hash Function #1

Hash Function #2

Hash Function #3

Hash Function #4

# Count-Min Sketch

$d = 4, w = 16$

0	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0
0	0	0	0	0	0	0	0	1	0	0	0	2	0	0	0
0	0	1	0	0	0	0	2	0	0	0	0	0	0	0	0

Hash Function #1

Hash Function #2

Hash Function #3

Hash Function #4

$$\begin{aligned}\text{Count}(x) &= \min(\text{count}[0][h_1(x)], \\ &\quad \text{count}[1][h_2(x)], \\ &\quad \text{count}[2][h_3(x)], \\ &\quad \text{count}[3][h_4(x)]) \\ &= \min(2, 3, 2, 2) = 2\end{aligned}$$

$$\begin{aligned}\text{Count}(y) &= \min(\text{count}[0][h_1(y)], \\ &\quad \text{count}[1][h_2(y)], \\ &\quad \text{count}[2][h_3(y)], \\ &\quad \text{count}[3][h_4(y)]) \\ &= \min(1, 3, 1, 1) = 1\end{aligned}$$

# Count-Min Sketch

If there are collisions on hashes we get higher counts

# Count-Min Sketch Code

```
01 class CountMinSketch(d: Int, private val w: Int) {
02     private val count = Array(d) { LongArray(w) }
03     private val hashers = IntStream
04         .rangeClosed(1, h)
05         .mapToObj { n → Hasher(w, Primes.primes[n * 17 + 4], Primes.primes[3 * n + 3]) }
06         .toList()
07
08     fun add(item: String) = hashers.forEachIndexed { idx, hasher →
09         count[idx][hasher.hashCode(item)]++
10     }
11
12     fun count(item: String) = hashers
13         .mapIndexed { idx, hasher → count[idx][hasher.hashCode(item)] }
14         .min()
15
16     private class Hasher(val size: Int, val base: Long, val multiplier: Long) {
17         fun hashCode(value: String) = value.map { it.code }
18             .fold(base) { acc, curr → acc * multiplier + curr }
19             .mod(size)
20     }
21 }
```

# Cardinality



# Cardinality

Counting the number of distinct items in a collection

- HyperLogLog

# Cardinality Use Cases

- **BigQuery:** APPROX\_COUNT\_DISTINCT, APPROX\_QUANTILES, APPROX\_TOP\_COUNT, APPROX\_TOP\_SUM
- **Snowflake:** HLL, HLL\_ACCUMULATE, HLL\_COMBINE, HLL\_ESTIMATE, HLL\_ACCUMULATE, HLL\_COMBINE
- **Number of users in search engines (where ads are paid per user)**
- **Materialized views in Data Warehouses**



# HyperLogLog

Flajolet et al. 2007

## Binary representation of random numbers:

- $\frac{1}{2}$  start with “0”
  - $\frac{1}{2}$  start with “1”
  - $\frac{1}{4}$  start with “00”
  - $\frac{1}{4}$  start with “01”
  - $\frac{1}{4}$  start with “10”
  - $\frac{1}{4}$  start with “11”
  - $\frac{1}{8}$  start with “000”
  - $\frac{1}{8}$  start with “001”
  - $\frac{1}{8}$  start with “010”
  - $\frac{1}{8}$  start with “011”
  - $\frac{1}{8}$  start with “111”
  - $\frac{1}{8}$  start with “101”
  - $\frac{1}{8}$  start with “110”
  - $\frac{1}{8}$  start with “111”
- General rule:  $p(\text{first } k \text{ bits}) = 2^{-k}$

# HyperLogLog

Underlying idea: let's imagine we have a big set of items hashed to numbers; if we add to the set the hash of a new item starting with "0000" we can say that it's likely that the set has size  $2^4 = 16$ .

## Caveats:

- only works on big cardinalities
- hash function must return uniformly distributed binary representations

# HyperLogLog

Consists of:

- a hash function
- an array of  $\text{int } M$

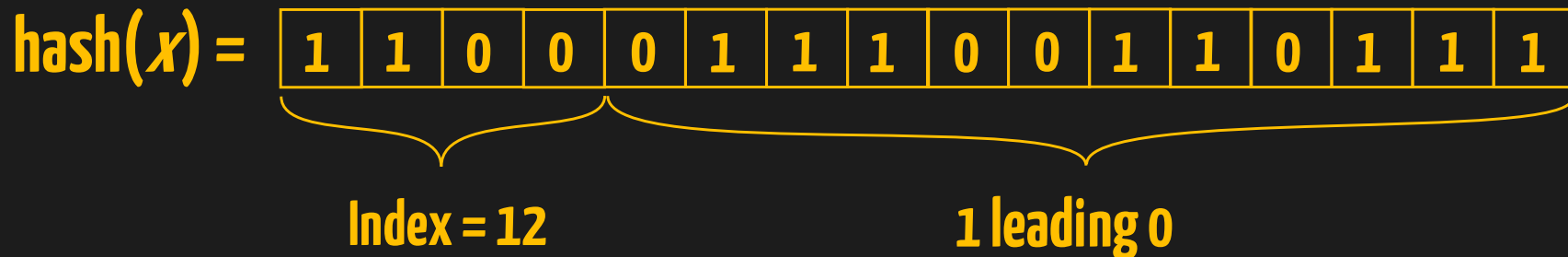
With only one hash function, we risk that one single item can skew the result. We could use multiple hash functions, but that takes extra computation time

# HyperLogLog

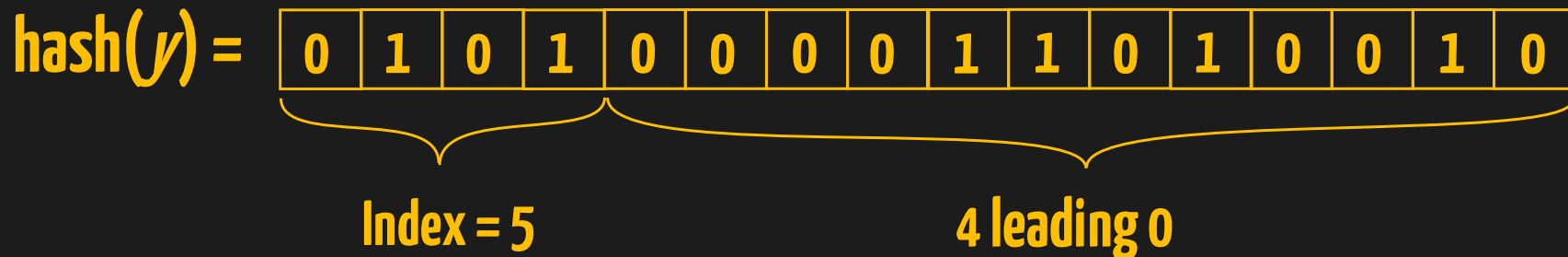
Since we don't want multiple hash functions, we use only one, but we pretend they're  $2^b$ .



# HyperLogLog



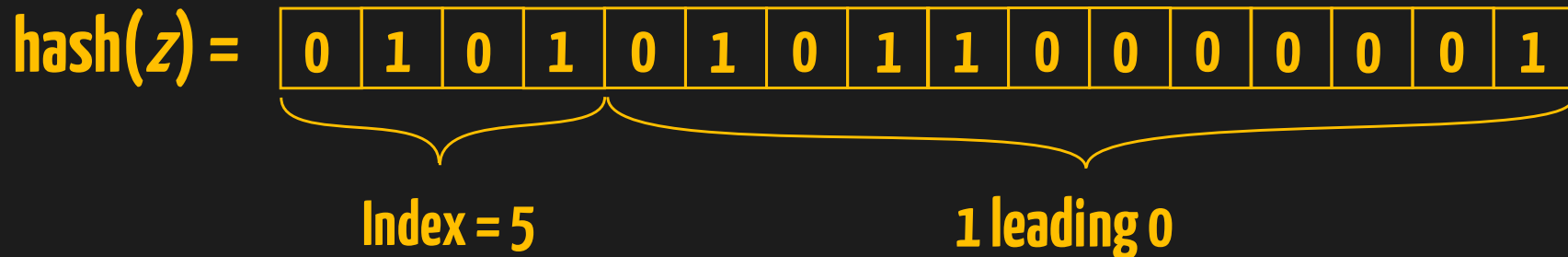
# HyperLogLog



Array M

0	0	0	0	0	4	0	0	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# HyperLogLog



Array M

0	0	0	0	0	4	0	0	0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# HyperLogLog

## Get the cardinality

- compute a normalized harmonic mean  $Z = \left( \sum_{i=1}^m 2^{-M[i]} \right)^{-1}$  of the values of  $M$
- return the harmonic mean with a correction factor  $\alpha = Z \cdot |M|^2 \cdot \alpha$

where  $\alpha = \begin{cases} \alpha_{16} = 0.673 \\ \alpha_{32} = 0.697 \\ \alpha_{64} = 0.709 \\ \alpha_m = \frac{0.7213}{1 + \frac{1.079}{m}} \text{ for } m \geq 128 \end{cases}$

$$\text{Error rate} = \frac{1.04}{\sqrt{(m)}}$$



# HyperLogLog

Array M 

4	5	2	3	5	4	7	2	6	5	4	5	3	6	2	5
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

$$\text{cardinality} = Z \cdot M^2 \cdot \alpha \simeq 0.72 \cdot 256 \cdot 0.673 \simeq 124.59$$

Array M 

4	5	12	3	5	4	7	2	6	5	4	5	3	6	2	5
---	---	----	---	---	---	---	---	---	---	---	---	---	---	---	---

$$\text{cardinality} = Z \cdot M^2 \cdot \alpha \simeq 0.88 \cdot 256 \cdot 0.673 \simeq 152.06$$

# HyperLogLog Code

```
01 class HyperLogLog {
02     private val p: Int = 4
03     private val m: Int = 2.0.pow(p.toDouble()).toInt()
04     private val alpha = 0.673
05     private val buckets = IntArray(m)
06
07     fun add(item: String) {
08         val hash = item.hashCode()
09         val index = hash.ushr(32 - p)
10         val leadingZeroes = Integer.numberOfLeadingZeros(hash shl p) + 1
11         buckets[index] = max(buckets[index], leadingZeroes)
12     }
13
14     fun count(): Long {
15         val harmonicMean = buckets.sumOf { 1.0 / (1 shl it) }
16         val estimate = alpha * (m * m).toDouble() / harmonicMean
17         return if (estimate ≤ 5.0 / 2.0 * m) {
18             (m * ln(m.toDouble() / estimate)).toLong()
19         } else {
20             estimate.toLong()
21         }
22     }
23 }
```

# Links

- Bloom filter: <https://dl.acm.org/doi/pdf/10.1145/362686.362692>
- Cuckoo Filter: <https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf>
- HyperLogLog: <https://algo.inria.fr/flajolet/Publications/FlFuGaMe07.pdf>
- Count-min Sketch: <http://dimacs.rutgers.edu/~graham/pubs/papers/cm-full.pdf>
- Code for this presentation:  
<https://github.com/andreaiacono/TalkProbabilisticDataStructures>

**Questions?**