Bloom Filters

Thomas Schwarz, SJ

Existence Data Structure

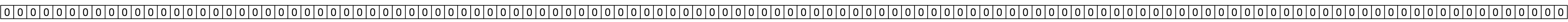
- Hyphenation rules for word-processing
  - About 90% of all words are separated using very simple rules
  - About 10% of all words have their hyphenation pattern stored in a dictionary
  - Goal: Make the common task simple
    - Determine quickly and accurately if a word is ***not*** in the dictionary

Bloom Filters

- A ***probabilistic*** data structure
  - Always correct when saying that a word is ***not*** in the dictionary
  - Almost always correct when it says that a word is in the dictionary

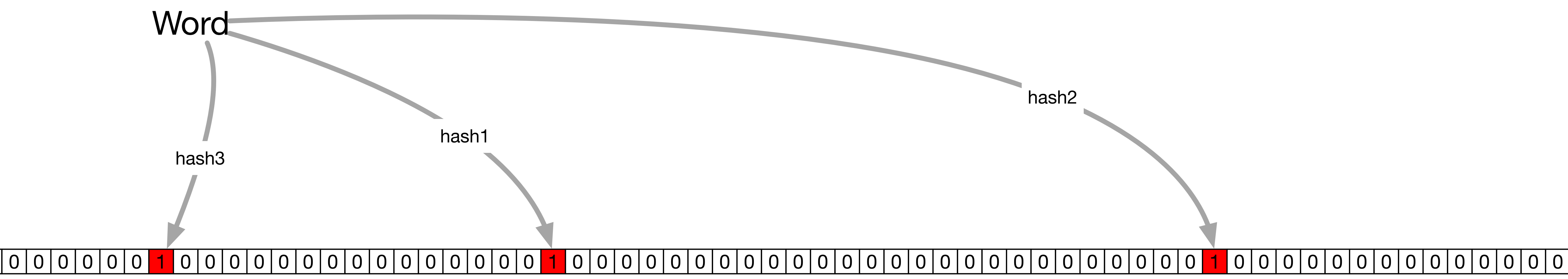
Bloom Filter

- Use a large binary array
  - E.g. a Boolean array in numpy or a bit-array constructed from an array of ints or a byte-array



  - When a word is inserted into a dictionary:
    - Use *k* hash functions to calculate indices into the array
    - Set the array 1 at those locations

Bloom Filter



- Inserting
    - Hash the word *k* times to obtain *k* addresses
    - Addresses might not be distinct
    - Set the bits at those addresses

Bloom Filter

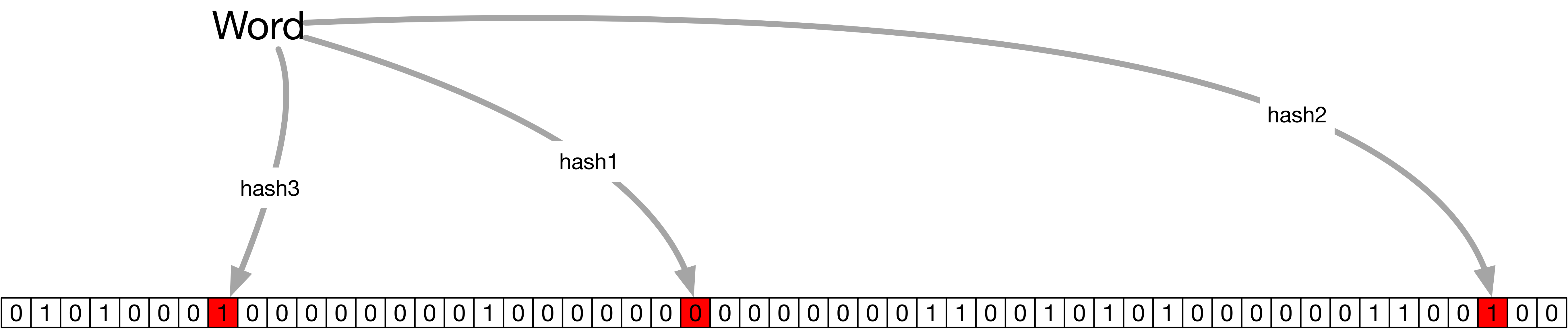
- As we insert more words, more bits will be set
    - When we insert, by chance we might set fewer than *k* bits
        - Because some hash values coincide
    - But we get close to $N \cdot k$ bits set after inserting N words

Bloom Filter

- Look-up
  - Almost the same procedure:
    - Calculate the hashes of the word
    - Check whether the bits at those indices are set
    - If not all are set:
      - return : Not in the dictionary
    - If all are set:
      - return : Probably in the dictionary

Bloom Filter

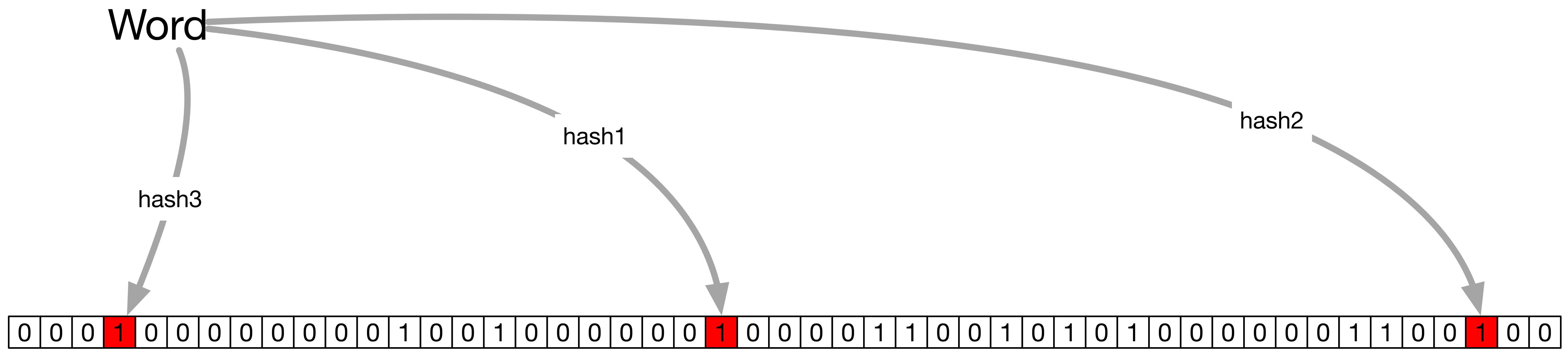
- Lookup



- This word is **not** in the dictionary
    - Bitarray value at the first index is 0

Bloom Filter

- Lookup



- This word is deemed to be in the dictionary
    - The bit array is set at the indices

Bloom Filter

- Can calculate the "false positive" rate
  - Under the assumption of perfect hash functions
  - m — array length, n —nr of words inserted, k — nr of hashes
  - $p_{\text{false}} \approx \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k}$
  - Full story: https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=903775

Bloom Filter

- To deal with false positive rate:
  - Increase the size of the array —> more memory use
  - Use more hash functions —> more computational time

Hash Functions

- Assume a bit array size of 1024
  - Simplest hash:
    - Multiply with a prime, then take the remainder modulo 1024
    - Easiest way, use a bit mask of 0x3ff
      - Example: 100 -> (1297 * 100)%1024

	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	0	1	0	1	0	1	0	0	1	0	0	129700
&	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	mask
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	0	1	0	0	676

Hash Functions

- Somewhat better hash functions
  - Use the middle bits of the square of an integer
    - In our case, just right-shift by a little

Hash-functions

	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	1	1	1	1	0	1	0	0	0	0	0	0	1	0	0	0	0	12970000
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	1	1	1	1	0	1	0	0	0	12970000 >> 8
&	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	(12970000 >> 8) &mask
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0	0	488

```
def hash(i):
  return ((1297 * i**2)>>8)&0x3ff
```

Some code

- Set constants

```
import numpy as np
import random
LENGTH = 1024
MASK = 0x3ff
```

- Use three hash functions

```
def hasher2(ix):
    return  ((17377*ix**2>>4)&MASK,
             ((1297*(ix+5)**2)>>8)&MASK,
             ((10607*(ix+7)**2)>>2)&MASK)
```

Some code

- Create your Bloom filter class using a numpy array

```
class Bloom:
    def __init__(self, LENGTH, MASK, hasher):
        self.length = LENGTH
        self.mask = MASK
        self.array = np.zeros(LENGTH, dtype=bool)
        self.hasher = hasher2
```

- You access the elements in a numpy array using the usual bracket notation

```
if self.array[0] and not self.array[1]:
    print('bit 0 is set and bit 1 not')
```

Some code

- Let's insert 100 integers into the array

```
my_bloom = Bloom(LENGTH, MASK, hasher2)
for i in random.sample(range(10000), 100 ):
    my_bloom.insert(i)
```

- And check what the Bloom filter says when we try out integers definitely not inserted

```
for i in range(30):
    x= random.randint(10000,10000000)
    print(x, my_bloom.look_up(x))
```

Result

- One false positive
  - Can do better with:
    - better hash functions
    - more hash functions
    - longer array

```
94426 False
2806057 False
7098498 False
9368615 False
4456201 False
161151 False
4010924 False
125807 False
5711174 False
4467586 False
284903 False
1123226 False
9890460 False
1490195 False
859823 False
1091281 True
8179607 False
5403847 False
9536029 False
3465063 False
3282630 False
367168 False
322341 False
3631739 False
747199 False
5480083 False
100842 False
4239027 False
5865436 False
5668470 False
```