Intro to probabilistic data structures

Download this presentation here:

About Me

- Software Engineer
- Currently at Spotify
- Interest in algorithms & DS

Agenda

Probabilistic Data Structures for this classes of problems:

- membership
- cardinality
- frequency

PDS will return approximate values

Membership


YOU GET A MEMBERSHIP, AND
YOU GET A MEMBERSHIP!
EVERYBODY GETS A
MEMBERSHIP!

Membership use cases

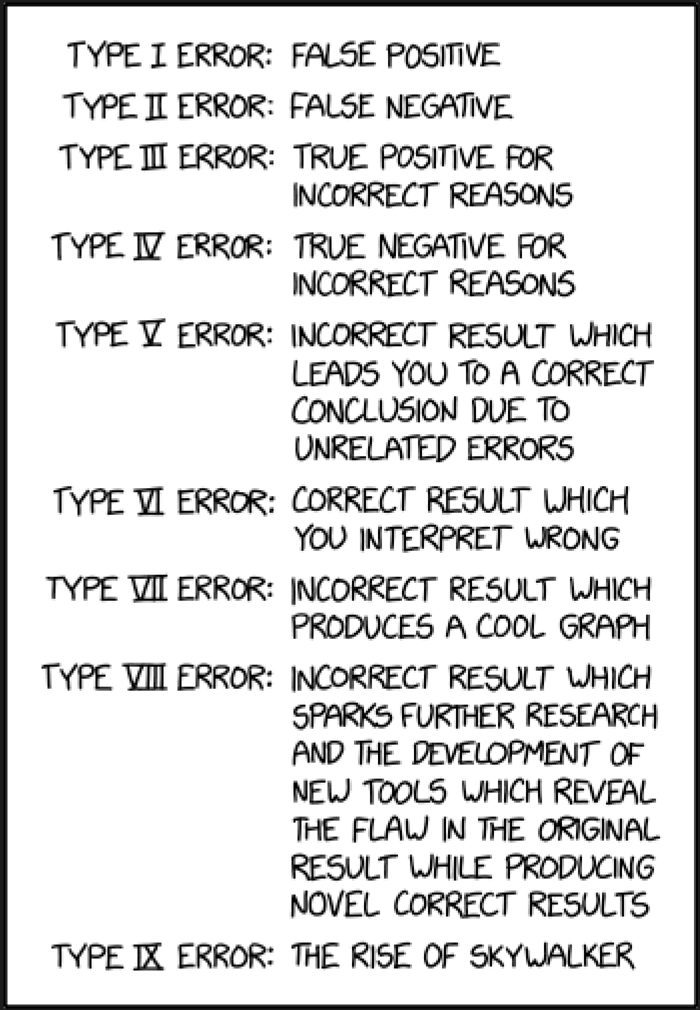
In general when wrong answer do not involve correctness but more work

- 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



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

Bloom Filter

Invented by Burton H. Bloom in 1970

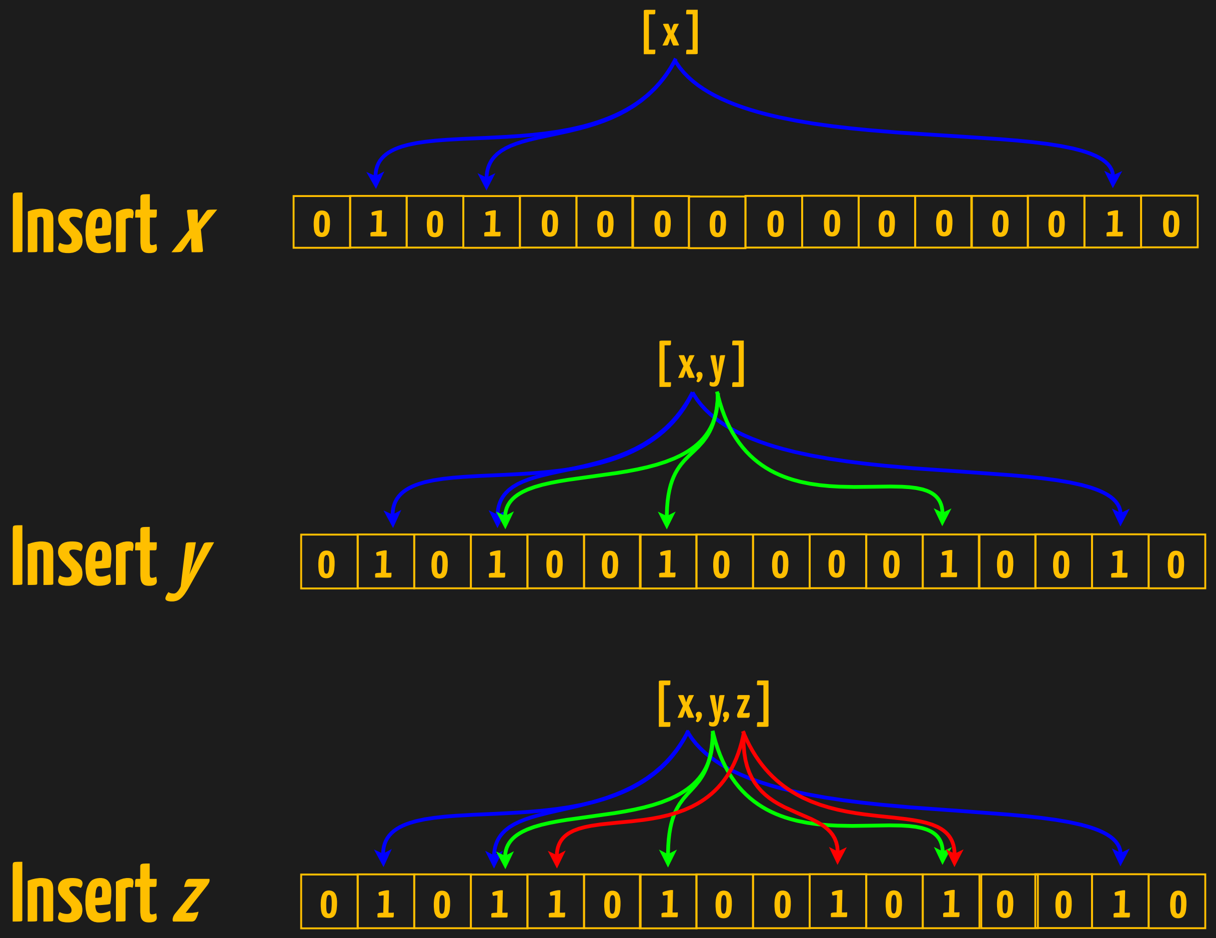
Consists of :

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

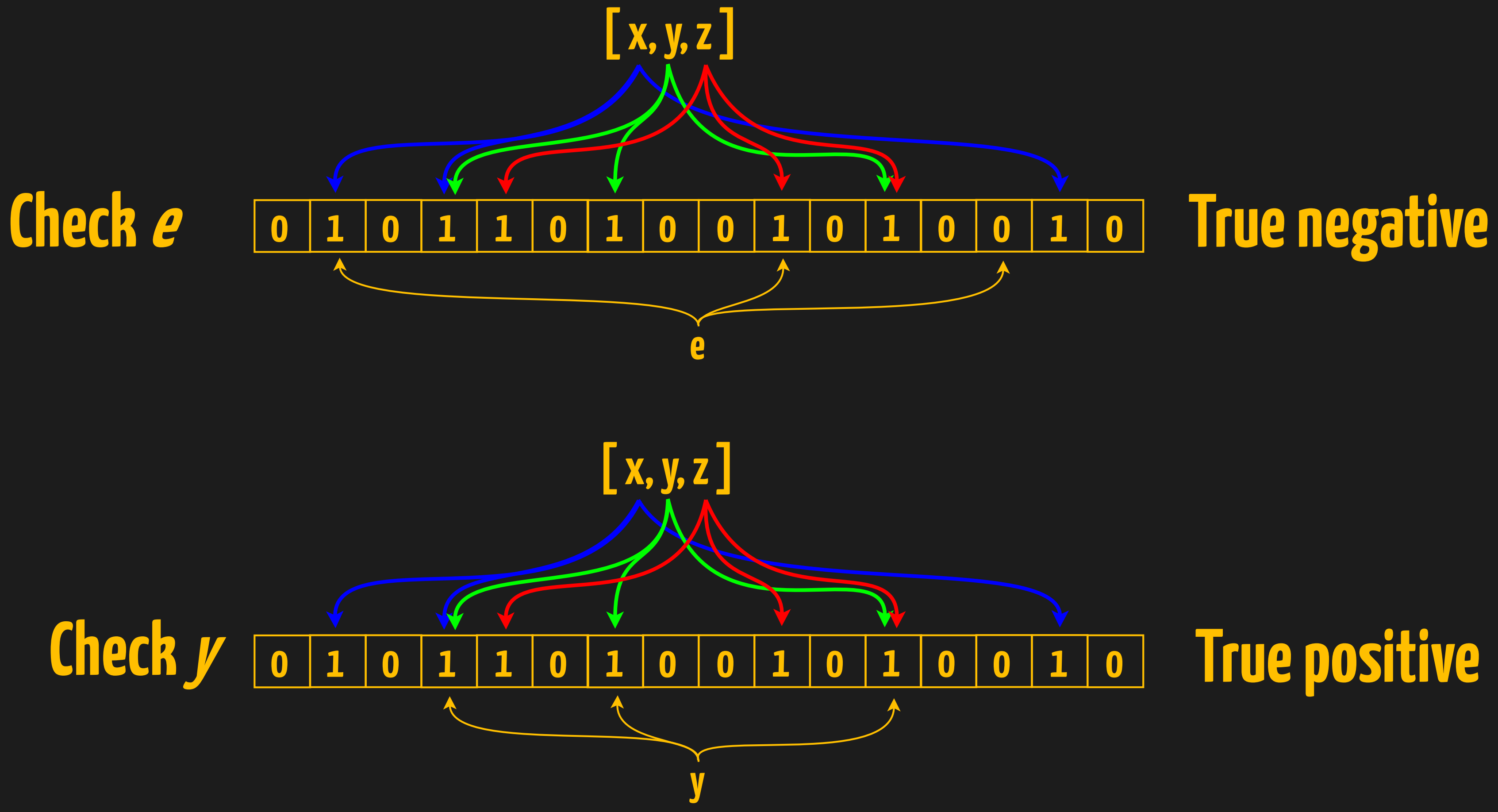
Membership check :

- if all bits are set

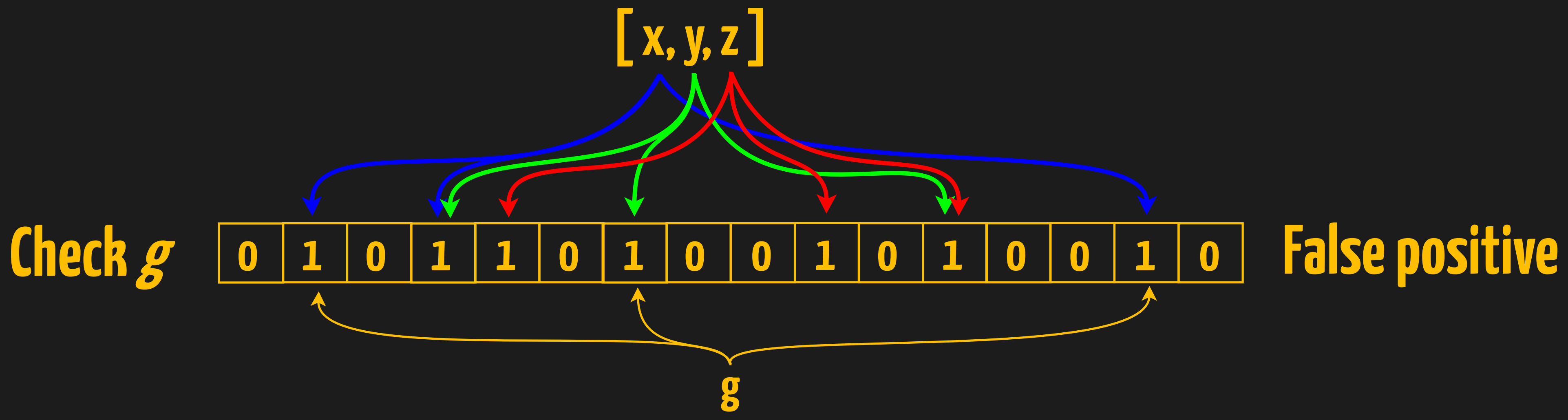
Bloom Filter with $m=16, k=3$



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
- ε: the wanted false positive rate, where $\varepsilon \in [0, 1]$

Then:

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

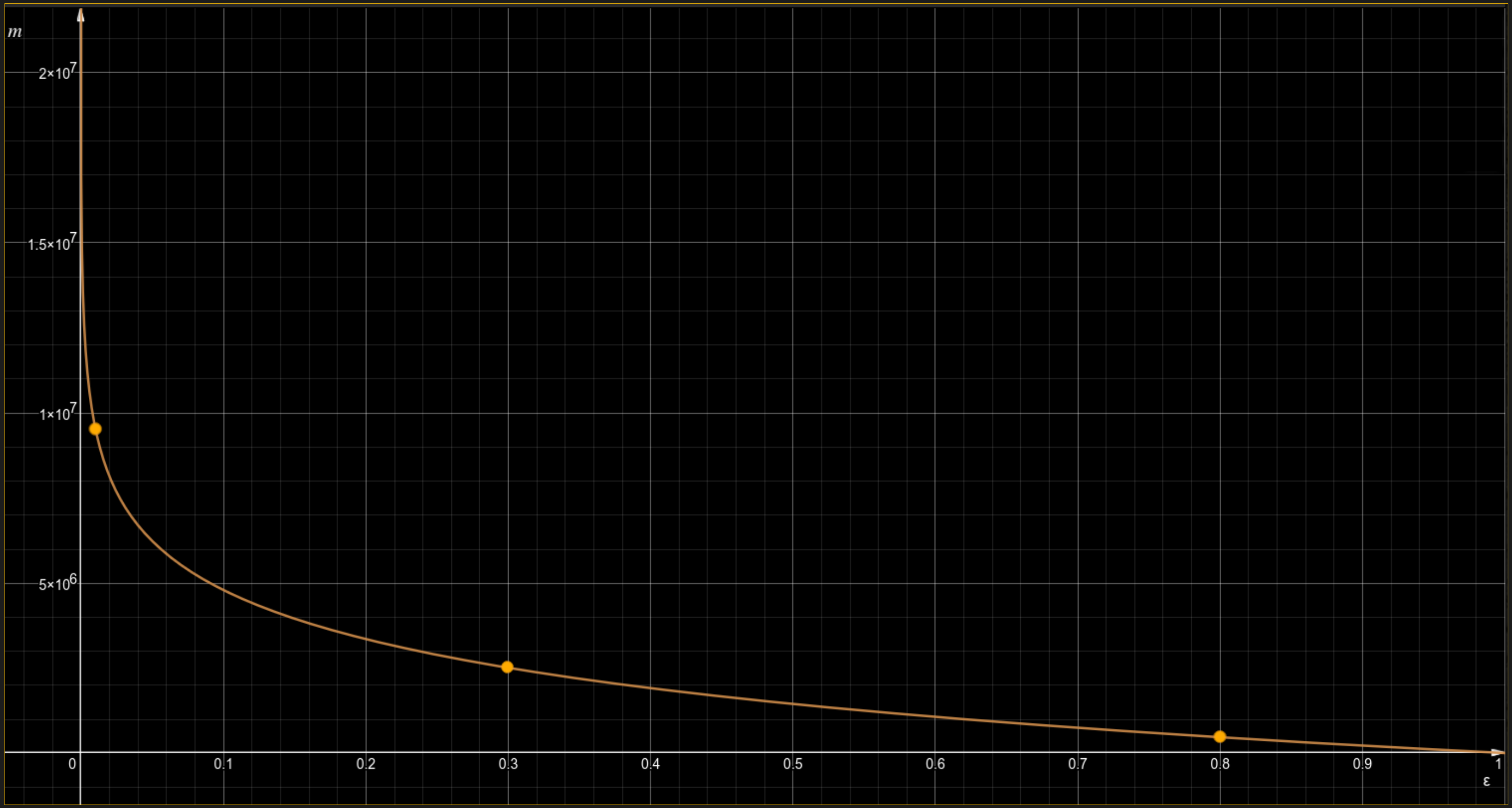
To check 1M items, with 1% error rate:

- $m = -\frac{1{,}000{,}000 \cdot -4.6}{0.48} \simeq 9.5 \cdot 10^6 \simeq 1.2\,\mathrm{MB}$
- $k \simeq 6.6 = 7$

We have a capped size!

Bloom Filter

Given $n = 1M$



$\varepsilon = 0.01 \Rightarrow m = 9.5M$ $\varepsilon = 0.3 \Rightarrow m = 2.5M$ $\varepsilon = 0.8 \Rightarrow m = 460K$

Bloom Filter Code

```kotlin
class StringBloomFilter(expectedSize: Int, errorRate: Double) {

  private val m: Int = -(expectedSize * ln(errorRate) / ln2squared).toInt() // bitset size
  private val k = ceil(-log2(errorRate)).toInt() // number of hash functions
  private val bitSet = BitSet(m)
  private val hashers = IntStream
      .rangeClosed(1, k)
      .mapToObj { n -> Hasher(primes[n + 4], primes[3 * n + 3]) }
      .toList()

  fun contains(item: String)=hashers.all { hasher->bitSet.get(abs(hasher.hashCode(item)) % m) }

  fun add(item: String)=hashers.forEach { hasher->bitSet.set(abs(hasher.hashCode(item)) % m, true)}

  private class Hasher(private val base: Int, private val multiplier: Int) {
      fun hashCode(value: String): Int = value
          .map { it.code }
          .fold(base) { acc, curr -> acc * multiplier + curr }
  }
}
```

Bloom Filter

Fun Fact

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

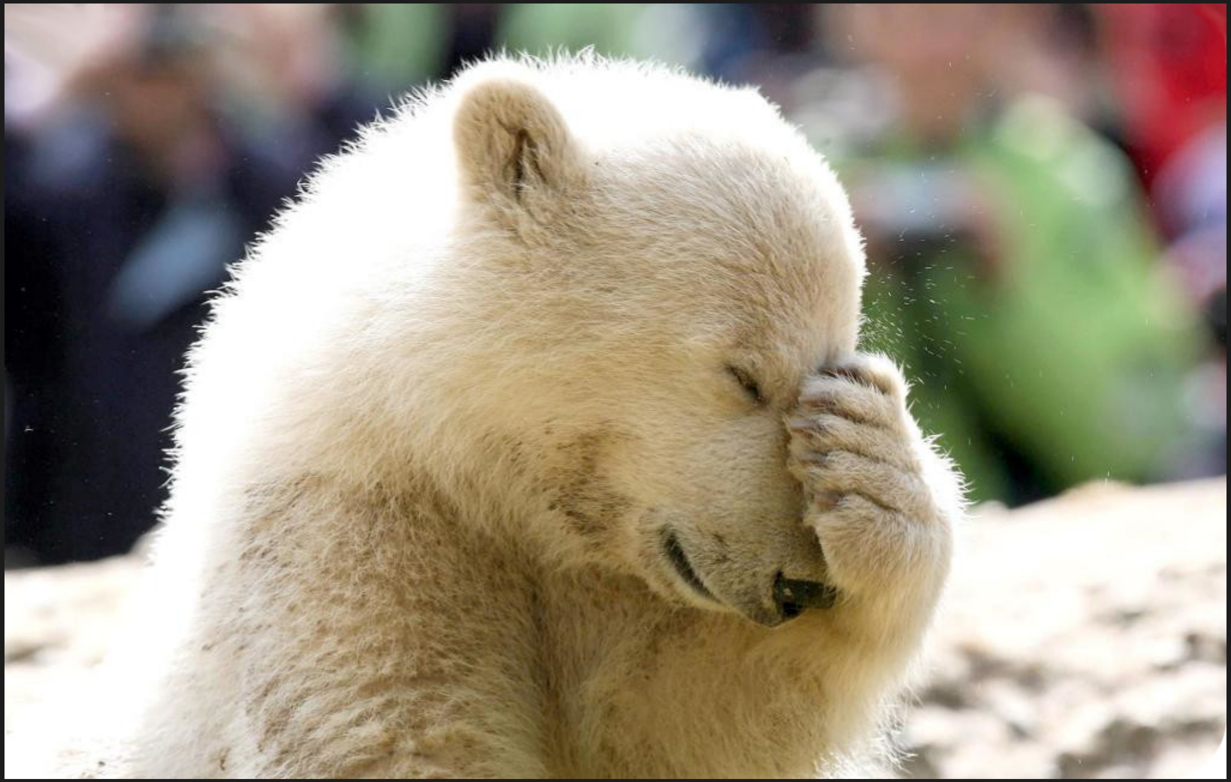
It adds two additional features:

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

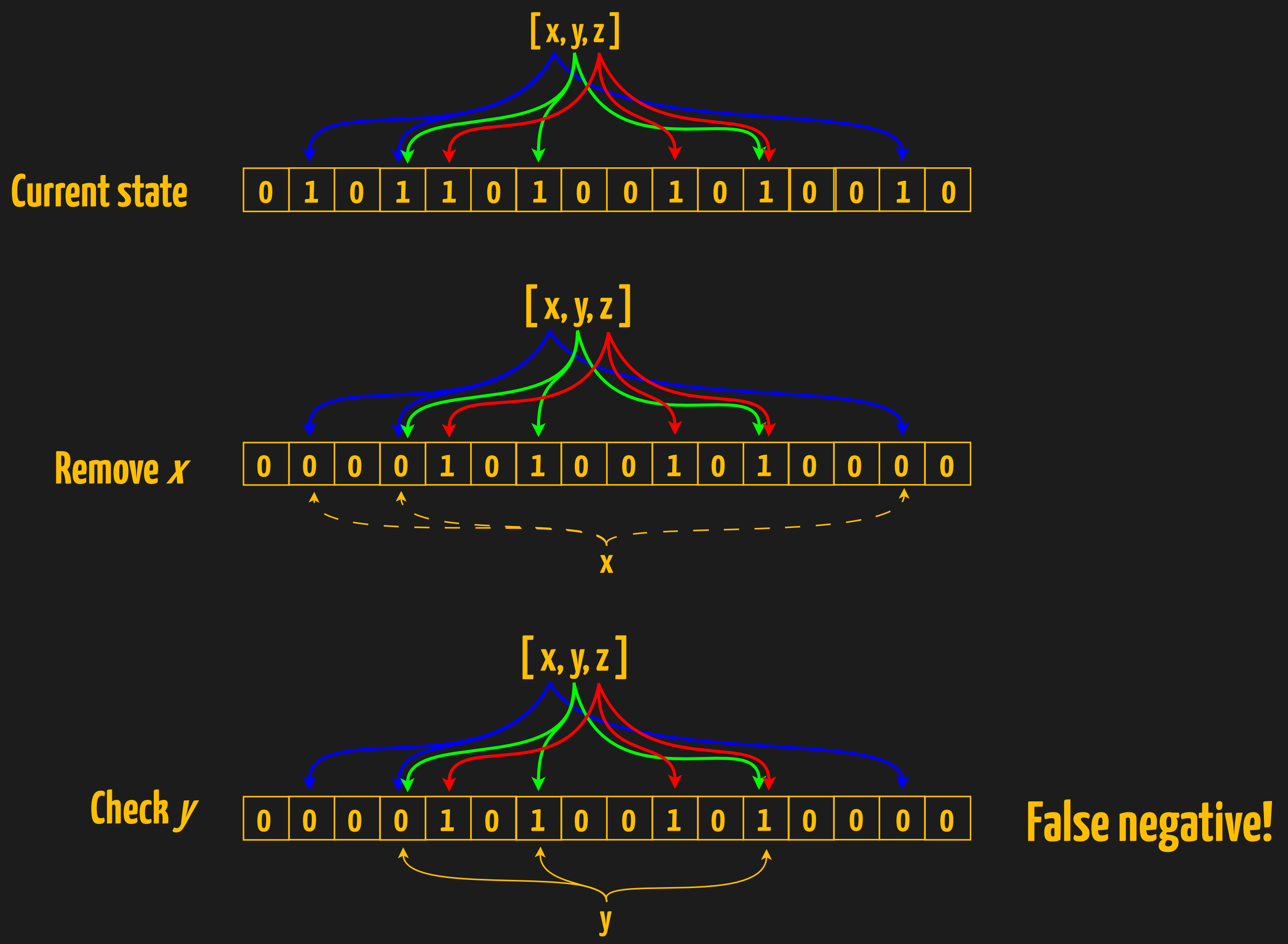
(source: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6304992/)

Bloom Filter

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



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



Counting Bloom Filter

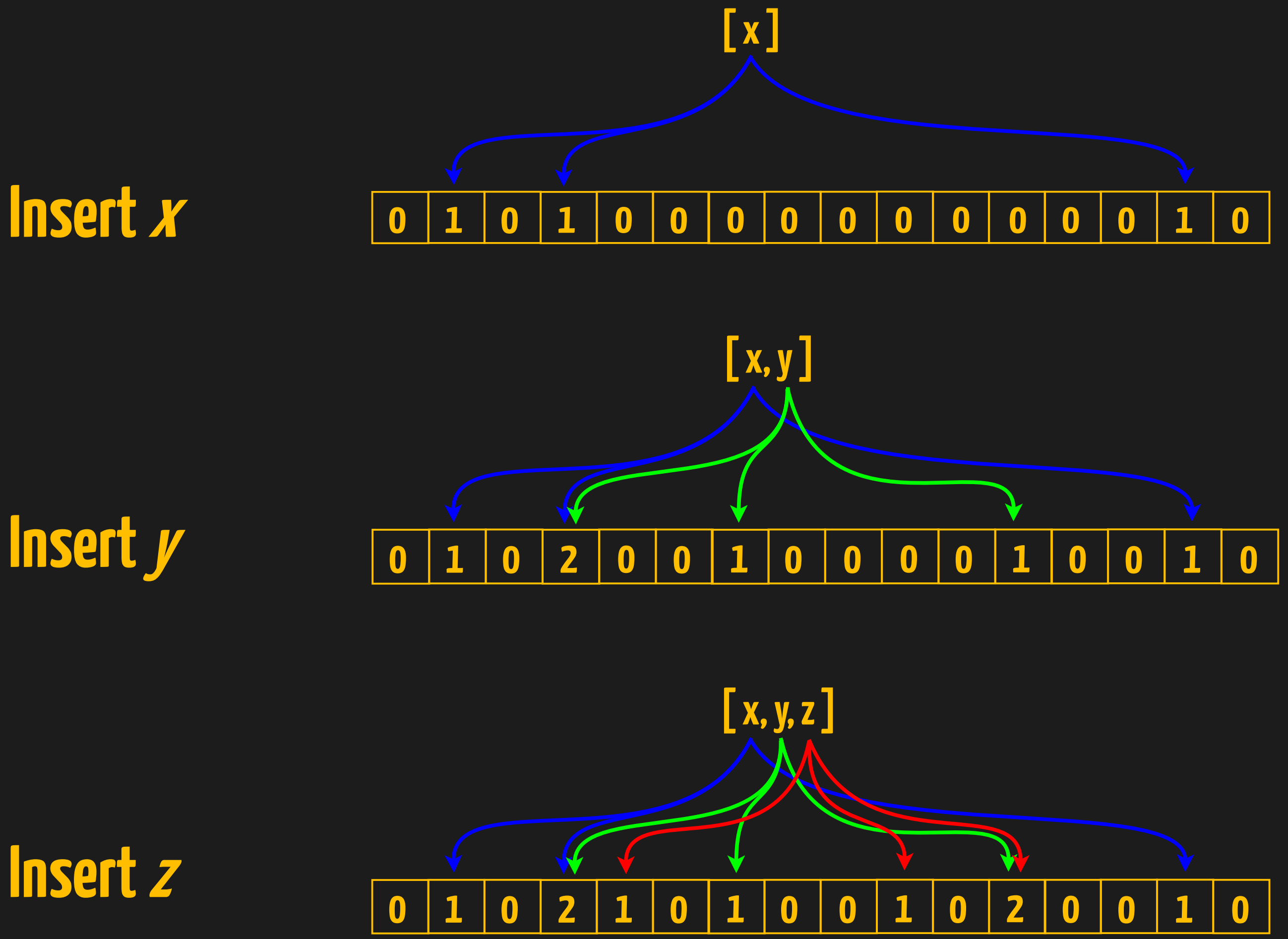
Consists of :

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

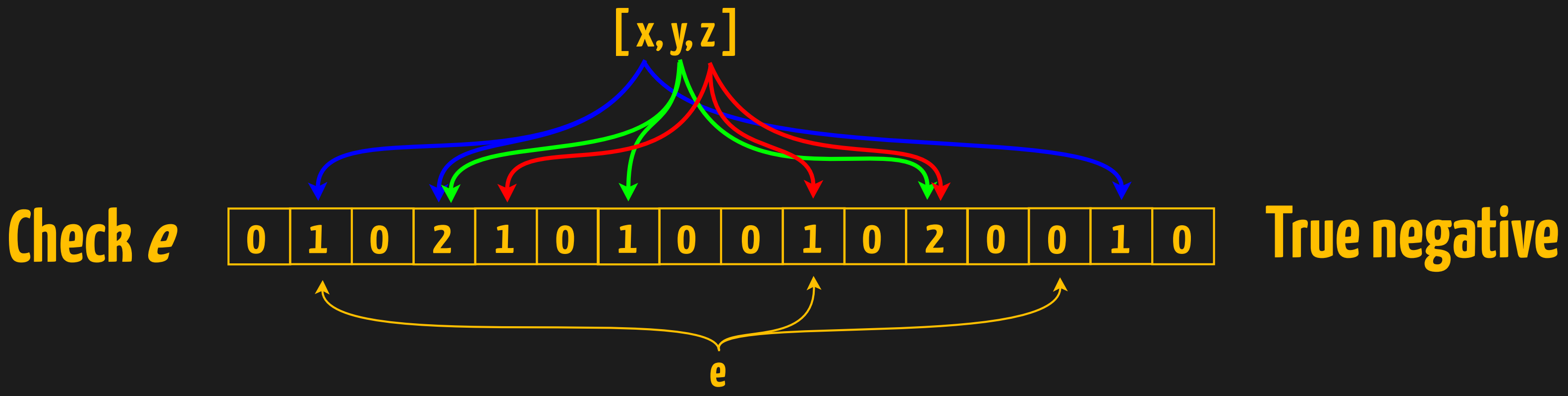
Membership check :

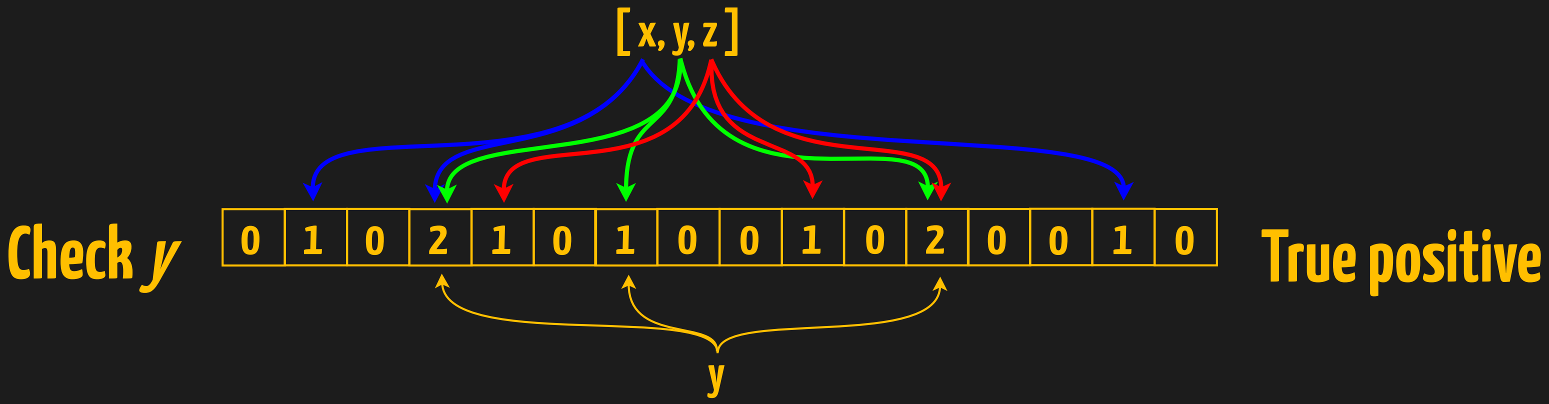
- if all counts are > 0

Counting Bloom Filter

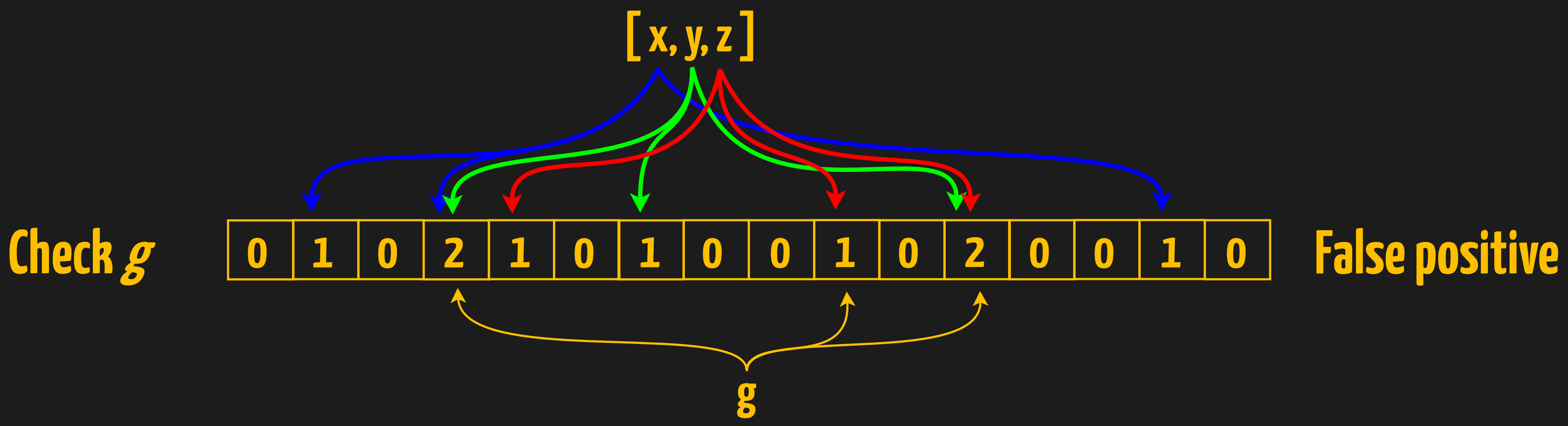


Counting Bloom Filter

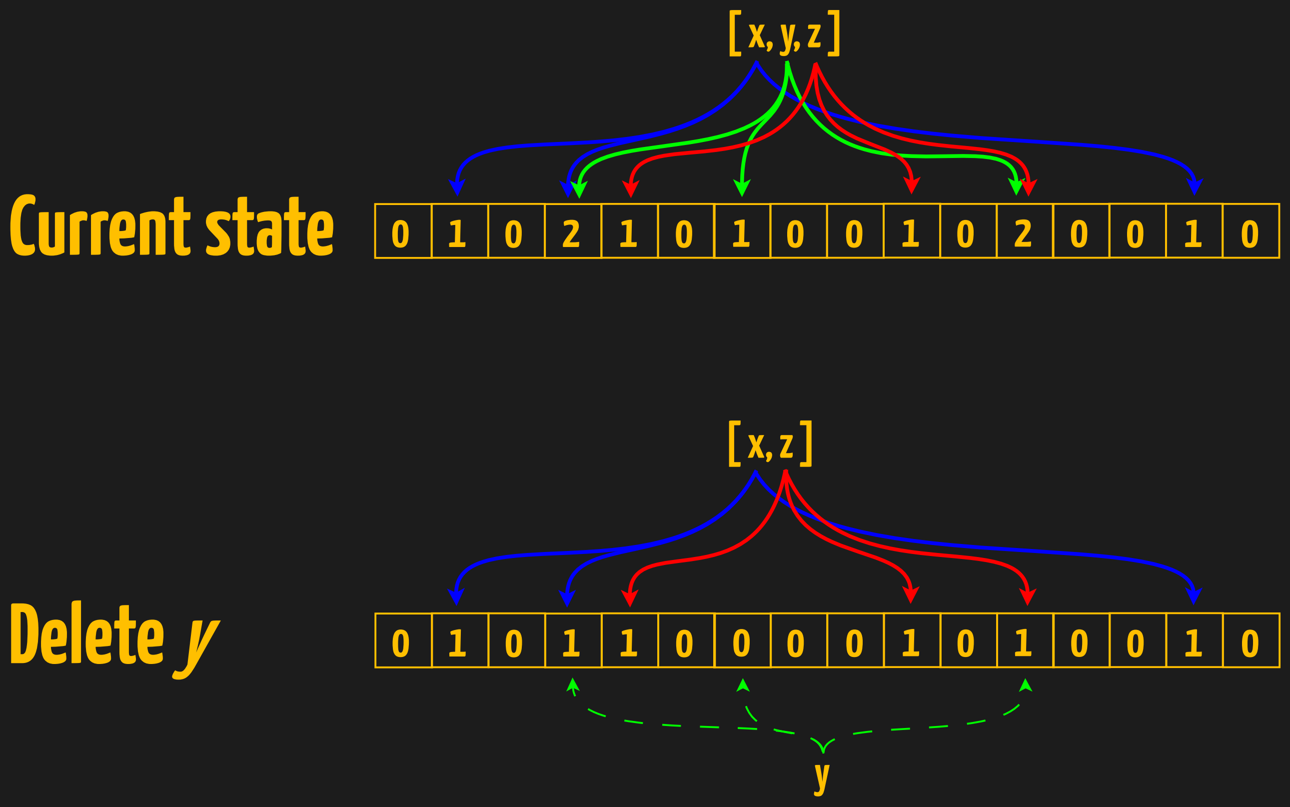




Counting Bloom Filter



Counting Bloom Filter



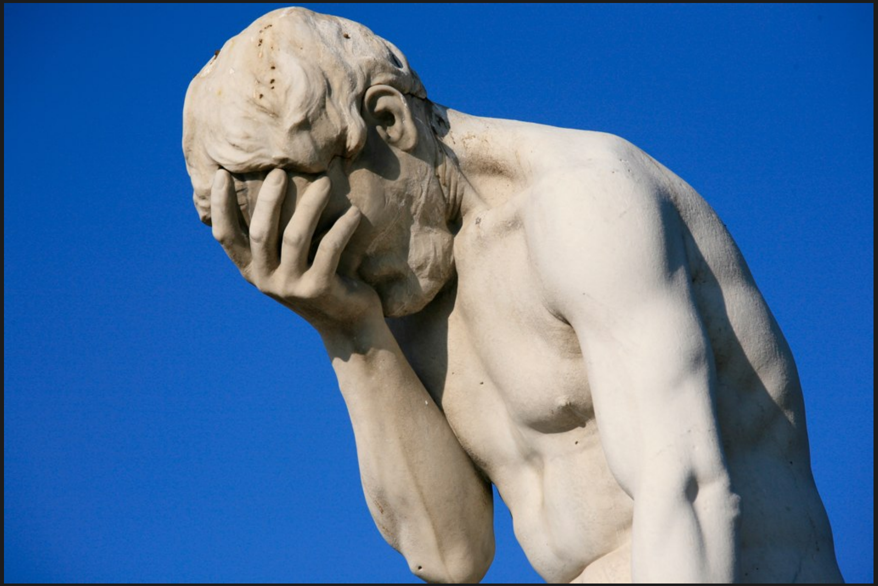
```kotlin
class StringCountingBloomFilter(expectedSize: Int, errorRate: Double) {
  private val m: Int = -(expectedSize * ln(errorRate) / ln2squared).toInt()
  private val k = ceil(-log2(errorRate)).toInt()
  private val counters = ByteArray(m) { Byte.MIN_VALUE }
  private val hashers = IntStream
    .rangeClosed(1, k)
    .mapToObj { n -> Hasher(primes[n + 4], primes[3 * n + 3]) }
    .toList()

 fun add(item: String)=hashers.forEach { counters[abs(it.hashCode(item)) % m]++ }
 fun delete(item: String)=hashers.forEach { counters[abs(it.hashCode(item)) % m]-- }
 fun contains(item: String)=hashers.all { counters[abs(it.hashCode(item)) % m] > MIN_VALUE}

  private class Hasher(private val base: Int, private val mult: Int) {
   fun hashCode(value: String) = value.map { it.code }
     .fold(base) { acc, curr -> acc * mult + curr }
   }
 }
```

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

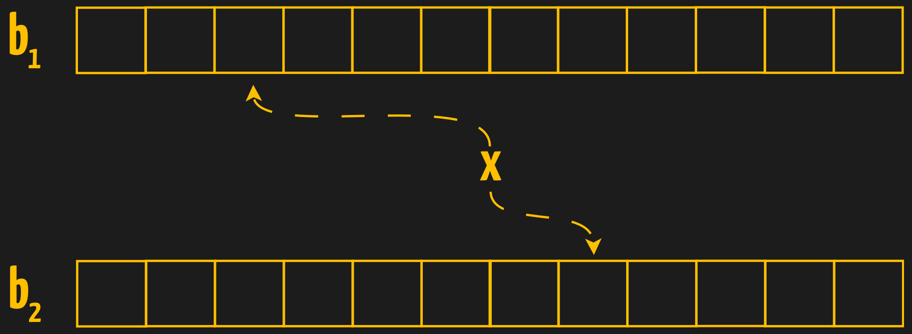
2^{a}) If one or both of the two locations are empty, just insert

2^{b}) Else randomly select one of the two locations, insert and kick out the old item

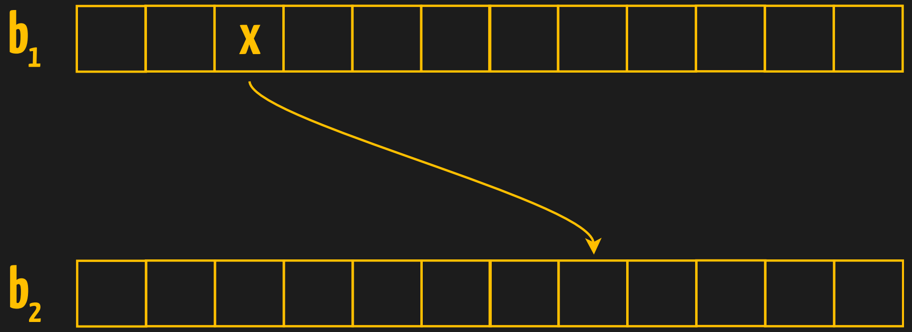
3) Insert the old item into the other location: if not empty, repeat this process

Cuckoo Hashing

Choose location for x

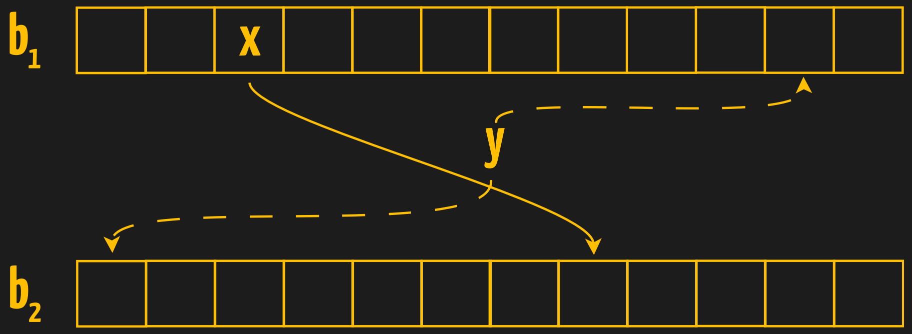


Insert and keep track

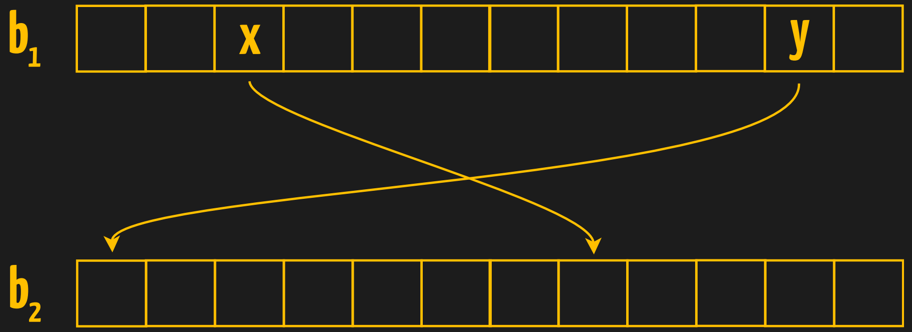


Cuckoo Hashing

Choose location for y

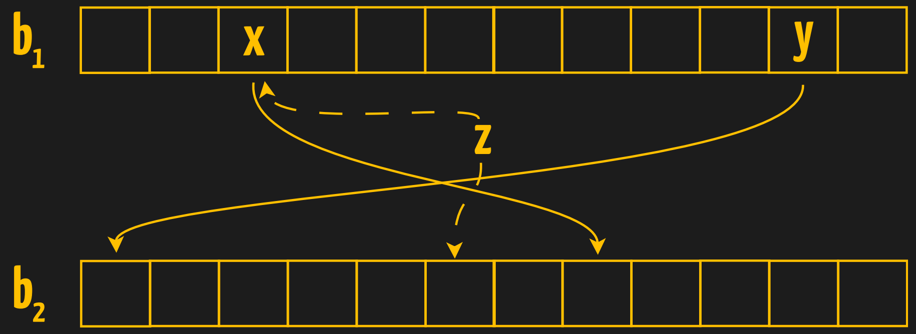


Insert and keep track

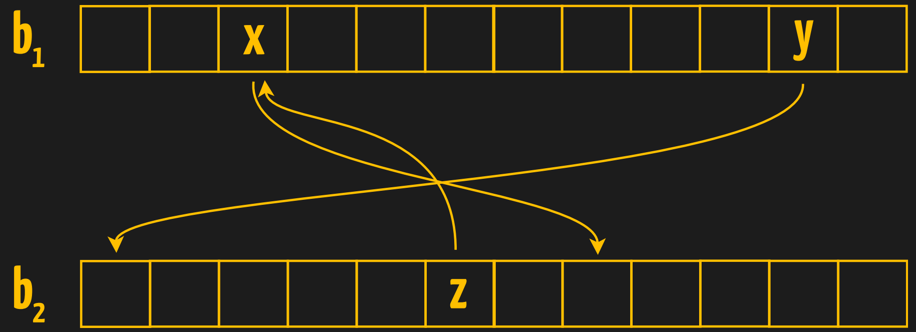


Cuckoo Hashing

Choose location for z

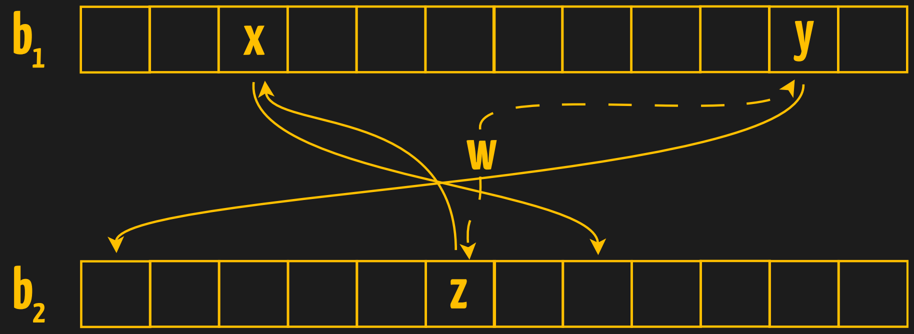


Insert and keep track

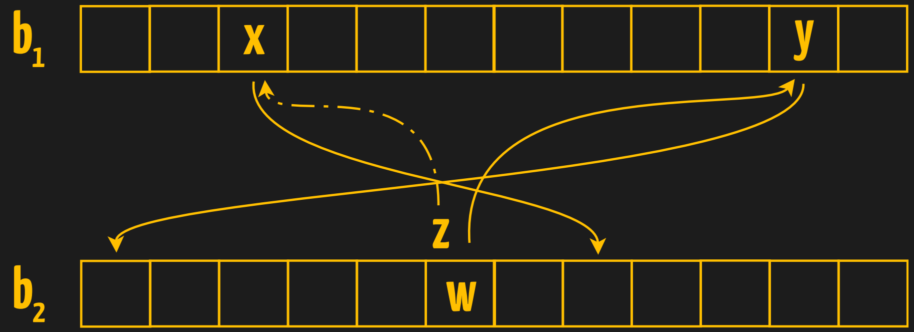


Cuckoo Hashing

Choose location for w

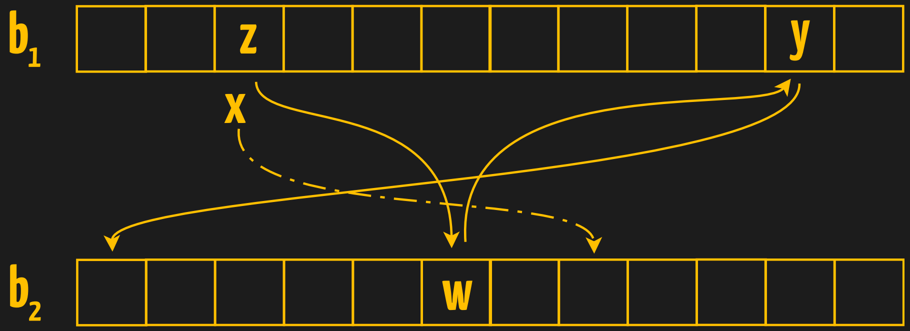


Randomly choose bucket #2 and kick out z!

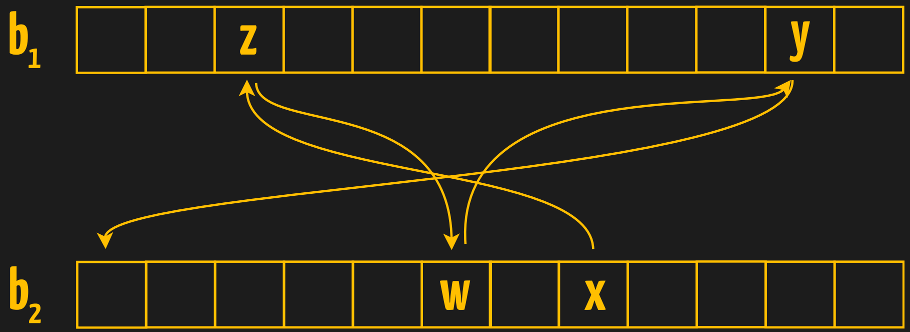


Cuckoo Hashing

z kicks out x



Insert x into its alternative location



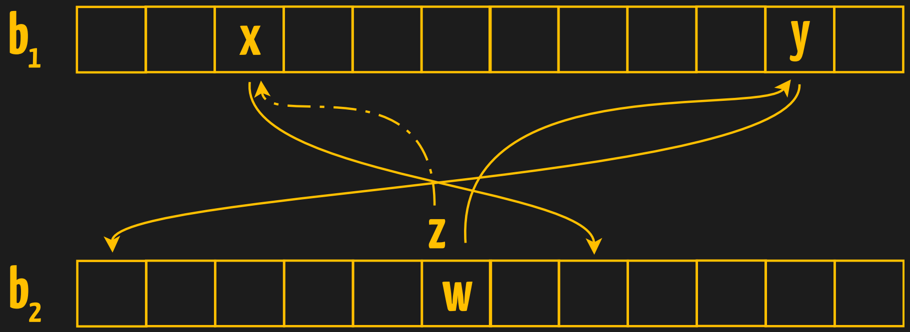
Cuckoo Hashing

- 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
- Incredibily 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 $\begin{cases} h_1(x) = hash(x) \\ h_2(x) = h_1(x) \oplus hash(fingerprint(x)) \end{cases}$

$$\Downarrow$$

$$h_1(x) = h_2(x) \oplus hash(fingerprint(x))$$

Cuckoo Filter

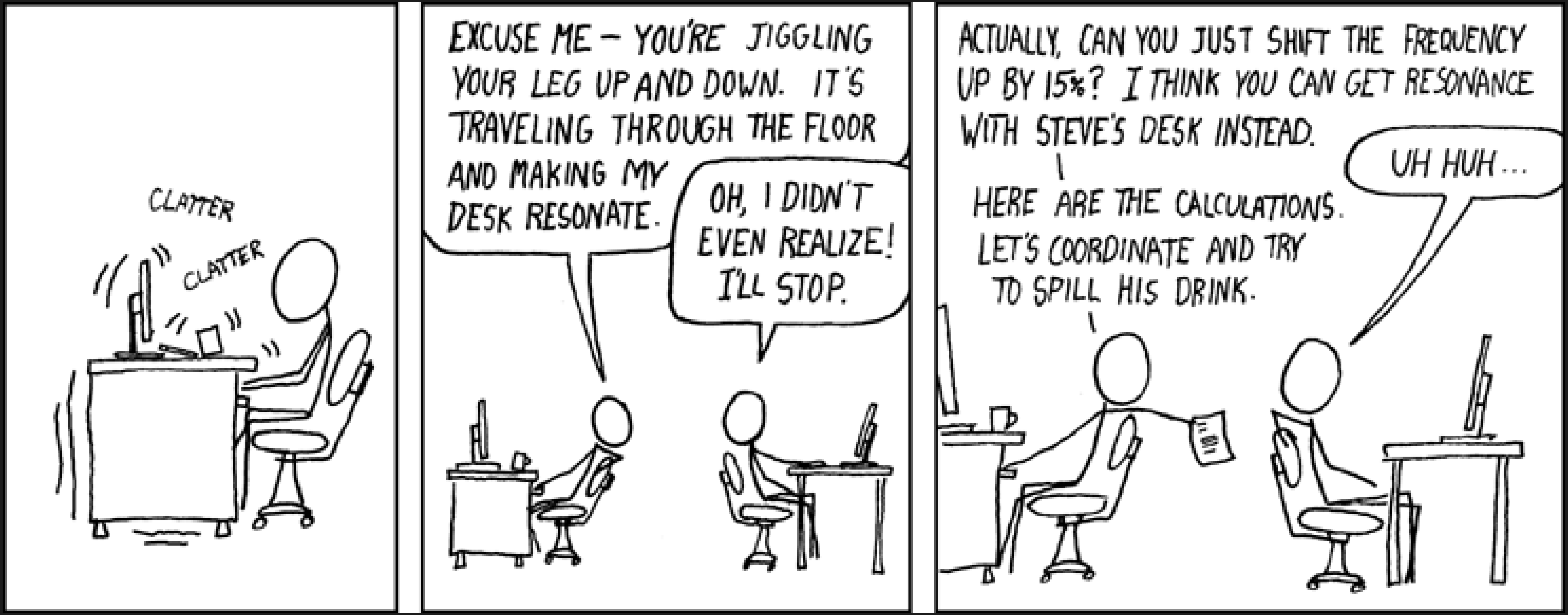
Cuckoo filter requires $\dfrac{\log_2\left(\frac{1}{\varepsilon}\right) + 1 + \log_2(b)}{\alpha}$ bits per key

where α is the load factor of the cuckoo hash table

Cuckoo Filter Code

```kotlin
fun insert(item: String): Boolean {
  var f = fingerprint(item)
  val i1 = hash(item)
  if (bucket[i1] == EMPTY) {
    bucket[i1] = f
    return true
  }
  val i2 = i1 xor hash(f)
  if (bucket[i2] == EMPTY) {
    bucket[i2] = f
    return true
  }

  // relocate existing items
  var i = if (Random.nextBoolean()) i1 else i2
  for (n in 0..<MAX_KICKS) {
    f = bucket[i].also { bucket[i] = f }
    i = i xor hash(f)
    if (bucket[i] == EMPTY) {
      bucket[i] = f
    }
  }
  return false // Max displacement attempts reached
}
```

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)
- array *count* of size *d* x *w* for counting the items
- $d = \ln\left(\frac{1}{\delta}\right)$
- $w = \left(\frac{e}{\varepsilon}\right)$

where ε is the error rate and δ its probability

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

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

Using 32 bit counters, `sizeof(`*count*`)` = $w \cdot d \cdot 4 \simeq 76$ 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	Hash Function #1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Hash Function #2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Hash Function #3
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	Hash Function #4

Insert x

$h_1(x) = 2$	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	Hash Function #1
$h_2(x) = 13$	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	Hash Function #2
$h_3(x) = 12$	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	Hash Function #3
$h_4(x) = 7$	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	Hash Function #4

Count-Min Sketch

$d = 4$, $w = 16$

Insert y

$h_1(y) = 6$	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	Hash Function #1
$h_2(y) = 13$	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	Hash Function #2
$h_3(y) = 8$	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	Hash Function #3
$h_4(y) = 2$	0	0	1	0	0	0	0	1	0	0	0	0	0	0	0	0	Hash Function #4

Insert x

$h_1(x) = 2$	0	0	2	0	0	0	1	0	0	0	0	0	0	0	0	0	Hash Function #1
$h_2(x) = 13$	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	Hash Function #2
$h_3(x) = 12$	0	0	0	0	0	0	0	0	1	0	0	0	2	0	0	0	Hash Function #3
$h_4(x) = 7$	0	0	1	0	0	0	0	2	0	0	0	0	0	0	0	0	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	Hash Function #1
0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	Hash Function #2
0	0	0	0	0	0	0	0	1	0	0	0	2	0	0	0	Hash Function #3
0	0	1	0	0	0	0	2	0	0	0	0	0	0	0	0	Hash Function #4

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

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

Count-Min Sketch

If there are collisions on hashes we get higher counts

Count-Min Sketch Code

```kotlin
class CountMinSketch(d: Int, private val w: Int) {
  private val count = Array(d) { LongArray(w) }
  private val hashers = IntStream
      .rangeClosed(1, h)
      .mapToObj { n -> Hasher(w, Primes.primes[n * 17 + 4], Primes.primes[3 * n + 3]) }
      .toList()

  fun add(item: String) = hashers.forEachIndexed { idx, hasher ->
                            count[idx][hasher.hashCode(item)]++
                           }

  fun count(item: String) = hashers
     .mapIndexed { idx, hasher -> count[idx][hasher.hashCode(item)] }
     .min()

  private class Hasher(val size: Int, val base: Long, val multiplier: Long) {
      fun hashCode(value: String) = value.map { it.code }
         .fold(base) { acc, curr -> acc * multiplier + curr }
          .mod(size)
  }
}
```

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:

- ½ start with “0”
- ½ start with “1”

- ¼ start with “00”
- ¼ start with “01”
- ¼ start with “10”
- ¼ start with “11”

- ⅛ start with “000”
- ⅛ start with “001”
- ⅛ start with “010”
- ⅛ start with “011”
- ⅛ start with “111”
- ⅛ start with “101”
- ⅛ start with “110”
- ⅛ start with “111”

General rule: $p\left(\text{first } k \text{ bits}\right) = 2^{-k}$

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:

- hash function
- array of int

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

$$\text{hash}(x) = \underbrace{\boxed{1}\boxed{1}\boxed{0}\boxed{0}}_{\text{array size } m = 2^4 = 16}\underbrace{\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{1}}_{\text{Remaining bits as the hash value}}$$

HyperLogLog

$$\text{hash}(x) = \underbrace{\boxed{1}\,\boxed{1}\,\boxed{0}\,\boxed{0}}_{\text{Index} = 12}\,\underbrace{\boxed{0}\,\boxed{1}\,\boxed{1}\,\boxed{1}\,\boxed{0}\,\boxed{0}\,\boxed{1}\,\boxed{1}\,\boxed{0}\,\boxed{1}\,\boxed{1}\,\boxed{1}}_{\text{1 leading 0}}$$

array	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

HyperLogLog

$$\text{hash}(y) = \underbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}}_{\text{Index} = 5}\underbrace{\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{0}}_{\text{4 leading 0}}$$

array	0	0	0	0	0	4	0	0	0	0	0	0	1	0	0	0

HyperLogLog

$$\text{hash}(z) = \underbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}}_{\text{Index} = 5}\underbrace{\boxed{0}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}}_{\text{1 leading 0}}$$

array	0	0	0	0	0	4	0	0	0	0	0	0	1	0	0	0

HyperLogLog

Get the count:

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

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

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

$m = 16$ (4 bits) $\rightarrow$ 0.065%

HyperLogLog Code

```kotlin
class HyperLogLog {
  private val p: Int = 4
  private val m: Int = 2.0.pow(p.toDouble()).toInt()
  private val alpha = 0.673
  private val buckets = IntArray(m)

  fun add(item: String) {
      val hash = item.hashCode()
      val index = hash.ushr(32 - p)
      val leadingZeroes = Integer.numberOfLeadingZeros(hash shl p) + 1
      buckets[index] = max(buckets[index], leadingZeroes)
  }

  fun count(): Long {
      val harmonicMean = buckets.sumOf { 1.0 / (1 shl it) }
      val estimate = alpha * (m * m).toDouble() / harmonicMean
      return if (estimate ≤ 5.0 / 2.0 * m) {
          (m * ln(m.toDouble() / estimate)).toLong()
      } else {
          estimate.toLong()
      }
  }
}
```

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?