CSE 250: Bloom Filters

Lecture 38

Dec 06, 2023

Reminders

- WA3 due Sun, Dec 10
  - PA3 showed you that even ‘anonymized’ data can be problematic
  - WA3: Look for other cases of problems
- Course Evals Bonus
  - Get to 90% completion across all 3 sections, we’ll release an exam question.
  - Section C: 24/112 as of Friday (22%)
- Do you like this class, especially the last 2 lectures?
  - Look at CSE 410 (soon to be CSE 350)
- Do you like/hate this class?
  - Email Eric and me about being a TA!

Use Case: Reading Data From Disk

Checking disk to see if a record is present is slow if you don't know that it exists.

- Even B+ Trees require at least one IO to tell you a record isn't there.

Idea: Keep an in-memory summary of the data.

- If the summary says the key is present: Access disk.
- If the summary says the key is not present: Return "record does not exist"

In Memory Summary

Version 1: Keep a list of all keys in the list in memory.

What ADT is appropriate? A Set

What data structures can implement a set?

- Sorted Array ($O(\log N)$ access)
- Balanced Binary Tree ($O(\log N)$ access, update)
- Hash Table (Expected $O(1)$ access, update)

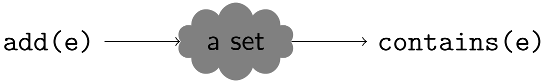
How much memory is needed? $O(N)$

A list of keys is basically a bigger Fence Pointer Table.

Lossy Sets

Set<T>

- `public void add(T e)`: Add the element e.
- `public boolean contains(T e)`: Return true if e is in the set.



What if we didn't need `contains` to be perfect?

Lossy Sets

Idea: Keep an in-memory summary of the data.

- If the summary says the key is present: Access disk.
  - A mistake here means we do unnecessary work.
  - Not ideal, but we go to disk without the summary. 🤷
- If the summary says the key is not present: Return "record does not exist"
  - A mistake here means the read returns a wrong result.
  - This would cause a bug (e.g., in a B+Tree). 🙁

We may be able to get a win if we trade rare false positives for space.

Lossy Sets

LossySet<T>

- `public void add(T e)`: Add the element `e`.
- `public boolean contains(T e)`:
  - if `e` is in the set, always return true.
  - if `e` is not in the set, usually return true.

`contains(e)` **is allowed to return true, even if** `e` **is not in the set.**
. . . but ideally, this happens as rarely as possible.

Lossy Sets

```
lossySet.add("Westley")
lossySet.add("Buttercup")
lossySet.add("Inigo")
```

```
System.out.println(lossySet("Westley"))    // → true
```

```
System.out.println(lossySet("Inigo"))    // → true
```

```
System.out.println(lossySet("Vizini"))    // → false
```

```
System.out.println(lossySet("Fezzik"))    // → true
```

Lossy Sets

Key Insight: If `contains` doesn't need to always be right, the lossy set doesn't need to store everything.

Lossy Sets

```java
public class TrivialLossySet<T> implements LossySet<T>
{
  public void add(T e) {  /* do nothing */ }
  public boolean contains(T e) {  return true; }
}
```

The trivial lossy set is correct, but not useful.

Lossy Sets

Idea: Bucket the keys

- First letter of the string
- Ranges of values
- Hash values (mod number of buckets)

Implementation: Store each bucket as one bit.

- `add(e)`: Set the bit for a's bucket to 1.
- `contains(e)`: Return true if the bit for a's bucket is 1.

Lossy Sets

```
public class HashtogramLossySet<T> implements LossySet<T>
{
  static int SIZE = 256;     // 256 is arbitrary
  boolean[] bits = new boolean[SIZE]

  public void add(T e) {
    int bucket = e.hashCode % SIZE;
    bits[bucket] = true;
  }

  public boolean contains(T e) {
    int bucket = e.hashCode % SIZE;
    return bits[bucket];
  }
}
```

HashtogramLossySet Example

1. `add("foo")`
2. `contains("bar")`

What does `contains("bar")` return?

- `true` iff `"foo".hashCode % SIZE == "bar".hashCode % SIZE`
- `false` iff `"foo".hashCode % SIZE != "bar".hashCode % SIZE`

What is the probability of each result?

- `true` with probability $\frac{1}{\mathtt{SIZE}}$
- `false` with probability $\frac{\mathtt{SIZE}-1}{\mathtt{SIZE}}$

Hashtogram Lossy Set

Problem: Collisions:

- One inserted value causes $\frac{1}{\texttt{SIZE}}$ of all calls to `contains` to return true.
- This number gets worse with more inserted values.

Lossy Hash Sets

Class Activity: Who was born in?
(Participation Optional)

- $\geq$ 2005?
- 2004?
- 2003?
- 2002?
- 2001?
- $\leq$ 2000?

Lossy Hash Sets

Class Activity: What is the color of your shirt/top? (Participation Optional)

- White?
- Black?
- Red?
- Green?
- Blue?
- Other?

Hash Set Collisions

There were fewer collisions with two features than one.

- ... but we need more space to store two features than one

Idea: Use the same set of histograms for both features.

Example:

- Birth Year (mod 25)
- Sibling's birth year (mod 25)

Each Record is assigned to two buckets.

Double Hashtogram Lossy Set

```
public class DoubleHashtogram<T> implements LossySet<T>
{
  static int SIZE = 256;      // 256 is arbitrary
  boolean[] bits = new boolean[SIZE]

  public int hash1(T e) { /* ... */ }
  public int hash2(T e) { /* ... */ }

  public void add(T e) {
    bits[hash1(e) % SIZE] = true;
    bits[hash2(e) % SIZE] = true;
  }

  public boolean contains(T e) {
    // ???
  }
}
```

Double Hashtogram Lossy Set

`bits[hash1(e)% SIZE]`	`bits[hash2(e)% SIZE]`	`contains(e)`
`true`	`true`	`true`
`true`	`false`	`false`
`false`	`true`	`false`
`false`	`false`	`false`

This is AND: `bits[hash1(e)% SIZE] && bits[hash2(e)% SIZE]`

Double Hashtogram Lossy Set

```
public class DoubleHashtogram<T> implements LossySet<T>
{
  static int SIZE = 256;      // 256 is arbitrary
  boolean[] bits = new boolean[SIZE]

  public int hash1(T e) { /* ... */ }
  public int hash2(T e) { /* ... */ }

  public void add(T e) {
    bits[hash1(e) % SIZE] = true;
    bits[hash2(e) % SIZE] = true;
  }

  public boolean contains(T e) {
    bits[hash1(e) % SIZE] && bits[hash2(e) % SIZE]
  }
}
```

Double Hashtogram Lossy Set Example

1. `add("foo")`
2. `contains("bar")`

What does `contains("bar")` return?

- true iff `hash1("foo") == hash1("bar")` AND `hash2("foo") == hash2("bar")`[1]
- false otherwise

What is the probability of each result?

- true with probability $\sim \left(\frac{1}{\texttt{SIZE}}\right)^2$
- false with probability $\sim \left(\frac{\texttt{SIZE}-1}{\texttt{SIZE}}\right)^2$

[1] mod size

Double Hashtogram Lossy Set

Which chance of collision is preferable?

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

How do we get 2 hash functions?

```
public int hash1(T e){
  return (1 + (e.hashCode)).hashCode
}
public int hash2(T e){
  return (2 + (e.hashCode)).hashCode
}
```

How do we get 2 hash functions?

```
    static int SEED1 = 123104912035;
    static int SEED2 = 406923456234;

    public int hash1(T e){
      return (SEED1 + (e.hashCode)).hashCode
    }
    public int hash2(T e){
      return (SEED2 + (e.hashCode)).hashCode
    }
```

Avoid sequentially adjacent values.

How do we get 2 hash functions?

```
static int SEED1 = 123104912035;
static int SEED2 = 406923456234;

public int hash1(T e){
  return (SEED1 ^ (e.hashCode)).hashCode
}
public int hash2(T e){
  return (SEED2 ^ (e.hashCode)).hashCode
}
```

Use bitwise-XOR ($\hat{}$) instead of addition ($+$)

How do we get K hash functions?

```
  static int SEED1 = 123104912035;
  static int SEED2 = 406923456234;
  static int SEED3 = 908057230543;

  public int hash1(T e){
    return (SEED1 ^ (e.hashCode)).hashCode
  }
  public int hash2(T e){
    return (SEED2 ^ (e.hashCode)).hashCode
  }
  public int hash3(T e){
    return (SEED3 ^ (e.hashCode)).hashCode
  }
```

We can generate as many hash functions as needed.

How do we get K hash functions?

```
static int SEEDS = [
  123104912035, 406923456234, 908057230543, ...
]

public int kthHash(int k, T e)
{
  return (SEEDS[k] ^ e.hashCode).hashCode
}
```

Bloom Filters

Parameters

- `SIZE` bits
- `HASHES` hash functions

Bloom Filters

```
class BloomFilter<T> extends LossySet<T>
{
  int SIZE   = /* ... */;
  int HASHES = /* ... */;
  boolean[] bits = new boolean[SIZE];

  public void add(T e) {
    for(k = 0; k < HASHES; k++)  {
      bits( kthHash(k, e) % SIZE ) = true;
    }
  }

  public boolean contains(T e) {
    /* ??? */
  }
}
```

Bloom Filters

```
class BloomFilter<T> extends LossySet<T>
{
  int SIZE   = /* ... */;
  int HASHES = /* ... */;
  boolean[] bits = new boolean[SIZE];
  public void add(T e) {
    for(k = 0; k < HASHES; k++)  {
      bits( kthHash(k, e) % SIZE ) = true;
    }
  }
  public boolean contains(T e) {
    for(k = 0; k < HASHES; k++)  {
      if( bits( kthHash(k, e) % SIZE ) ) { return false; }
    }
    return true;
  }
}
```

Bloom Filter Parameters

How do we set a bloom filter's parameters?

- `SIZE`
  Intuitively: More space means fewer collisions
- `HASHES`
  Intuitively: More hash functions means...
  - ...More chances for one of **b**'s bits to be unset.
    (lower collision chance)
  - ...More bits set (higher collision chance).

Increasing `SIZE` trades space for a lower false positive rate.
For `HASHES`, there's a midpoint that minimizes collision chance

Bloom Filters: Analysis

The probability that 1 bit is set by 1 hash function.

$$\frac{1}{\texttt{SIZE}}$$

Bloom Filters: Analysis

The probability that 1 bit is <u>not</u> set by 1 hash function.

$$1 - \frac{1}{\texttt{SIZE}}$$

Bloom Filters: Analysis

The probability that 1 bit is not set by HASHES hash functions.

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

Bloom Filters: Analysis

The probability that 1 bit is not set by HASHES hash functions.
...over N distinct calls to add.

$$\left(1 - \frac{1}{\texttt{SIZE}}\right)^{\texttt{HASHES} \cdot N}$$

Bloom Filters: Analysis

The probability that 1 bit is set by at least one of `HASHES` hash functions.

...over N distinct calls to `add`.

$$1 - \left(1 - \frac{1}{\texttt{SIZE}}\right)^{\texttt{HASHES} \cdot N}$$

Bloom Filters: Analysis

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

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

. . . over N distinct calls to add.

$$\left(1 - \left(1 - \frac{1}{\texttt{SIZE}}\right)^{\texttt{HASHES}\cdot N}\right)^{\texttt{HASHES}}$$

Bloom Filters: Analysis

The probability that all `HASHES` randomly selected bits of element **b**
...are set by <u>at least one</u> of `HASHES` hash functions.
...over N distinct calls to `add`.

$$\approx \left(1 - e^{-\frac{\texttt{HASHES} \cdot N}{\texttt{SIZE}}}\right)^{\texttt{HASHES}}$$

The chance of collision in a Bloom filter with parameters `HASHES`, `SIZES` after inserting N elements.

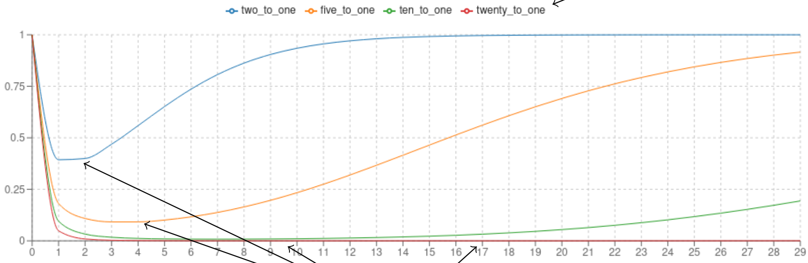
Bloom Filters: Analysis

$$\approx \left(1 - e^{-\frac{\texttt{HASHES} \cdot N}{\texttt{SIZE}}}\right)^{\texttt{HASHES}}$$

As $e^{\frac{\texttt{HASHES} \cdot N}{\texttt{SIZE}}}$ grows, the chance of collision shrinks.

Bloom Filters: Analysis

$\frac{\texttt{SIZE}}{N}$



Optimum at $\texttt{HASHES} = c \cdot \frac{\texttt{SIZE}}{N}$

Bloom Filters: Analysis

$$\texttt{HASHES} = c \cdot \frac{\texttt{SIZE}}{N}$$

$$N = c \cdot \frac{\texttt{SIZE}}{\texttt{HASHES}}$$

N and `SIZE` are linearly related ($O(N)$ buckets required)

Bloom Filters: In Practice

- $\frac{SIZE}{N} = 5 \quad \rightarrow \quad \sim 10\%$ collision chance.
- $\frac{SIZE}{N} = 10 \quad \rightarrow \quad \sim 1\%$ collision chance.

10 $\underline{\text{bits}}$ vs

- 32 bits for one integer (3 to 1 savings)
- 64 bits for one double/long (6 to 1 savings)
- 512 bits for a 64 byte record (50 to 1 savings)

Bloom Filters: In Practice

- **vs** B+Tree or Binary Search implementing Set
  - $O(\texttt{HASHES} \cdot \textbf{cost}_{\text{hash}} \approx O(1)$ **vs** $O(\log N \cdot \textbf{cost}_{\text{compare}})$
  - No directory pages required (better memory/IO)
- **vs** Hash Table implementing Set
  - Unqualified $O(\texttt{HASHES} \cdot \textbf{cost}_{\text{hash}} \approx O(1)$ **vs** Expected $O(\textbf{cost}_{\text{hash}}$
  - No 'fill factor' (constant factor extra memory required)
- **vs** Array implementing Set