**CS 677**: Big Data

# Bloom Filters

Lecture 14

# Moving on…

- We've learned a bit about Spark, and we'll start using it soon
- In the meantime, let's look at some algorithms/data structures specifically designed for big data
- Inspiration: Spark has a method called `.countApprox()`

# Today's Schedule

- Bloom Filters
- Configuring a Bloom Filter

# Today's Schedule

- **Bloom Filters**
- Configuring a Bloom Filter

# Setting the Scene

- Imagine, if you will, a centralized component that has an index of all the data in a distributed system
  - It simplifies things, but at what cost?
  - Single point of failure
  - Lots of memory is required to store the file system namespace
- If we are Google (in other words, rich 💵 💵 💵), we can buy a very fancy machine to take care of this
  - Based on your tuition, we probably *should* have this at USF… 😄

# What if there was another way?

- We don’t have the hardware for a truly Google-scale “NameNode” / “Controller”, etc.
- Instead, we can store our index on a large disk, but that is going to be slow
- Given a request for a file, let’s predict whether we actually have it stored somewhere or not without consulting our index
  - What is the **probability** that a node contains the data?
- We’ll do this with a data structure called *bloom filters*

# Bloom Filter [1/2]

- Compact data structure to test for set membership
- Supports two functions:
  - `put(data)` : places some information in the filter
  - `get(data)` : reports the **probability** of whether or not the data was put in the filter
- May produce false positives but **never** false negatives
  - If the bloom filter says it wasn't inserted, then it definitely wasn't!

# Bloom Filter [2/2]

- Bloom filters give us two answers:
  - **Maybe** (with a probability)
  - **Definitely not**
- You can think of it as a HashSet that throws away the keys
  - We can't get the data back
    out of the bloom filter,
    but it is **very** compact
    in memory!



# Use Cases [1/3]

- When is a data structure that throws away data actually useful? Surprisingly quite often!
- Perhaps we can't store the entire set in memory
  - NameNode index
  - Word dictionary for spell checking (embedded devices)
  - "Malicious websites" list: don't keep the entire list, check if a URL might be malicious, then look up in an online database

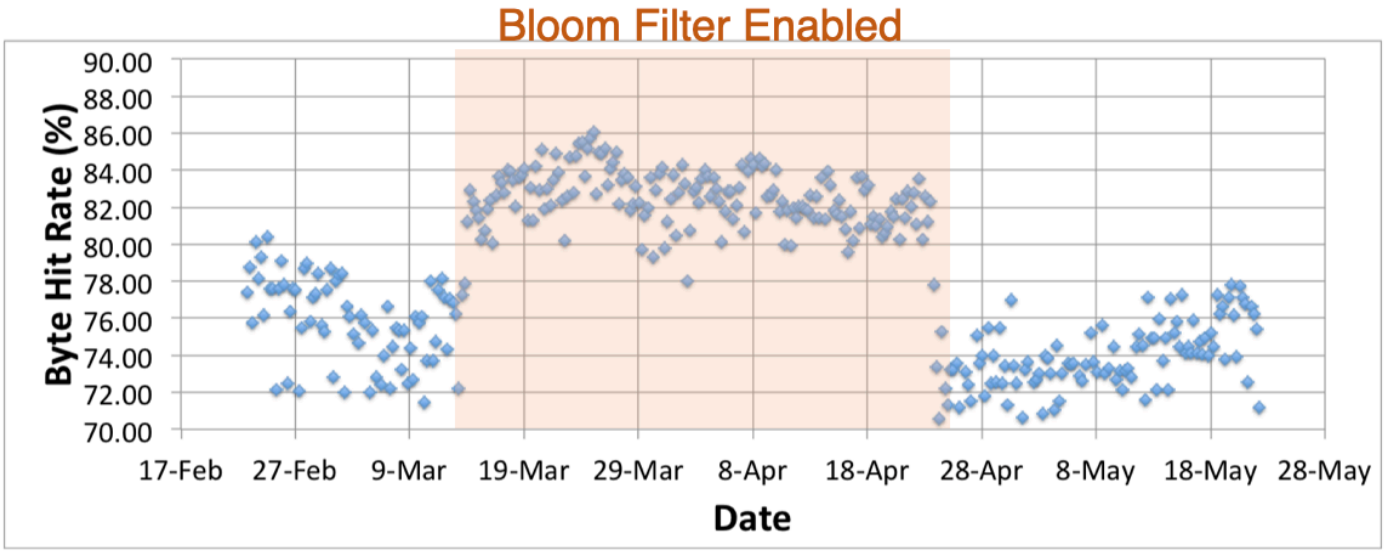
# Use Cases [2/3]

- Generating a unique name or ID for something: if the bloom filter replies “definitely not,” then you know it’s unique!
- Preventing caches from including “one hit wonders”
  - ~75% of the unique URLs you click are one-time visits
  - Only cache URLs that were accessed at least once before (second request)
    - No need to store entire list of URLs visited

# Use Cases [3/3]

- Acting as a gatekeeper to a high-latency hardware (such as hard disks – avoid HDD accesses if they're not actually needed!)

# Content Delivery: Akamai




The bloom filter provides a better hit rate because "one hit wonders" are no longer using up space in the cache.

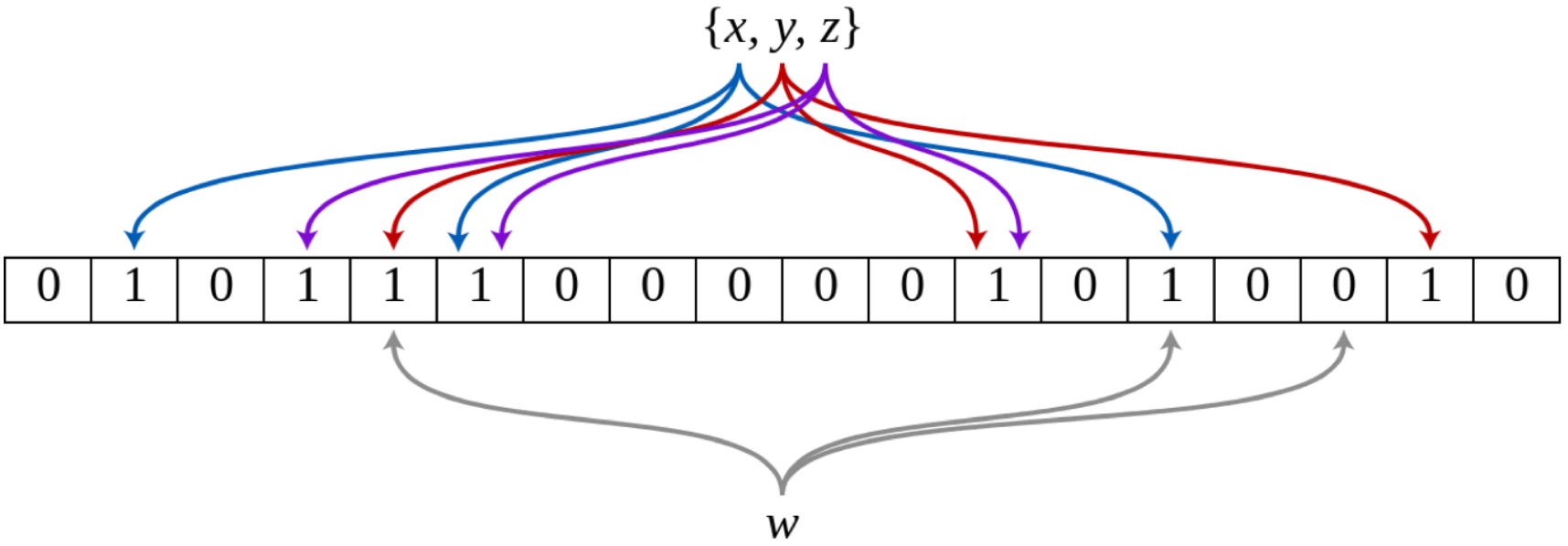
# How it Works: put()

1. Create a bit array of $m$ bits
2. For each `put(key)` operation, hash the key $k$ times
   - Where $k$ is the number of hash functions we're using
   - There is a formula to determine how many we need… stay tuned
3. Map each hashed value to its corresponding index in the bit array
   - $hash \% m$
4. Set these bits to $1$

# How it Works: get()

1. For each `get(key)` operation, hash the key $k$ times
2. Map each hashed value to its corresponding index in the bit array
   - (same as `put()` so far)
3. If **any** of the bits at these indexes are set to $0$, then `key` is **definitely** not in the set
4. If all the bits are set to $1$, then `key` *might* be in the set

# A Picture




Source: https://en.wikipedia.org/wiki/Bloom_filter

# Interactive Demo

https://www.jasondavies.com/bloomfilter/

# Building a Bloom Filter [1/2]

- To implement a bloom filter, we need:
  - An array of bits
  - Multiple hash functions
- When putting an item in the filter, the data is passed to the hash functions
- The **hash space** of each function is mapped to our array of bits

# Building a Bloom Filter [2/2]

- We take the position in the hash space, map it to our array of bits, and then set the corresponding bit to $1$
  - Repeat for all hash functions
- To perform a lookup, we repeat the process
  - If all the positions in the bit array are set to $1$, then we can return a “maybe”
  - If **any** of the positions are a $0$, then we return “no”
    - Short-circuits the lookup process

# Collisions

- If the size of our bit array is small, then there is a good chance of **collisions**
- Two inputs map to the same bit:
  - hash(my_dog.jpg) => bit #3
  - hash(secret_passwords.txt) => bit #3
- This is the source of **uncertainty** in bloom filters
- How do we decide how large to make our filters?

# Today's Schedule

- Bloom Filters
- **Configuring a Bloom Filter**

# Sizing a Bloom Filter

- A reasonable starting point: bit arrays sized at 20% of your input size
- If we can accept more uncertainty, we can maintain smaller bit arrays in memory
- Fewer than 10 bits per element are required for a 1% false positive probability, independent of the size or number of elements in the set
  - – Bonomi et al., “An Improved Construction for Counting Bloom Filters”

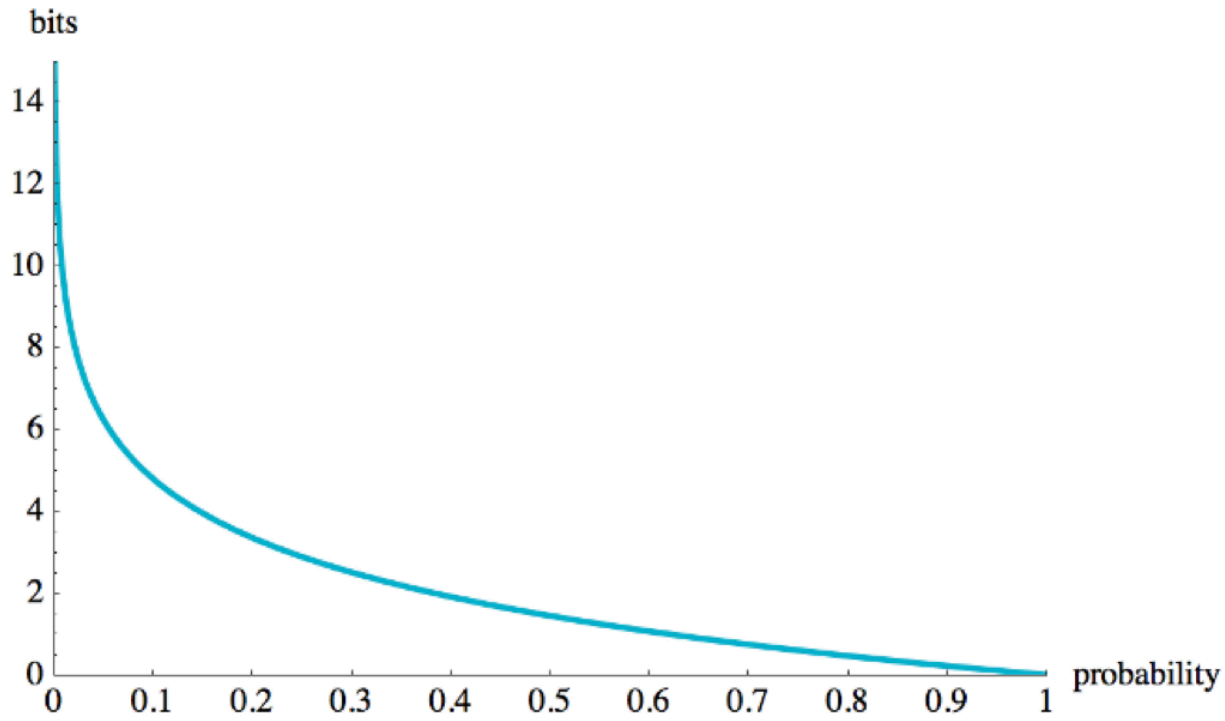
# False Positive Rate vs. Bits [1/2]

| False Positive Rate | Bits per Element |
|---|---|
| 50% | 1.44 |
| 10% | 4.79 |
| 2% | 8.14 |
| 1% | 9.58 |
| 0.1% | 14.38 |
| 0.01% | 19.17 |

(*Assuming that an optimal number of hash functions is used.)

Source: https://corte.si/posts/code/bloom-filter-rules-of-thumb/index.html

# False Positive Rate vs. Bits [2/2]




Source: https://corte.si/posts/code/bloom-filter-rules-of-thumb/index.html

# Caveat: False Positive Rate

- New research from 2020 shows the FPR from the original paper was incorrect
- See schedule page for details…

# Resizing a Bloom Filter

- If our false positive probability starts to go up, we need to resize the bloom filter
  - Kind of not *really* possible…
- Generally accomplished by creating a new bloom filter and then inserting the values back in
  - Do we actually have the original data?
  - Big downside in situations where we can't predict how many values we'll see
  - This is really expensive! Avoid at all costs.

# Choosing Hash Functions

- Most of the examples we've dealt with have three hash functions, but that's not optimal
  - Let's think about the trade-off here…
    - More hash functions means decreased false positives
    - It also means you need more bits per element
- The rule of thumb is ~(0.6 to 0.7) * bits per element
- Usually we'll have more than 3 hash functions, but remember that hash functions are **expensive**!
  - Shoot for optimal but dial back if not fast enough

# Bloom Filter Calculator

- We can play with values of $m$, $k$, and $p$ here:
  - https://hur.st/bloomfilter/

# Hash Algorithms

- As you can imagine, using cryptographic hash functions doesn't buy us anything extra here
  - MD5, SHA-1 are going to be slow, but they are at least uniformly distributed (and sometimes these are hardware accelerated, so…)
- It's better to use a hash function that's designed to be uniform and fast
- *murmur3* is one good candidate

# Speeding up Hashing

- We can use the Kirsch-Mitzenmacher Optimization to greatly reduce hashing costs
  - Create two hash functions and then combine them for each “virtual” hash function

```
def generateHashes(obj):
    h1 = murmur3_hash(obj)
    h2 = murmur3_hash(obj, h1) # (seed)
    for i in range(3):
        hash[i] = h1 + (i * h2)
```

# Deletions

- With the standard algorithm we've discussed, we can't delete elements from the set
  - Why not?
- How could we modify the algorithm to allow deletes?
- Answer: *counting bloom filters*. Each cell in our array becomes a counter instead
  - Deletions decrement the counter
  - If the counter drops to 0, that's the same as having the bit switched from 1 to 0 (nothing there!)

# Thought Experiment: P1

- Could we use bloom filters in P1? There are a couple ways.
- Allowing efficient single storage node search
- Insert file paths in the bloom filter as they are stored in the system
  - (When the controller receives a heartbeat informing them of new chunk data)
- When doing a retrieval, consult the **in-memory** bloom filter before accessing the **on-disk** index
  - Bloom filter responds with “not found”? Then no need to hit the disk!