
Go ahead… just try taking one of my blocks

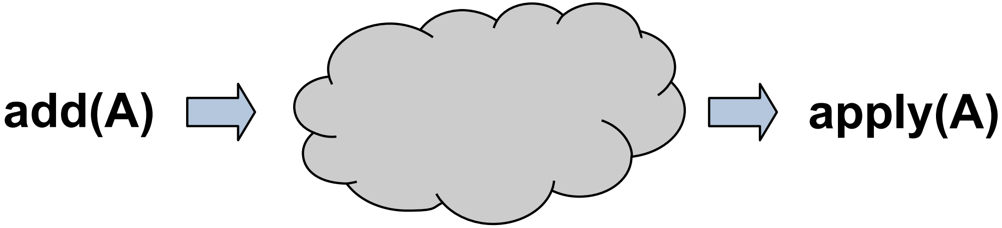
CSE 250
Lecture 33

Bloom Filters

Fall 2022

“Lossy Sets”

- Set[A]
  - **add(a: A)**: Insert **a** into the set
  - **apply(a: A)**: Return true if **a** is in the set



- What if we didn’t need apply to be perfect?

Reading Data From Disk

- Checking disk to see if a data record is present is slow.
  - Even B+ Trees usually require an IO to tell you the record isn't there
- **Idea**: Keep an in-memory summary of the data.
  - If summary says key in layer: access the layer
  - If summary says key not in layer: skip the layer
- Need some guarantees
  - If summary incorrectly says key in layer: Extra work
- Not great, but we were going to do the work anyway
  - If summary incorrectly says key not in layer: Error!
- Changes semantics. Not good!

OK!

NOT ok!

Lossy Sets

- LossySet[A]
  - **add(a: A)**: Insert **a** into the set.
  - **apply(a: A)**:
- If **a** is in the set, <u>always</u> return true
- If **a** is not in the set, <u>usually</u> return false
  - Is allowed to return true, even if **a** is not in the set

Lossy Sets

```
scala> lossySet.add(“Wesley”)
scala> lossySet.add(“Buttercup”)
scala> lossySet.add(“Inigo”)

scala> lossySet(“Wesley”)
val res0: Boolean = true

scala> lossySet(“Inigo”)
val res1: Boolean = true

scala> lossySet(“Vizini”)
val res2: Boolean = false

scala> lossySet(“Fezzik”)
val res3: Boolean = true
```

Lossy Sets

Key Insight: If apply doesn’t need to always be right,
The lossy set doesn’t need to store everything.

Lossy Sets

```
class TrivialLossySet[A] extends LossySet[A]
{
  def add(a: A): Unit = { /* do nothing */ }

  def apply(a: A): Boolean = true
}
```

Correct, but not very useful

Lossy Sets

- **Idea:** Histogram
  - Bucketize the keys
    - First letter of string
    - Ranges of values
  - Keep one <u>bit</u> per bucket
- add(a: A): Set the bit for a’s bucket (to 1)
- apply(a: A): Return true if the bit for a’s bucket is set

Lossy Sets

```
class StringHistogramLossySet extends LossySet[String]
{
  val bits = new Array[Boolean](256)

  def add(a: String): Unit = {
    val bucket = a(0).toInt
    bits(bucket) = true
  }

  def apply(a: A): Boolean = {
    val bucket = a(0).toInt
    return bits(bucket)
  }
}
```

Lossy Sets

- **Idea:** Hash-Based Histogram
  - Bucketize the keys into N buckets
- Hash function
  - Keep one <u>bit</u> per bucket
- add(a: A): Set the bit for a’s bucket (to 1)
- apply(a: A): Return true if the bit for a’s bucket is set

Lossy Sets

```
class LossyHashSet[A](_size: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def add(a: A): Unit = {
    val bucket = a.hashCode % _size
    bits(bucket) = true
  }

  def apply(a: A): Boolean = {
    val bucket = a.hashCode % _size
    return bits(bucket)
  }
}
```

Lossy Hash Sets

- **add(a)** then **apply(b)**
  - What does apply(b) return, and when?
    - true: hash(a) = hash(b) mod _size
    - false: hash(a) != hash(b) mod _size
  - What is the probability of each, with N buckets?
    - true: $^{1}/_{N}$
    - false: $^{N-1}/_{N}$

Lossy Hash Sets

- Show of hands
  - Who was born in:
    - >= 2004
    - 2003?
    - 2002?
    - 2001?
    - 2000?
    - <= 1999?

Lossy Hash Sets

- Show of hands
- What is the color of your shirt?
- White?
- Black?
- Red?
- Green?
- Blue?

Lossy Hash Sets

- Fewer collisions with TWO features than with one
  - … but need more space to store both features
- **Idea:** Use the same number line for both features
  - e.g.: Birth Year + Sibling's Birth Year(s)
  - Assign each record to 2 buckets

Lossy Hash Sets

```
class LossyDoubleHashSet[A](_size: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def hash1(a: A): Int = ???
  def hash2(a: A): Int = ???

  def add(a: A): Unit = {
    bits( hash1(a) % _size ) = true
    bits( hash2(a) % _size ) = true
  }

  def apply(a: A): Boolean = ???
}
```

Lossy Hash Sets

- apply(a): Unit = ???

bits(hash1(a) % _size)	bits(hash2(a) % _size)	apply(a)
T	T	**T**
T	F	**F**
F	T	**F**
F	F	**F**

bits(hash1(a)) && bits(hash2(a))

Lossy Hash Sets

```
class LossyDoubleHashSet[A](_size: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def hash1(a: A): Int = ???
  def hash2(a: A): Int = ???

  def add(a: A): Unit = {
    bits( hash1(a) % _size ) = true
    bits( hash2(a) % _size ) = true
  }

  def apply(a: A): Boolean = {
    return bits( hash1(a) % _size ) && bits( hash2(a) % _size )
  }
}
```

Lossy Hash Sets

- **add(a)** then **apply(b)**
  - What does apply(b) return, and when?
    - true: hash1(a)=hash1(b) AND hash2(a)=hash2(b) (mod _size)
    - false: otherwise
  - What is the probability of each, with N buckets?
    - true: $\sim(^1/_N)^2$
    - false: $\sim(^{N-1}/_N)^2$

Lossy Hash Sets

Which chance of collision is preferable?

$$\frac{1}{N} \qquad \frac{1}{N^2}$$

How do we get 2 hash functions?

```
def hash1[A](a: A) =
  hash( 1 + a.hashCode )

def hash2[A](a: A) =
  hash( 2 + a.hashCode )
```

How do we get 2 hash functions?

```
val SEED1 = 123104912035
val SEED2 = 406923456234

def hash1[A](a: A) =
  hash( SEED1 + a.hashCode )

def hash2[A](a: A) =
  hash( SEED2 + a.hashCode )
```

Don’t use sequentially adjacent values

How do we get 2 hash functions?

```
val SEED1 = 123104912035
val SEED2 = 406923456234

def hash1[A](a: A) =
  hash( SEED1 ^ a.hashCode )

def hash2[A](a: A) =
  hash( SEED2 ^ a.hashCode )
```

Use bitwise-XOR instead of +

How do we get K hash functions?

```
val SEED1 = 123104912035
def hash1[A](a: A) =
  hash( SEED1 ^ a.hashCode )

val SEED2 = 406923456234
def hash2[A](a: A) =
  hash( SEED2 ^ a.hashCode )

val SEED3 = 908057230543
def hash3[A](a: A) =
  hash( SEED3 ^ a.hashCode )
```

Generate as many hash functions as needed

How do we get K hash functions?

```
val SEEDS = Seq(123104912035, 406923456234, ...)
def ithHash[A](a: A, i: Int) =
  hash( SEEDS(i) ^ a.hashCode )
```

Bloom Filters

- Overall Structure
  - _**size** bits
  - _**k** hash functions

Bloom Filters

```
class BloomFilter[A](_size: Int, _k: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def add(a: A): Unit = {
    for(i <- 0 until _k) { bits( ithHash(a, i) % _size ) = true }
  }

  def apply(a: A): Boolean = ???
}
```

Bloom Filters

```
class BloomFilter[A](_size: Int, _k: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def add(a: A): Unit = {
    for(i <- 0 until _k) { bits( ithHash(a, i) % _size ) = true }
  }

  def apply(a: A): Boolean = {
    for(i <- 0 until _k) {
      if( !bits( ithHash(a, i) % _size ) { return false; }
    }
    return true
  }
}
```

Bloom Filters

```
class BloomFilter[A](_size: Int, _k: Int) extends LossySet[A]
{
  val bits = new Array[Boolean](_size)

  def add(a: A): Unit = {
    for(i <- 0 until _k) { bits( ithHash(a, i) % _size ) = true }
  }

  def apply(a: A): Boolean = {
    return (0 until _k).foreach { i => bits(ithHash(a, i) % _size) }
  }
}
```

Bloom Filter Parameters

- _size
  - Intuitively: More space, fewer collisions
- _k
  - Intuitively: more hash functions means…
    - …more chances for one of **b**’s bits to be unset.
    - …more bits set = higher chance of collisions.

Bloom Filters: Analysis

$$\frac{1}{N}$$

The probability that 1 bit is set by 1 hash function

Bloom Filters: Analysis

$$1 - \frac{1}{N}$$

The probability that 1 bit is **not** set by 1 hash function

Bloom Filters: Analysis

$$\left(1 - \frac{1}{N}\right)^k$$

The probability that 1 bit is **not** set by k hash functions

Bloom Filters: Analysis

$$\left(1 - \frac{1}{N}\right)^{kn}$$

The probability that 1 bit is **<u>not</u>** set by k hash functions

... over n distinct calls to **add**

Bloom Filters: Analysis

$$1 - \left(1 - \frac{1}{N}\right)^{kn}$$

The probability that 1 bit is set by **at least one** of k hash functions

... over n distinct calls to **add**

Bloom Filters: Analysis

$$\approx \left(1 - \left(1 - \frac{1}{N}\right)^{kn}\right)^{k}$$

The probability that all k randomly selected bits of element **b**

... are set by **<u>at least one</u>** of k hash functions

... over n distinct calls to **add**

Bloom Filters: Analysis

The chance of collision in a Bloom filter with parameters <u>k</u>, <u>N</u> after <u>n</u> distinct elements have been added

$$\approx \left(1 - e^{-\frac{kn}{N}}\right)^k$$

The probability that all k randomly selected bits of element **b**

... are set by **<u>at least one</u>** of k hash functions

... over n distinct calls to **add**

Bloom Filters: Analysis

$$\approx \left(1 - e^{-\frac{kn}{N}}\right)^k$$

As $e^{kn/N}$ grows, the chance of collision shrinks

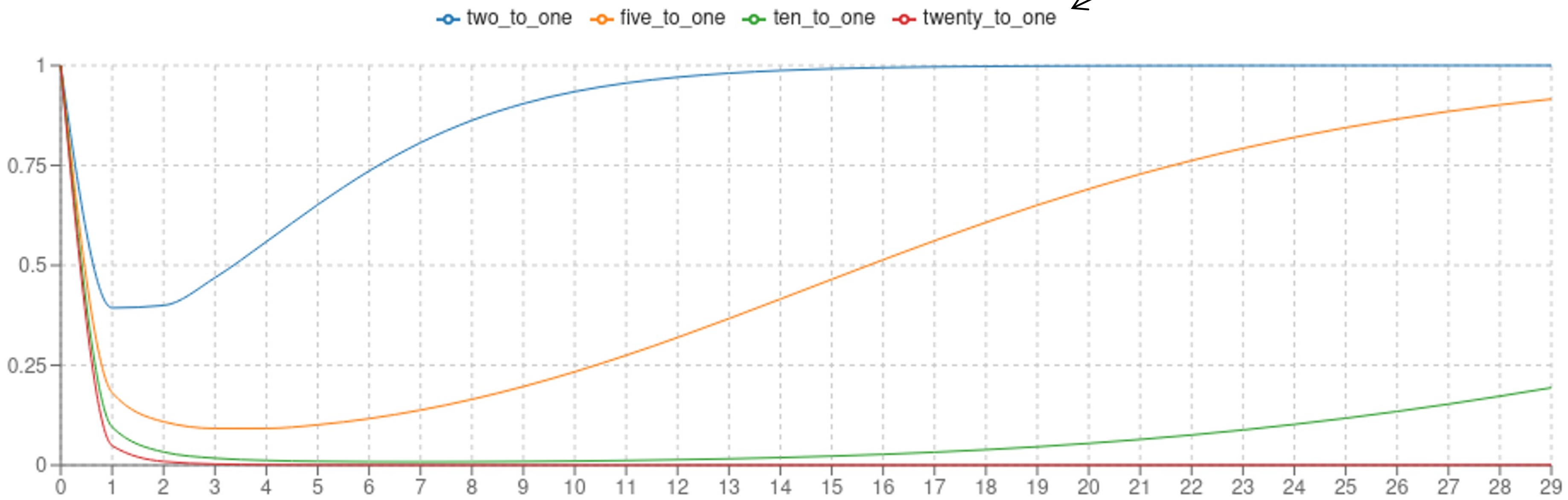
Bloom Filters: Analysis

$$\left(1 - e^{-\frac{kn}{N}}\right)^k$$

- **Ideal**: Pick N, k that minimize collision chance:
  - N
    - Smaller N, more opportunities for collisions
    - Bigger N, more space used
  - k
    - Smaller k, fewer tests, more chance of collisions
    - Bigger k, more bits set, more chance of collisions
    - Sweet spot in the middle

Bloom Filters: Analysis

$$\frac{N}{n}$$



Optimum at: $k = c \cdot \frac{N}{n}$

Bloom Filters: Analysis

$$k = c \cdot \frac{N}{n}$$

$$n = c\frac{N}{k}$$

N and n are linearly related
O(n) buckets required

Bloom Filters: Analysis

- N/n = 5 → ~10% collision chance
- N/n = 10 → ~1% collision chance

- 10 bits vs
  - 32 bits for one Int (3 to 1 savings)
  - 64 bits for a Double/Long (6 to 1 savings)
  - ~8000 bits for a full record (800 to 1 savings)

Bloom Filters: Analysis

- vs B+Tree or Binary Search Tree implementing Set
  - $O(k \cdot \mathbf{cost}_{hash}) \approx O(1)$ **vs** $O(\log(n) \cdot \mathbf{cost}_{compare})$ runtime
  - No directory pages (constant factor extra memory required)
- vs Hash Table implementing Set
  - Guaranteed $O(k \cdot \mathbf{cost}_{hash}) \approx O(1)$ **vs** Expected $O(\mathbf{cost}_{hash})$
  - No ‘fill factor’ (constant factor extra memory required)
- vs Array implementing Set
  - $O(k \cdot \mathbf{cost}_{hash}) \approx O(1)$ **vs** $O(n \cdot \mathbf{cost}_{compare})$