Download this presentation here:

Intro to probabilistic data structures

About Me

- Software Engineer @ AssemblyAI
- Interest in Algorithms & DS and AI

Agenda

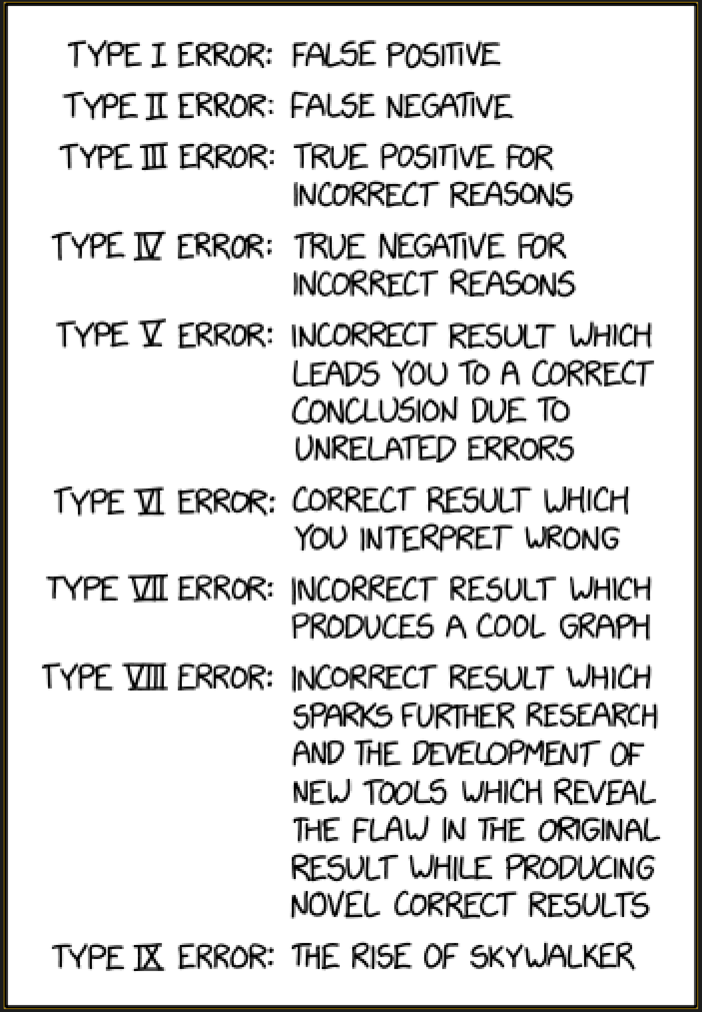
- membership
- cardinality
- frequency

In total we'll see five PDS

Probabilistic Data Structures return approximate values:

- Approximate values in case of numbers
- False positives in case of booleans

but they save a lot of memory



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

Membership

Checking if an item belongs to a set

Membership use cases

In general when a wrong answer does not involve correctness but more work

Some examples:

- Taken username
- Ad placement: has the user already seen this ad?
- Fraud detection (has the user paid from this location before?)

Membership

- Bloom Filter
- Counting Bloom Filter
- Cuckoo Filter

Bloom Filter

Invented by Burton H. Bloom in 1970

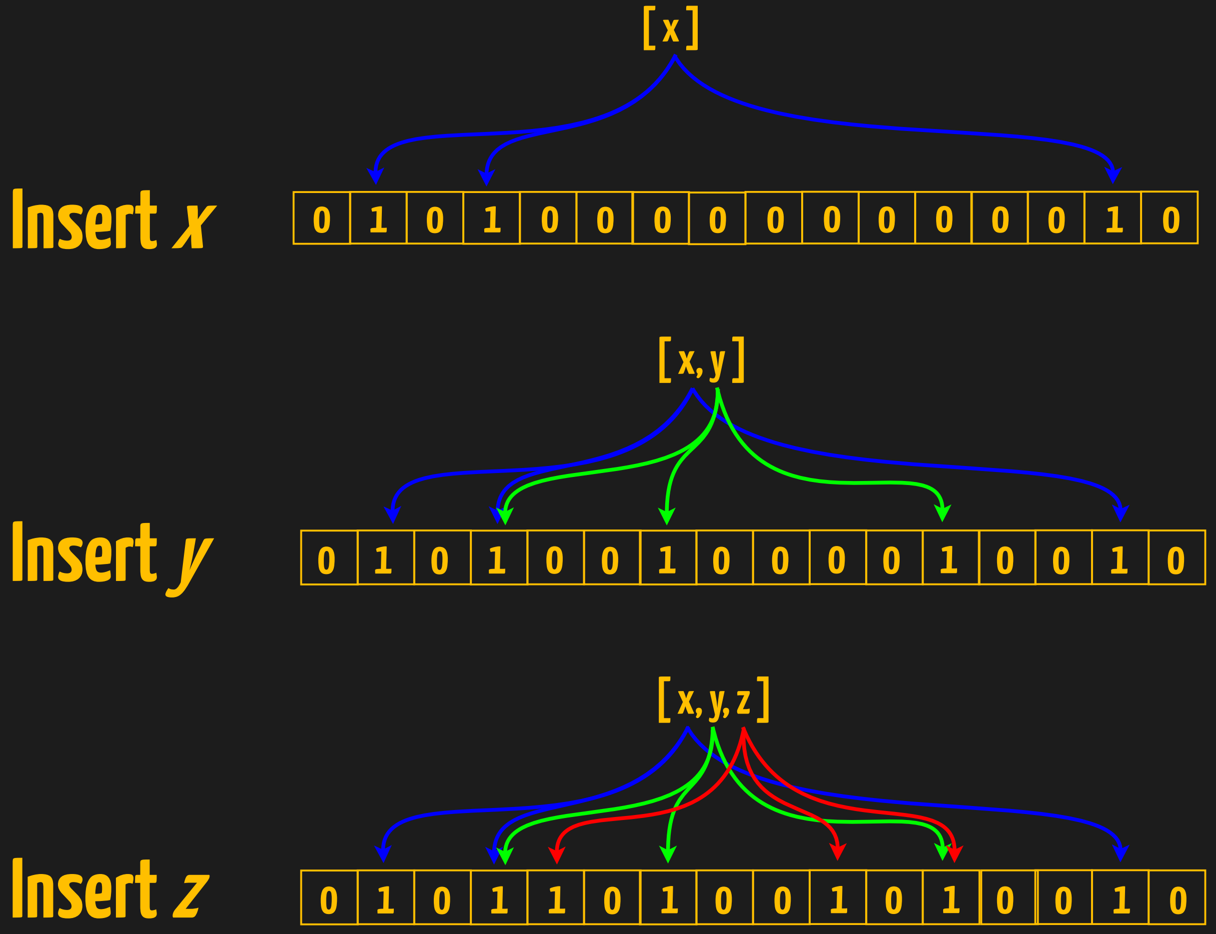
Consists of :

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

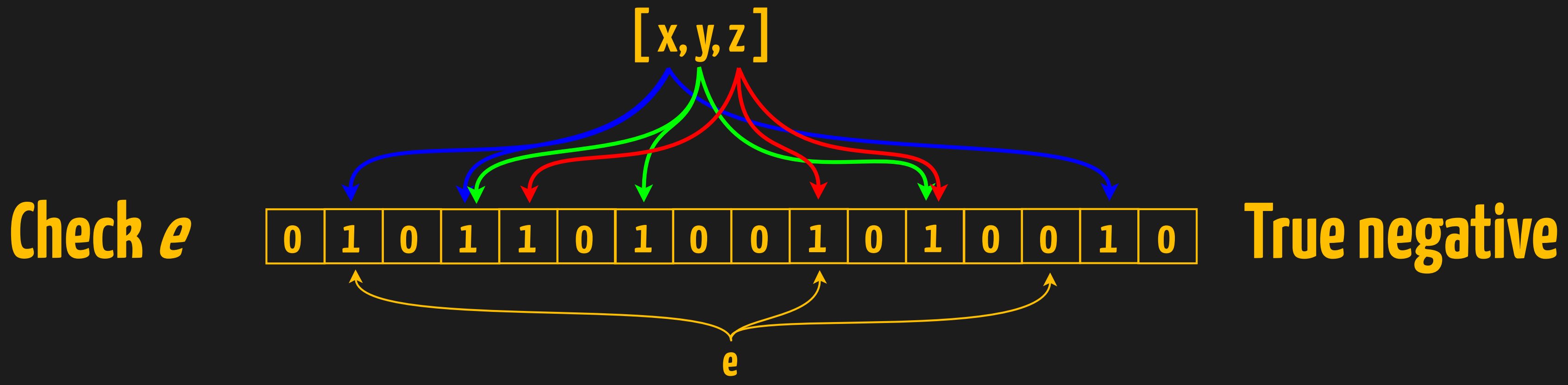
Membership condition :

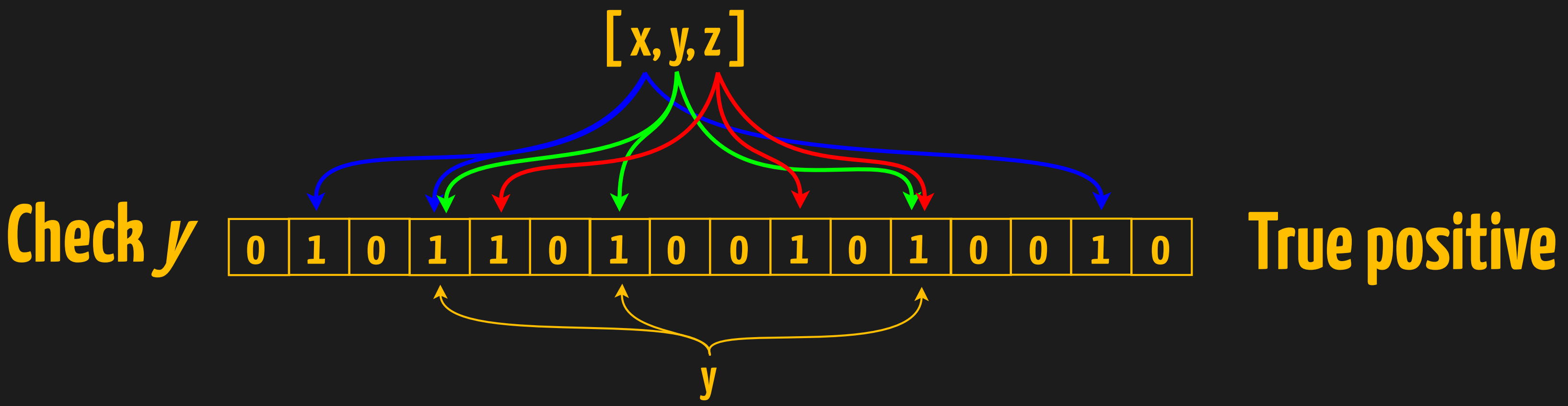
- if all bits are set

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



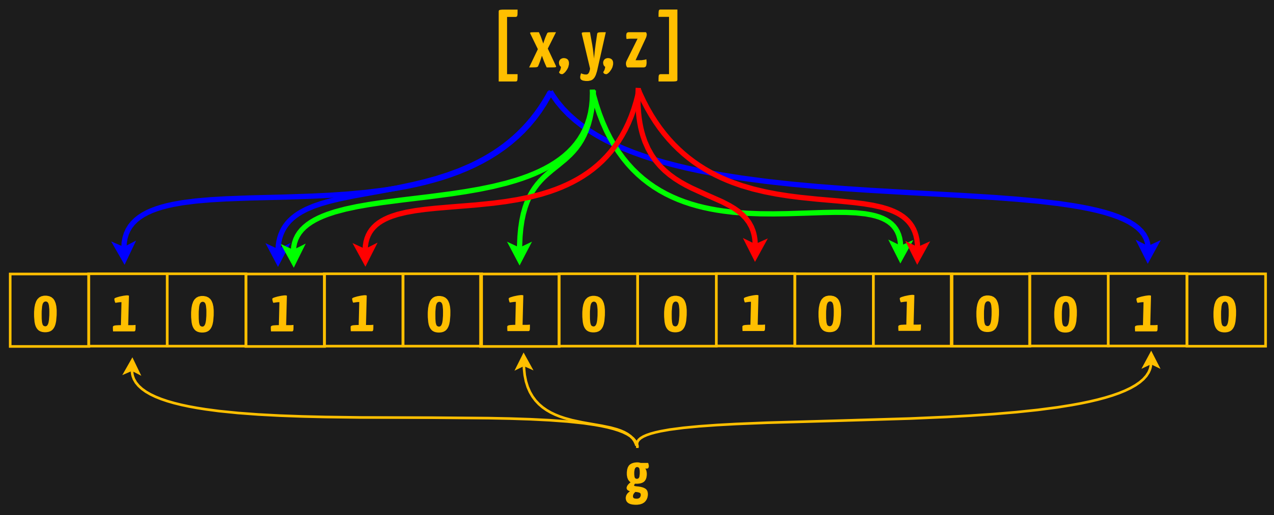
Bloom Filter with $m=16, k=3$





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

Check *g*



False positive

Bloom Filter

Approximate optimal size

Given:

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

Then:

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

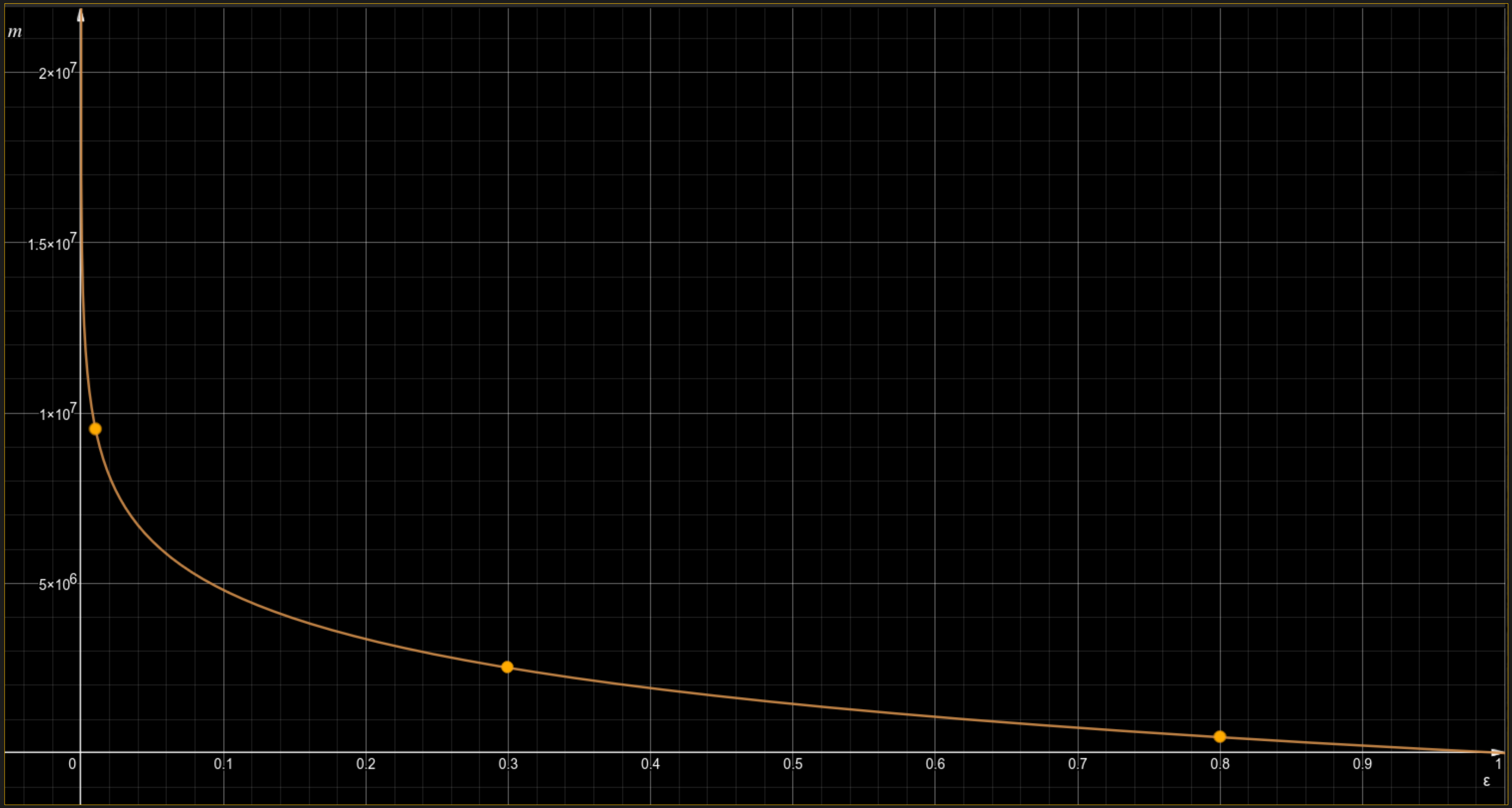
To check 1M items, with 1% error rate:

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

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

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

It adds two additional features:

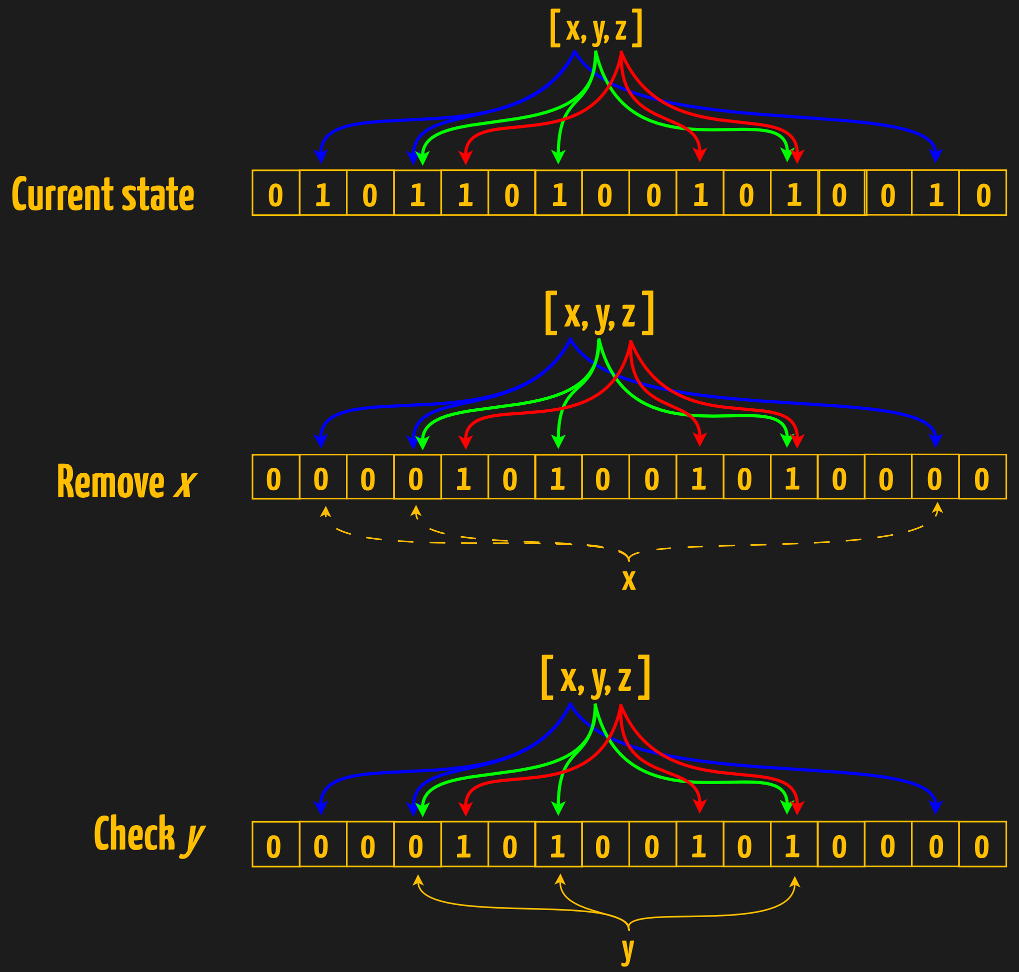
- based on similarity to previously experienced odors
- time elapsed since the odor was last experienced

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

Bloom Filter Code

```
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 = (1..k).map { n -> Hasher(primes[n + 4], primes[3 * n + 3]) }

  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 }
      }
}
```

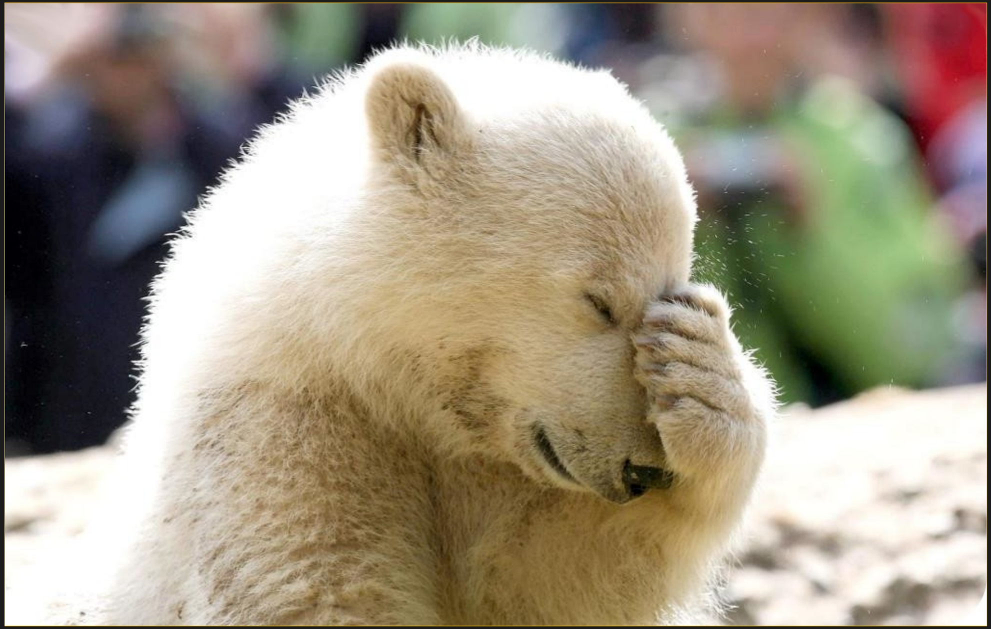
What if we remove an element?



False negative!

Bloom Filter

What's the use of a DS that may return both false positives and false negatives?



Counting Bloom Filter

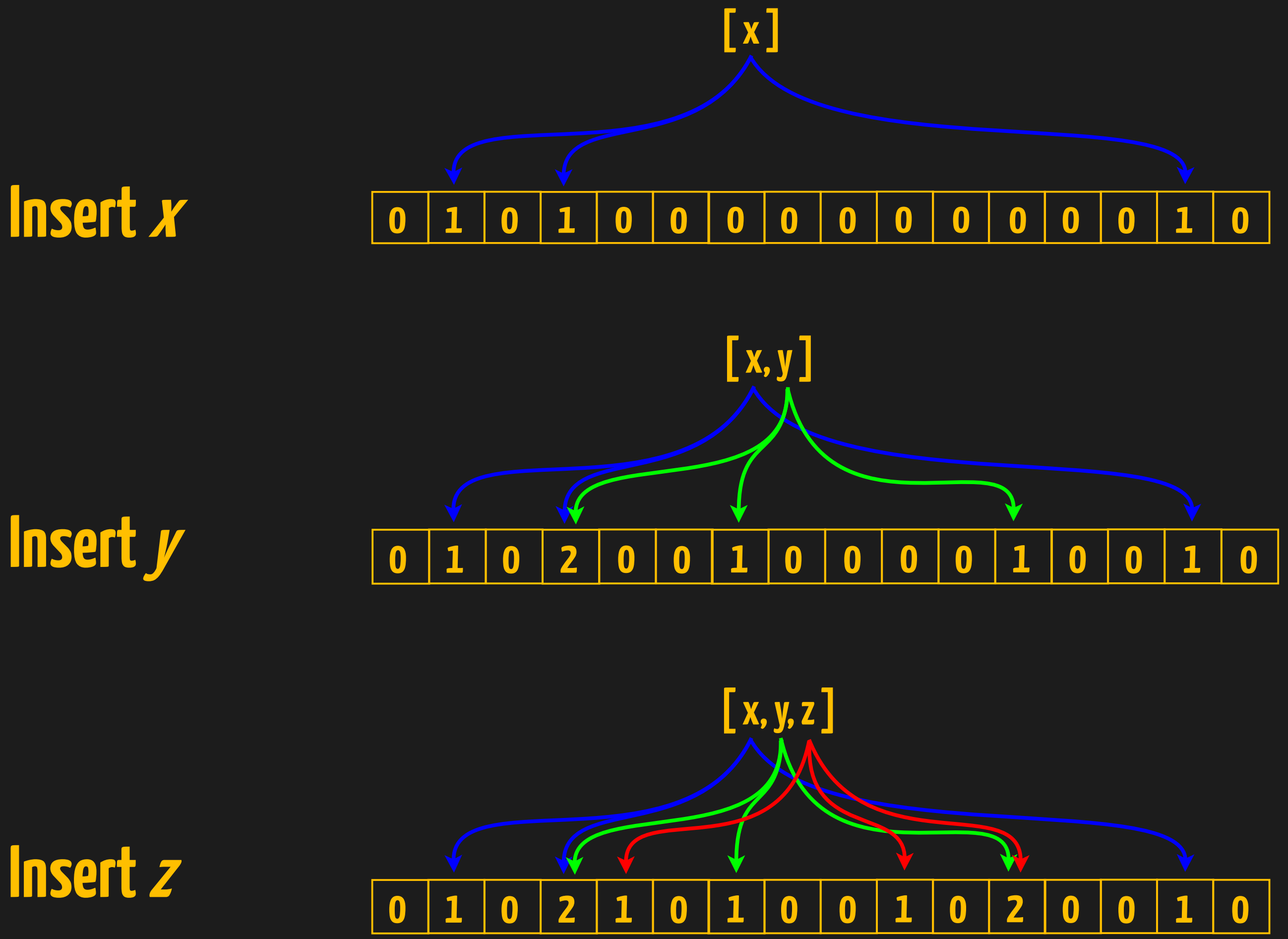
Consists of :

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

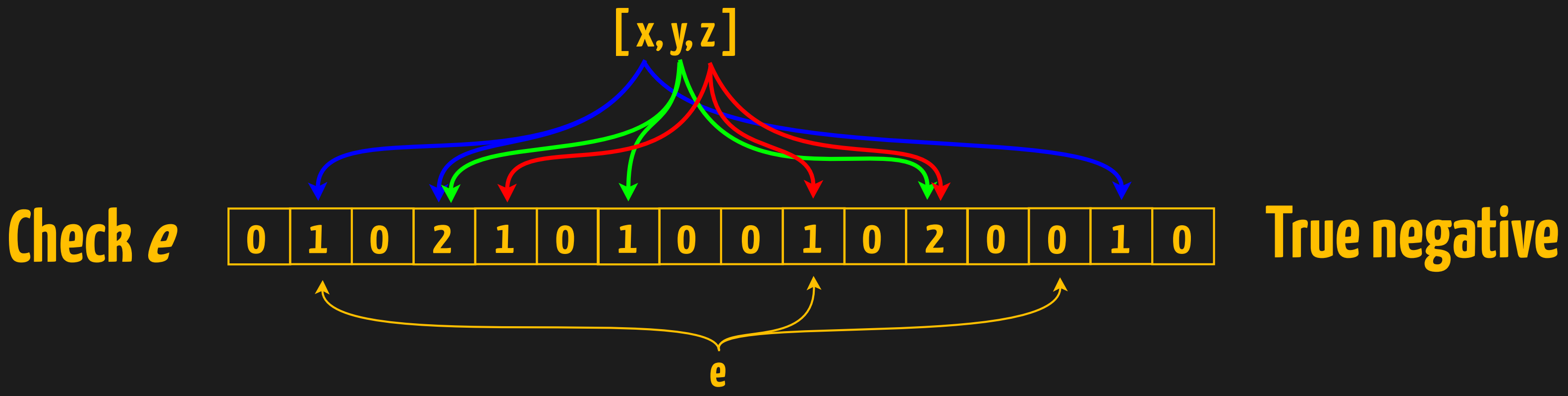
Membership condition :

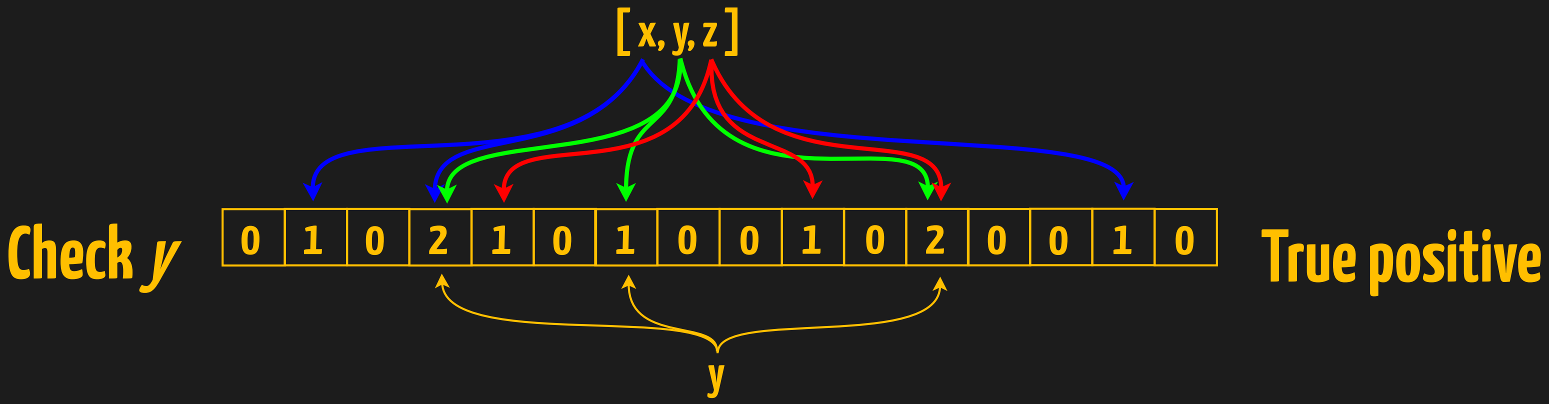
- if all counts are > 0

Counting Bloom Filter

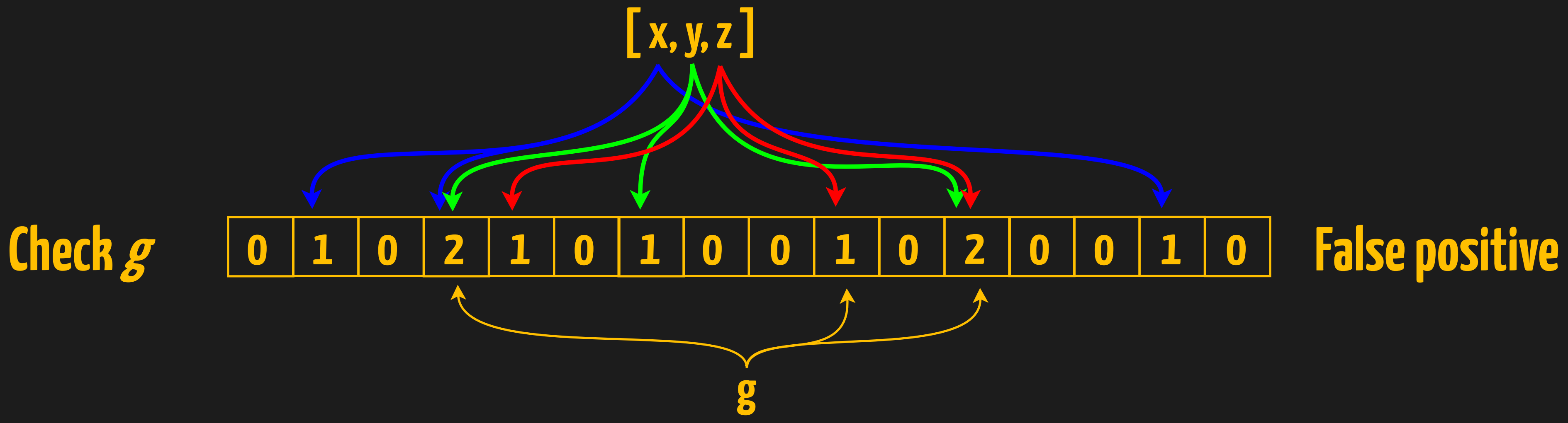


Counting Bloom Filter

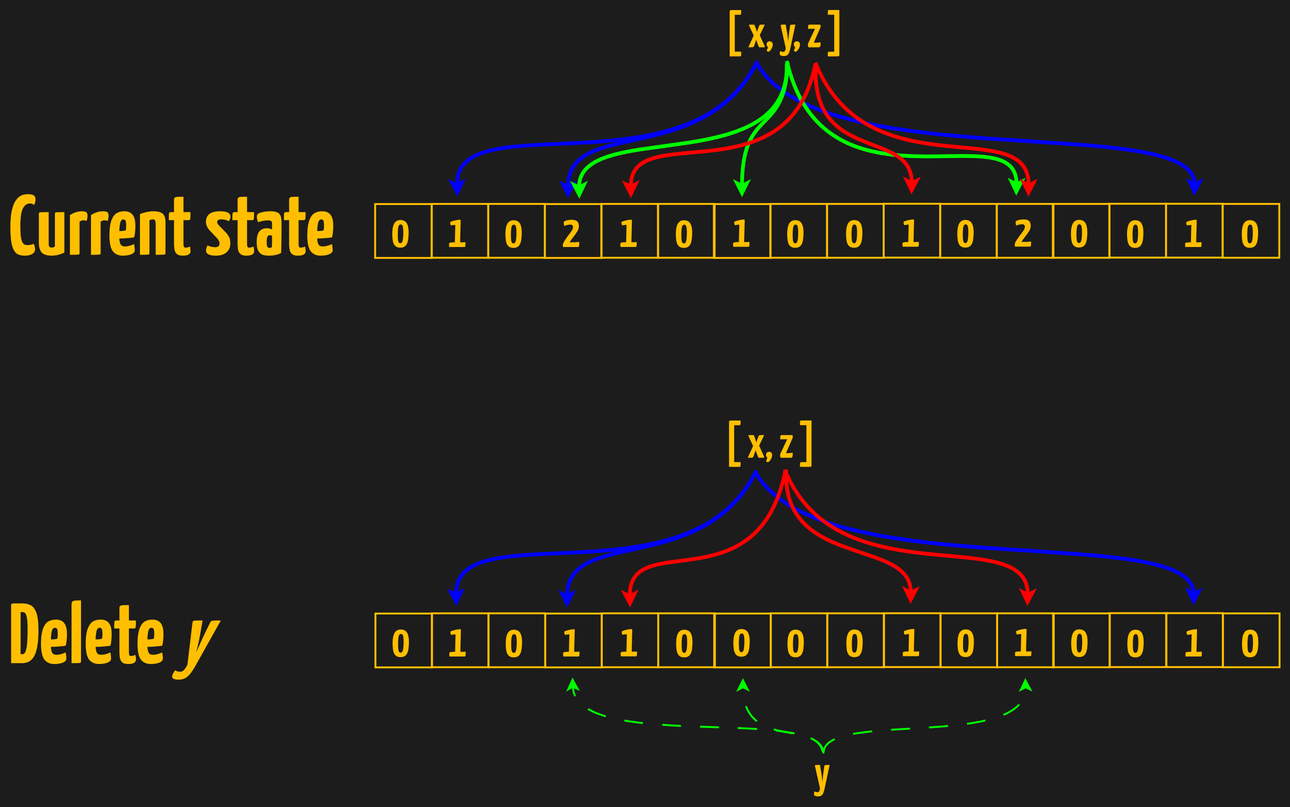




Counting Bloom Filter



Counting Bloom Filter



Counting Bloom Filter Code

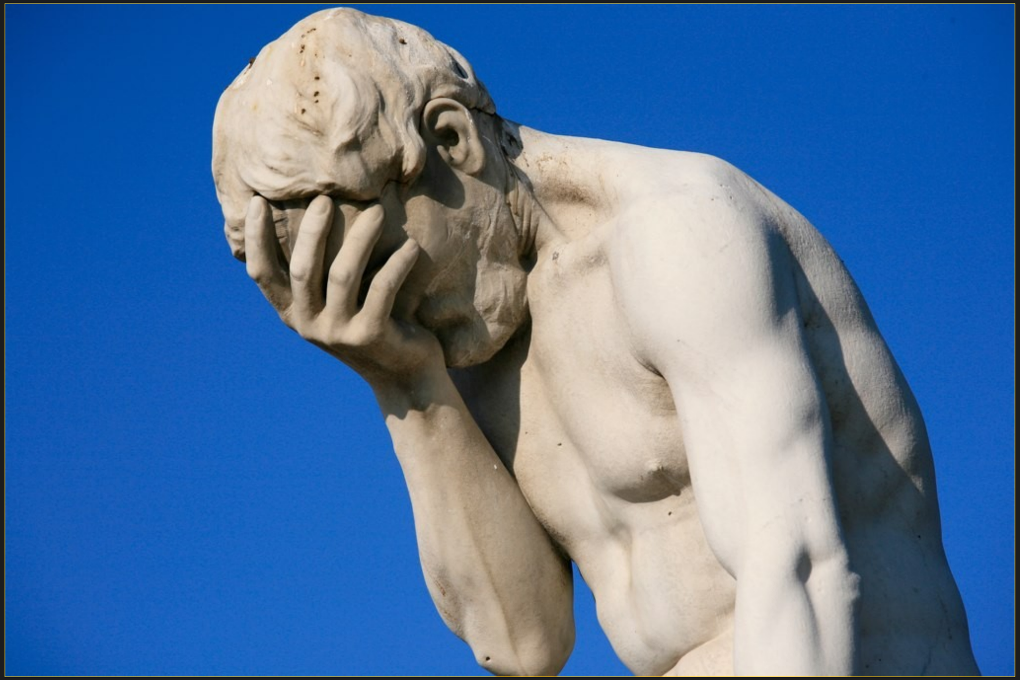
```
class StringCountingBloomFilter(expectedSize: Int, errorRate: Double) {
  private val m: Int = -(expectedSize * ln(errorRate) / ln2squared).toInt()
  private val k = ceil(-log2(errorRate)).toInt()
  private val counters = ByteArray(m) { Byte.MIN_VALUE }
  private val hashers = (1..k).map { n -> Hasher(primes[n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

  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

It uses a lot more memory than Bloom Filter:

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

It also might overflow



Cuckoo Filter

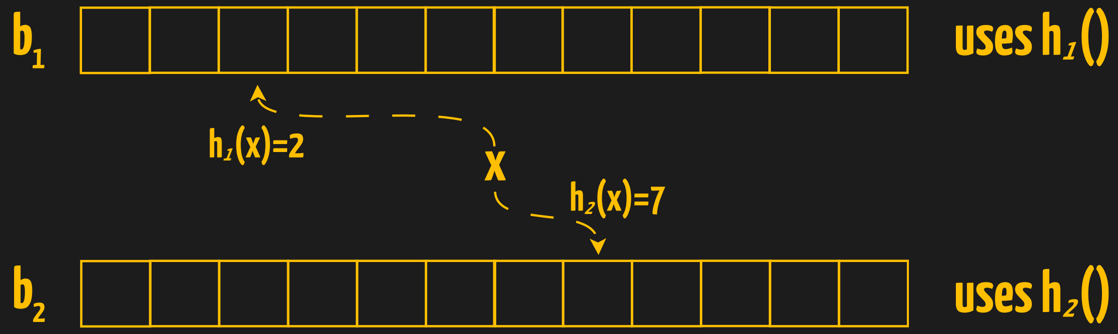
Described by Fan, Andersen, Kaminsky, and Mitzenmacher in 2014

Based on Cuckoo Hashing (a *classic* Data Structure), which consists of :

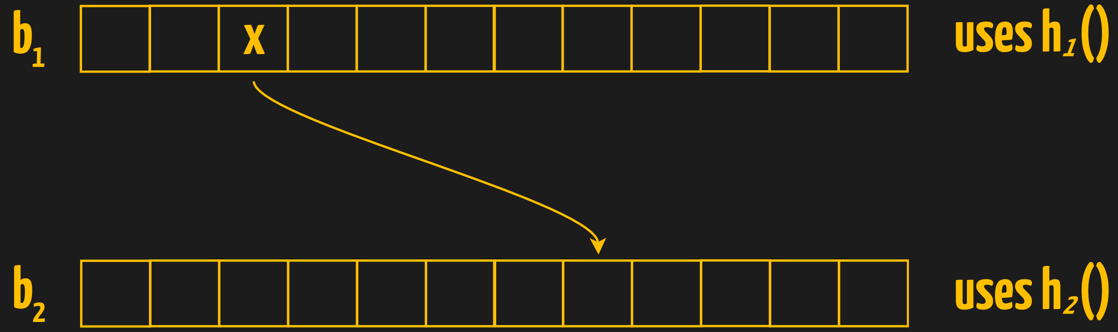
- 2 arrays: b_1 and b_2
- 2 hash functions: $h_1()$ and $h_2()$

Cuckoo Hashing

Choose location for x

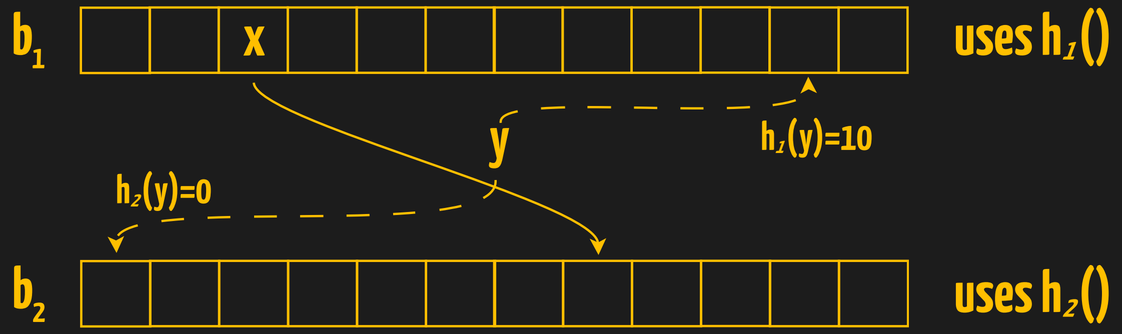


Insert

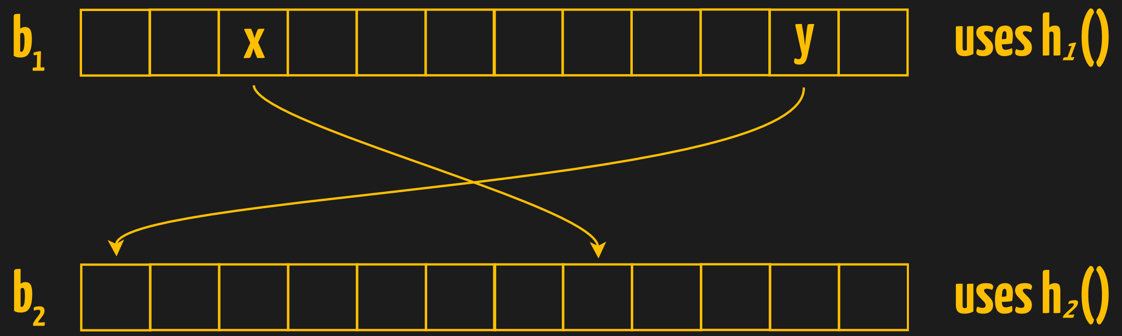


Cuckoo Hashing

Choose location for y

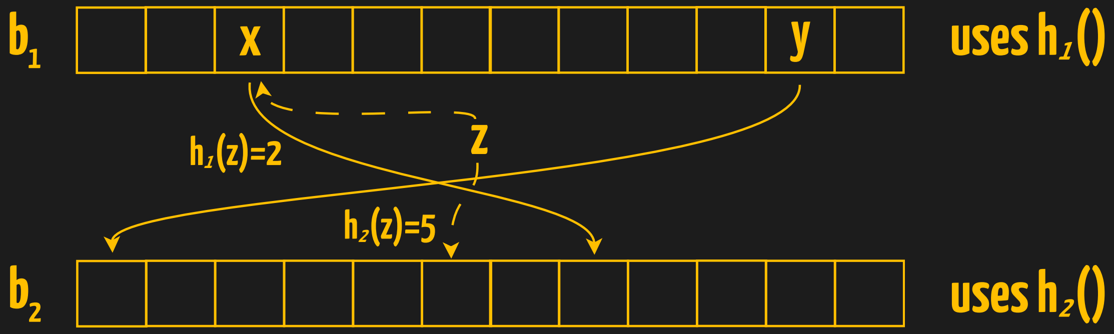


Insert

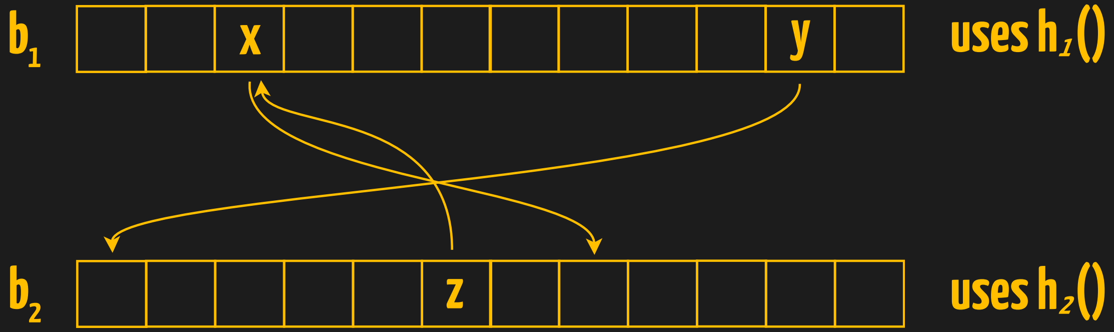


Cuckoo Hashing

Choose location for z

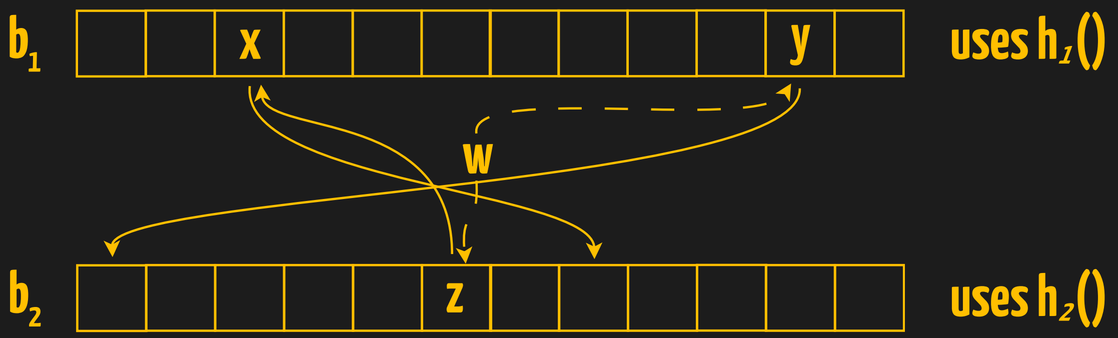


Insert

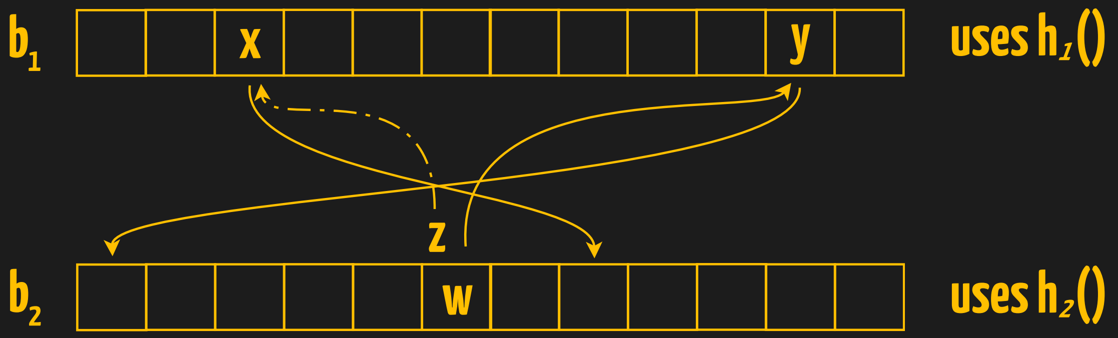


Cuckoo Hashing

Choose location for w

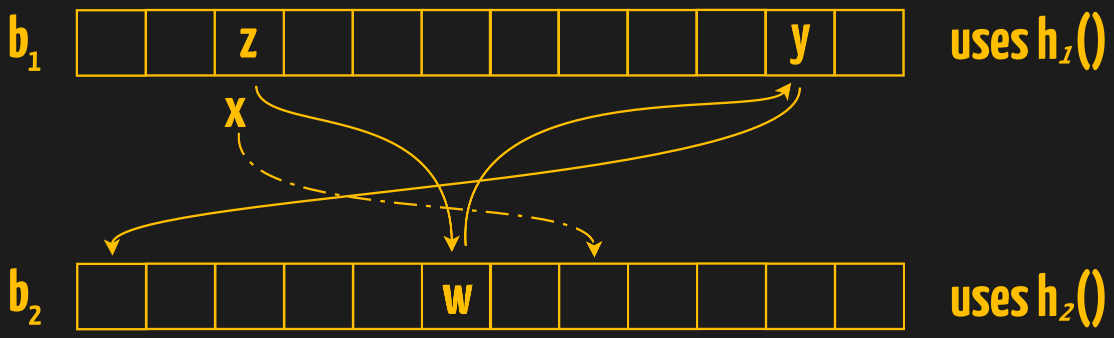


Randomly choose b_2 and kick out z!

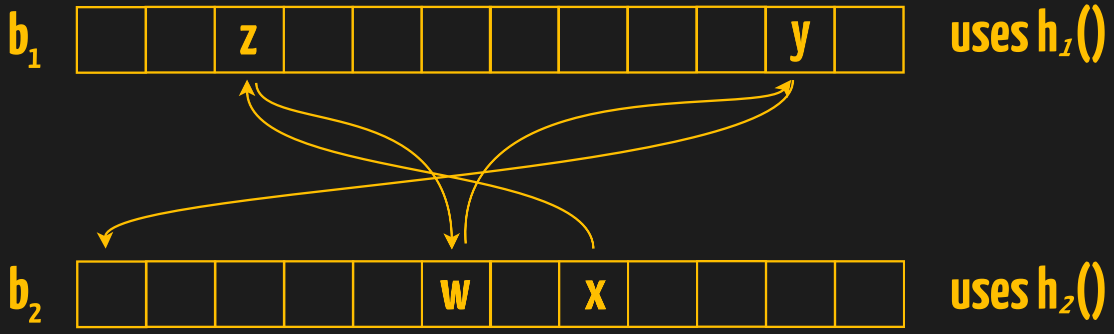


Cuckoo Hashing

z kicks out x



Insert x into its alternate location



Cuckoo Hashing

Performance

- O(1) lookup runtime complexity
- O(1) delete runtime complexity
- Insert: if it fails (cycle or threshold) the two hash tables need to be enlarged and rehashed
- O(1) insert amortized runtime complexity

Cuckoo Filter

- 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?

Solution: create a hash that keeps track of fingerprint

Cuckoo Filter

- a fingerprint function returning a f-bit value
- two hash functions $\begin{cases} h_1(x) = hash(x) \\ h_2(x) = h_1(x) \oplus hash(fingerprint(x)) \end{cases}$

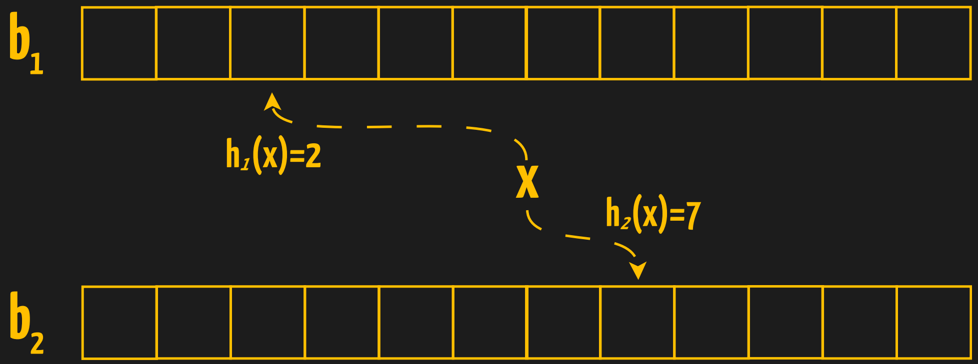
$$\Downarrow$$

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

(thanks to XOR simmetry)

Cuckoo Filter

Choose location for x

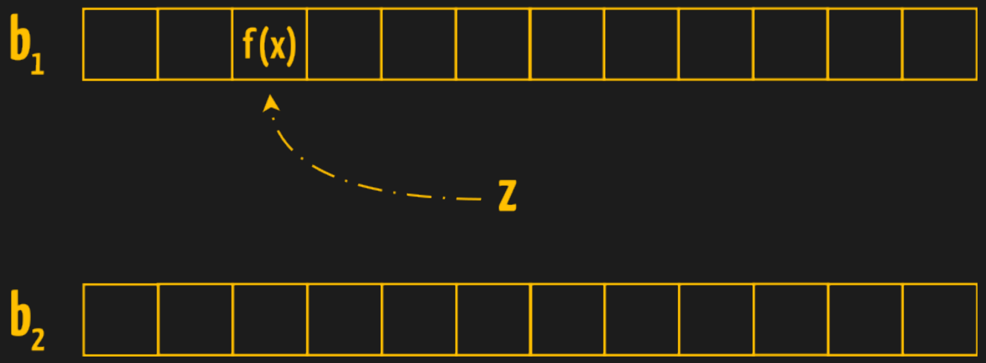


Insert fingerprint(x)

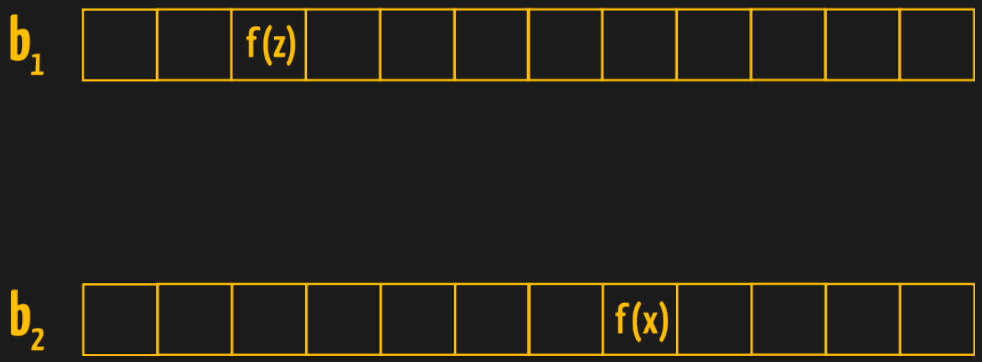
Cuckoo Filter

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

x is kicked out by *z*



Move fingerprint(x) and insert fingerprint(z)



Cuckoo Filter

Lookup of *x*

$$f(x) == b_1[h_1(x)] \quad || \quad f(x) == b_2[h_2(x)]$$

When a matching fingerprint is found in any of the *b* arrays, the entry might be in the filter. We can have false positives when a different entry has the same fingerprint as the searched one.

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.

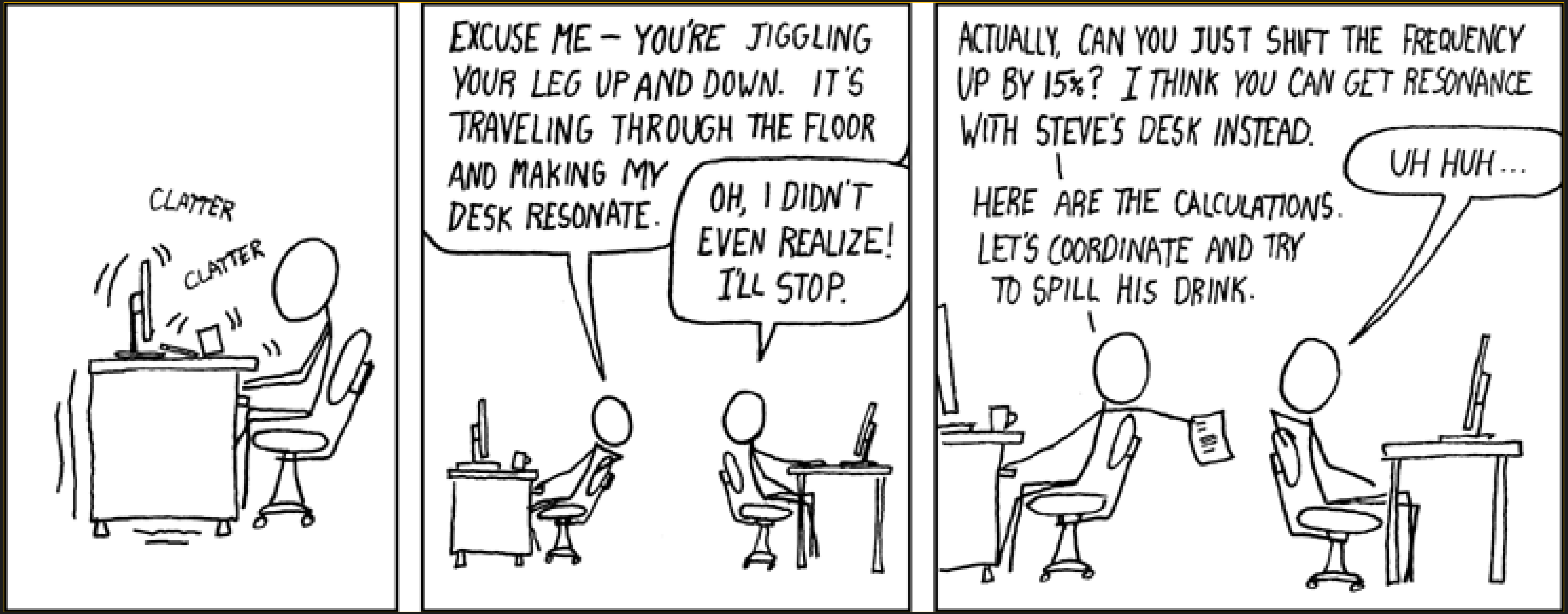
Cuckoo Filter Code

```
fun insert(item: String): Boolean {
  var f = fingerprint(item)
  val index1 = hash(item)
  if (bucket[index1] == EMPTY) {
    bucket[index1] = f
    return true
  }
  val index2 = index1 xor hash(f)
  if (bucket[index2] == EMPTY) {
    bucket[index2] = f
    return true
  }

  // relocate existing items
  var i = if (Random.nextBoolean()) index1 else index2
  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 kicks reached
}
```

Membership Libraries

- Guava: https://github.com/google/guava (Bloom Filter)
- Hadoop https://hadoop.apache.org/ (Bloom and Counting Bloom Filters)
- Fastfilter https://github.com/FastFilter/fastfilter_java (Bloom, Counting Bloom, Cuckoo and other filters)
- Boom Filters: https://github.com/tylertreat/BoomFilters (Bloom, Counting Bloom, Cuckoo and other filters)

Frequency



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

Frequency

Counting the number of times an item appears in a stream

- Count-Min Sketch

Frequency Use Cases

Some examples:

- 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)
- a bidimensional array *count* of size *d* x *w* for counting the items, where:

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

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

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

Count-Min Sketch

$d = 4$, $w = 16$

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

Count-Min Sketch

d = 4, *w* = 16

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 = (1..k).map { n -> Hasher(primes[n + 4], primes[3 * n + 3]) }

  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)
  }
}
```

Frequency Libraries

- Apache Spark: https://spark.apache.org/
- Boom Filters: https://github.com/tylertreat/BoomFilters

Cardinality

Cardinality

Counting the number of distinct items in a collection

- HyperLogLog

Cardinality Use Cases

Some examples:

- **Google BigQuery:** `APPROX_COUNT_DISTINCT`, `APPROX_QUANTILES`, `APPROX_TOP_COUNT`, `APPROX_TOP_SUM`
- **Snowflake:** `HLL`, `HLL_ACCUMULATE`, `HLL_COMBINE`, `HLL_ESTIMATE`
- 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:

- a hash function
- an array of int M

With only one hash function, we risk that one single item can skew the result, so we could use multiple hash functions.

HyperLogLog

Insert x

$h_1(x) = 0000100100101001$

$h_2(x) = 0011100110011010$

$h_3(x) = 0101011110100011$

$h_4(x) = 1111100111101001$

$h_5(x) = 0010110110000110$

...

$h_{16}(x) = 0001100001001100$

Array M

4	2	1	0	2	1	0	0	3	0	2	1	1	0	1	3

HyperLogLog

Insert y

$h_1(y) = 1100010101110101$

$h_2(y) = 0101110010010101$

$h_3(y) = 0001100010101010$

$h_4(y) = 0101010100111101$

$h_5(y) = 0111001100110100$

...

$h_{16}(y) = 0110011010100101$

Array M

4	2	3	1	2	1	0	0	3	0	2	1	1	0	1	3

HyperLogLog

Computing multiple hash functions is computationally expensive!
Can we do it with only one function?

We can split the outcome of one hash function in two and use one part as the index of the array and the other as the value.

$$\mathrm{hash}(x) = \underbrace{\boxed{1}\boxed{1}\boxed{0}\boxed{0}}_{\text{array size } m\, =\, 2^4 = 16}\underbrace{\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{0}\boxed{1}\boxed{1}\boxed{1}}_{\text{Remaining 12 bits as the hash value}}$$

HyperLogLog

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

Array M																
	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0

HyperLogLog

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

Array M

0	0	0	0	0	4	0	0	0	0	0	0	1	0	0	0

HyperLogLog

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

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 Z with a correction factor α : cardinality $= Z \cdot |M|^2 \cdot \alpha_m$

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

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

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

```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()
        }
    }
}
```

Cardinality Libraries

- Apache DataSketches: https://datasketches.apache.org
- Boom Filters: https://github.com/tylertreat/BoomFilters

Probabilistic Data Structures

Pros: - Save huge amount of memory

Cons: - Uncertainty
- Cold start problem: need snapshots
- Distributed?

Original Papers

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

Code shown in this presentation: https://github.com/andreaiacono/TalkProbabilisticDataStructures

Slides of this presentation:

Questions?