

Probabilistic data structures

Bloom Filters

- Bloom filters (BF) are used for answering set membership.
- Given a BF, u can ask it a question
 - *Contains (Element e)*
 - returns true or false.
 - Element can be an integer, or string, or whatever...as long as u can compute a hash value for it).

Bloom Filters

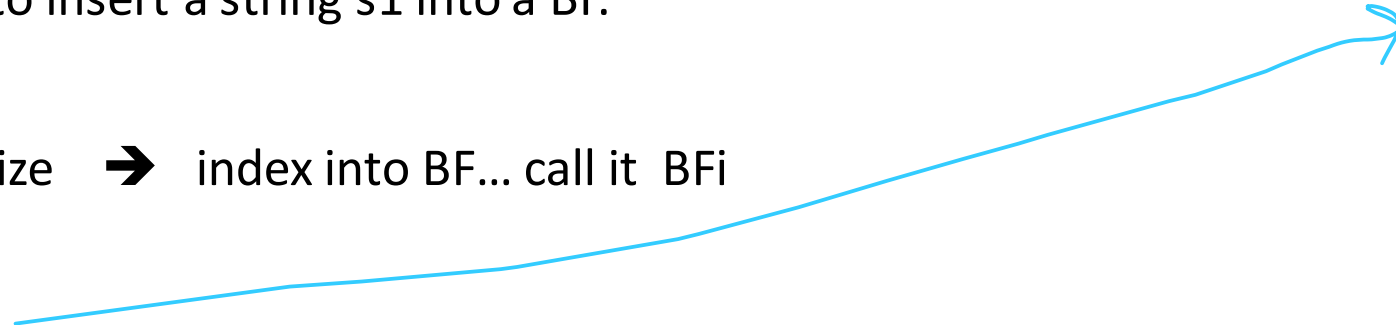
- Bloom filters are a probabilistic data structure.
- We say *probabilistic* because its result is not 100% certain.
 - Note that this does NOT mean its not deterministic 😊
- Return value of `Contains` answers the membership question:
 - True:
 - Element was *probably* added to the BF ➡ Potential false positive.
 - False:
 - Element was *definitely* not added to the BF

Bloom Filters

- True: Element was *probably* added to the BF → Potential false positive.
- False: Element was *definitely* not added to the BF
- Given the possibility of false positives, BFs cannot be used where a false positive cannot be tolerated.
- So why do we use them when we have other options such as Hash tables, BST, trie, etc.
 - They are extremely space efficient.
 - In other words, there is a space vs accuracy tradeoff
 - Note: accuracy tradeoff is only when returning true.

Bloom Filters

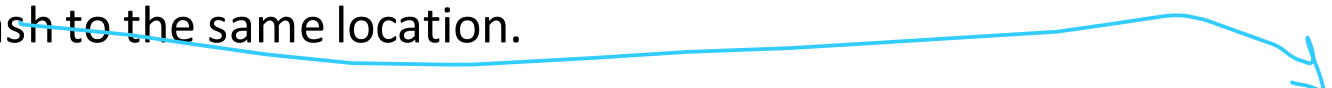
- Think of a BF as an array of Booleans (for now).
- Lets say we need to insert a string $s1$ into a BF.
- $\text{Hash}(s1) \% \text{BF size} \rightarrow \text{index into BF... call it } BFi$
- $\text{array}[BFi] = \text{true}$
- Now, to check if BF contains $s1$:
 - $\text{return array}[\text{Hash}(s1) \% \text{BF}]$
- Similar to hash tables But do u see any difference compared to a hash table ?



false
false
true
false
false
false
false

Bloom Filters

- In a hash table, we store the element (at a certain location, based on hashing, etc).
- But in BF, we do not store the element... we simply have a means of answering true or false.
 - We do that by turning on a bit at the location.
- But what about collisions now?
- We could have two elements that hash to the same location.
- $\text{array} [\text{Hash}(s1) \% \text{BF_size}] = \text{true}$
- And let's say we have another string $s2$ such that:
 - $\text{Hash}(s2) \% \text{BF_size} == \text{Hash}(s1) \% \text{BF_size}$
- Now, $\text{Contains}(s2)$ will return true, but we didn't add it to the BF.
 - That can, and does happen... this is the reason for false positives in BF.
 - We will talk about how to keep probability of false positives low.



false
false
true
false
false
false
false

Bloom Filters

- Now, the array in BF is usually not an array of Booleans, but an array of bits.
 - Why?
 - u can use an array of integers and access bits within them.
- Example:
 - Integer size = 32 bits. Lets call this `SIZE_INT`
 - Array of 100 integers will give me 3200 bits... so I can pretend that is an array of 3200 bits (actually, it is).
 - So, the `BF_size` in our formula would be 3200.
 - How do I turn on a bit, say, `BFi`
 - Or how do I check a bit's value.

Bloom Filters

Numbers are bit indices (0 thru 7,
for our example)

- Assume integer size = 8 bits. Lets call this `SIZE_INT`



- To access bit `BFi`:
 - First, we get the int that will contain bit `BFi`
 - `int RelevantInt = array [BFi / SIZE_INT]`
 - Then, from that int, we want to look at the relevant bit.
 - Bit index inside the `RelevantInt`:
 - `RelevantBit = BFi % SIZE_INT`
 - So, we really want the value of bit *RelevantBit* inside the int *RelevantInt*
 - Lets give some values to these variables:
 - `BFi = 0` → this implies we want the value of the first bit... easy... bit 0 of int at index 0.
 - RelevantInt* is `array [0 / SIZE_INT]`, which is `array [0 / 8]`, which is `array [0]`
 - RelevantBit* is `0 % SIZE_INT`, which is `0 % 8`, so we want bit 0
 - `BFi = 7` → this implies we want the value of the 7th bit... easy... bit 7 of int at index 0.
 - RelevantInt* is `array [7 / 8]`, which is `array [0]`
 - RelevantBit* is `7 % 8`, so we want bit 7

Bloom Filters

- `int RelevantInt = array [BFi / SIZE_INT]`
- `RelevantBit = BFi % SIZE_INT`

Numbers are bit indices (0 thru 7,
for our example)

- `BFi = 14`

- *RelevantInt* is array [14 / 8], which is array [1]
- *RelevantBit* is 14 % 8, so we want bit 6

- `BFi = 8`

- *RelevantInt* is array [8 / 8], which is array [1]
- *RelevantBit* is 8 % 8, so we want bit 0

- `BFi = 23`

- *RelevantInt* is array [23 / 8], which is array [2]
- *RelevantBit* is 23 % 8, so we want bit 7



- Now, what will the code look like to get a particular bit from an int?

Bloom Filters

- Getting a particular bit from an int:

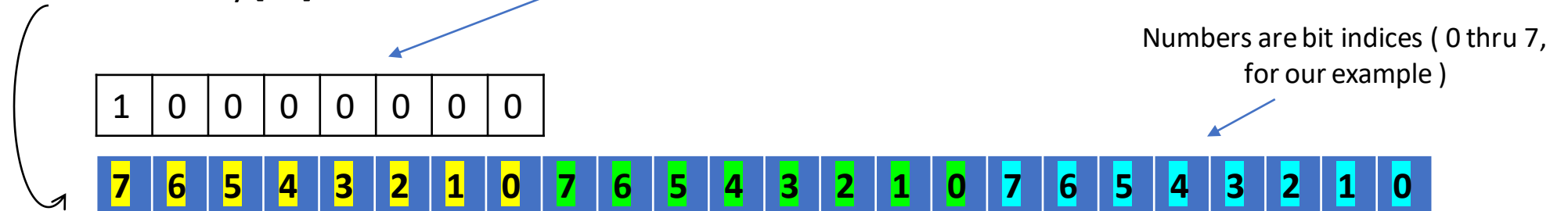
- BFi = 23

- RelevantInt* is $\text{array}[23 / 8]$, which is $\text{array}[2]$
- RelevantBit* is $23 \% 8$, so we want bit 7

$\text{array}[23 / 8] \ \& \ (0x1 \ll (23 \% 8))$

→ $\text{array}[2] \ \& \ (0x1 \ll 7)$

→ Bit 7 of $\text{array}[2]$



- Generic formula ? (next slide)

Bloom Filters

- Getting a particular bit from an int:

- $BFi = 23$

- *RelevantInt* is $\text{array}[23 / 8]$, which is $\text{array}[2]$
- *RelevantBit* is $23 \% 8$, so we want bit 7

- $\text{array}[23 / 8] \ \& \ (0x1 \ll (23 \% 8))$

→ $\text{array}[2] \ \& \ (0x1 \ll 7)$

→ Bit 7 of $\text{array}[2]$

- OR

- **Generic formula:**

- $\text{array}[BFi / \text{SIZE_INT}] \ \& \ (0x1 \ll (BFi \% \text{SIZE_INT}))$

- Of course, $BFi / \text{SIZE_INT}$ should yield a value within array bounds (may need to check value of BFi)

Bloom Filters

- There are ways to keep the probability of false positives low.
 - Remember, false positives come from collisions.
 1. Large array size (*large* compared to number of entries expected to be stored)
 2. Multiple hash functions, single array
 3. Multiple hash functions, multiple arrays
- 1. Large array size
 - This will reduce probability of collisions.

Bloom Filters

2. Multiple hash functions, single array

- We run each element thru k hash functions and turn on those bits.

- $H_1(s_1)$

- $H_2(s_1)$

- ...

- $H_k(s_1)$

1
0
1
0
0
0
0
1
0
0

- Assume function H_i contains the modulo BF_Size calculation inside it.

- What does implementation for `Contains(s1)` look like?

Bloom Filters

- $H_1(s_1)$
 - $H_2(s_1)$
 - ...
 - $H_k(s_1)$
-
- Contains (s1) will do:
 - $\text{array}[H_1(s_1)] == 1$
and $\text{array}[H_2(s_1)] == 1$
...
 - and $\text{array}[H_k(s_1)] == 1$

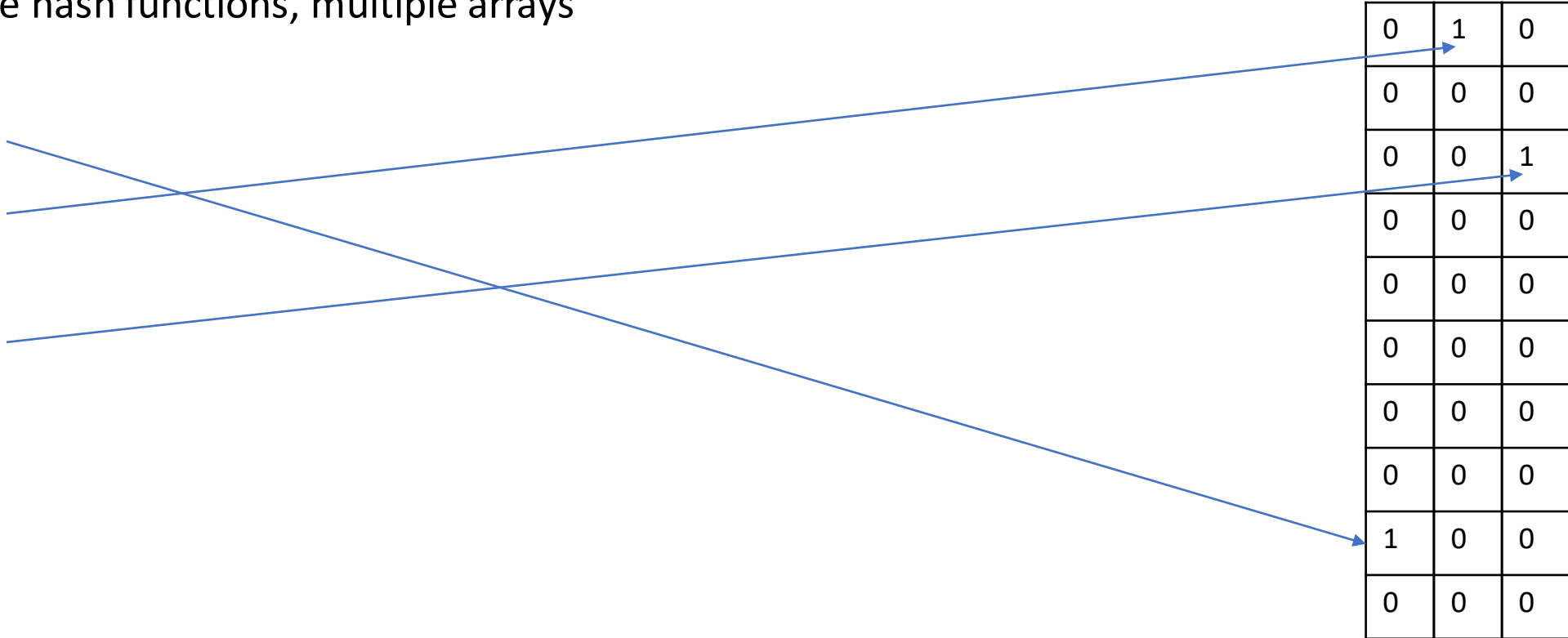
1
0
1
0
0
0
0
0
1
0

- All K entries need to be 1.
- Even if one of them is 0, Contains will return false (of course, will / should do short circuit evaluation).

Bloom Filters

3. Multiple hash functions, multiple arrays

- $H_1(s_1)$
- $H_2(s_1)$
- ...
- $H_k(s_1)$



0	1	0
0	0	0
0	0	1
0	0	0
0	0	0
0	0	0
0	0	0
1	0	0
0	0	0

Bloom Filters

- Deleting an entry in a BF.
- For deleting from a BF, could we do the following?
 - Compute the hash of that entry and set the corresponding bits to 0?
 - foreach hashFunction H_i where $i = 1$ to k
 - $\text{array}[H_i(s)] = 0$
 - Would this work?

Bloom Filters

- Deleting an entry in a BF.
- If we set corresponding bits to 0, then its possible we may remove some other elements from the BF. This would lead to false negatives, rendering the BF unusable.
- Deletion is not supported in the regular bloom filters.
 - Counting Bloom filters can address the deletion issue to some extent.

Bloom Filters

- Lets say we add a bunch of items to a BF, and then we realize we need a bigger one because the number of items is lot more than anticipated.
- What options do we have:
 - Resize the BF, like we did for hash tables?

Bloom Filters

- Lets say we add a bunch of items to a BF, and then we realize we need a bigger one because the number of items is lot more than anticipated.
- What options do we have:
 - Resize the BF, like we did for hash tables?
 - If we resize, we need to rehash and insert... but do we have the elements we inserted?
- Limitation: Size needs to be known when creating the BF.
 - Look up Scalable Bloom Filters.

Bloom Filters

- **Use case 1**

- You visit a site in your browser. And u r visiting it for the first time.
- You type in the url in ur browser, and go to that site.
- Lets pause here for a second and think about what the browser **could** be doing behind the scenes(I say could, because I am not making a claim that's what any particular browser **is** doing)
 - The browser would like to check that the site u intend to go to is NOT a malicious site.
 - The way it could do that is to check if that site exists in a set of known malicious sites.
 - If it does, the browser would warn u (or perhaps refuse to go that site, whatever).
 - If not, it's the happy path and browser shows u that page.
- Some details:
 - BF could be used for an initial check:
 - if BF returns true for membership (potential false positive), then a more definitive check may need to be done (which would be higher latency).
 - If BF returns false, then we are sure its not a site known to us as malicious.
- To summarize, the problem here is to **check membership** of a site in a given set of sites, in this case, a set of malicious sites.

Bloom Filters

- Use case 2

- You are a frequent visitor to a site that supplies various articles in ur area of interest.
- This site also sends u regular emails about articles of interest to u (or shows links to articles of interest when u visit that site).
- For these suggested articles to make sense (and not look spammy), one of the criteria to satisfy would be that “u should not have already read that article”.
- So, its likely the site would do the following before dishing up a list of suggested articles for u:
 - Generate a list of articles in ur areas of interest.
 - Show first N articles to you, as long as u have **not** already read those articles.
 - So this check is a membership check in an “*articles that have been read by you*” set.
 - Only if the article is a **not** in that set, they will add it to the suggested list.
- What about potential false positive?
 - Impact: They may not recommend an article that u haven’t read... that’s tolerable.
- To summarize, the problem here is to **check membership** of an article in a given set of articles, in this case, a set of articles already read by u.

LAB

Implement the following functions for your own bloom filter:

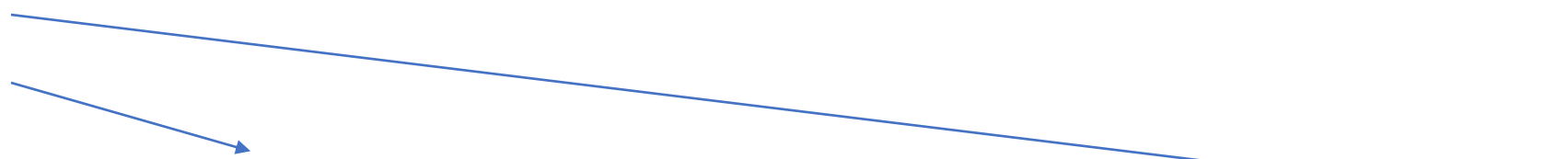
1. Add (string url)
 - This adds a given string to a bloom filter.
 2. Contains (string url)
 - Returns true if the given string is present in the bloom filter, else returns false
- You can use just **one hash function** for this lab.
 - For testing your functions, u can do something like:
 - Add (“abcdef”) , Add (“abcdef123”), Add (“abcdef456”) , Add (“abcdefxyz”)
 - Call Contains on the above added strings. You should get a true for these.
 - Call Contains on the something other than above added strings. You should get a false.

Counting Bloom Filters

- This is a slight variation of the regular bloom filter.
- Instead of keeping one bit for the result of a hash, we keep more than one bits (say, 4 bits).
 - All entries are initialized to 0

0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---

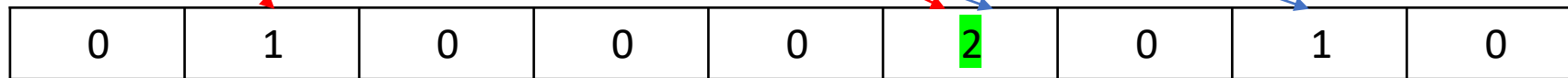
- Insertion into a counting BF:
 - Increment the value at the index produced by hash function.
 - Add (s1)
 - Add (s2)



0	1	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---	---

Counting Bloom Filters

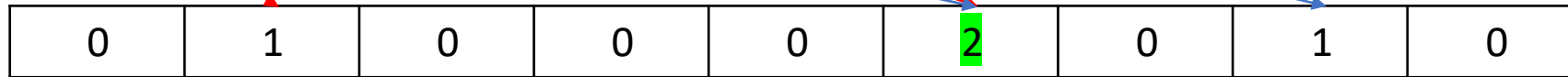
- Use multiple hash functions.
- Insertion into a counting BF:
 - Increment the value at the indices produced by hash functions.
 - Add (s1)
 - Add (s2)



- Note the collision.

Counting Bloom Filters

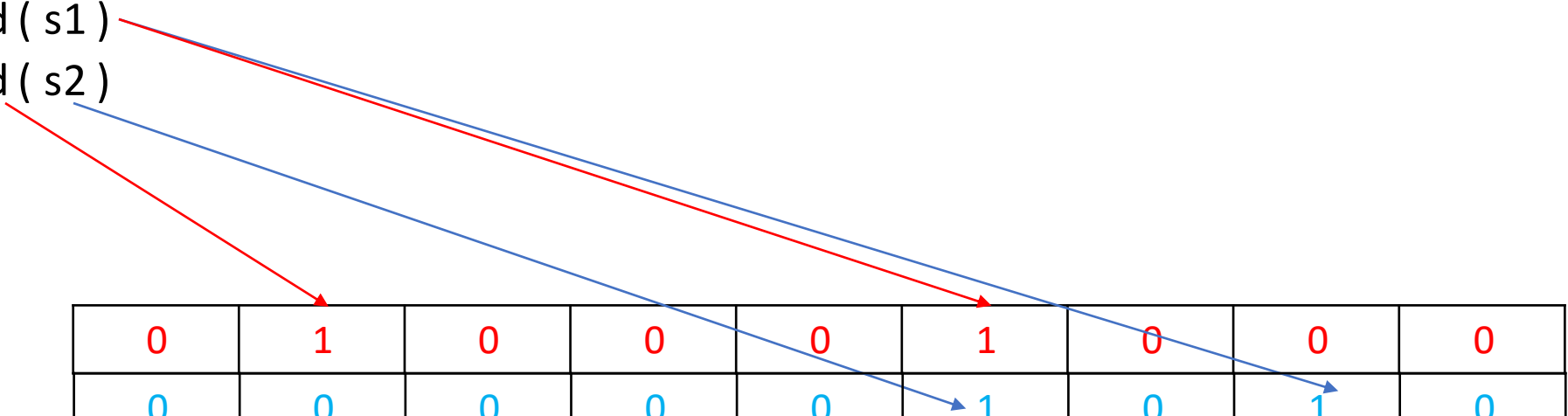
- Insertion into a counting BF:
 - Increment the value at the indices produced by hash functions.
 - Add (s1)
 - Add (s2)



- How does Contains work?
 - Contains (s1) returns
 - $\text{Min} (\text{BF} [\text{Hash}_1(s1)], \text{BF} [\text{Hash}_2(s1)])$
 - $\rightarrow \text{Min} (2, 1)$
 - $\rightarrow 1$

Counting Bloom Filters

- Insertion into a counting BF: Using two arrays (one for each hash function)
 - Increment the value at the indices produced by hash functions.
 - Add (s1)
 - Add (s2)



0	1	0	0	0	1	0	0	0
0	0	0	0	0	1	0	1	0

- How does Contains work?
 - Contains (s1) returns
 - $\text{Min} (\text{BF} [\text{Hash}_1(s1)], \text{BF} [\text{Hash}_2(s1)])$
 - $\rightarrow \text{Min} (1, 1)$
 - $\rightarrow 1$

Counting Bloom Filters

- Deletion from a counting BF:
 - Decrement the value at the index produced by hash function.
- Contains:
 - Will return true if the value at index is > 0
- We can use k hash functions here as well.

Counting Bloom Filters

- These take up more space than Bloom filters (e.g.: 4X more space if u use 4 bits at each entry).
- The bit counter should not overflow... so the number of bits is important and to be chosen appropriately.
- CBFs can overestimate the count, but will not underestimate.

Cardinality estimation

- Scenario:
 - You have a popular site.
 - You want to count the number of *unique* daily visitors to the site.
 - Or *unique* visitors to a page (or a bunch of pages).
 - If a visitor V1 visits more than once in a day, we want to count that as 1 unique visitor for that day.
- What are ur options?
 1. Could sort the visitor IDs and determine unique IDs.
 2. Could add IDs to a hash table and get a count.
- + The count would be accurate
- - This is a memory and compute intensive operation... think big data... think hundreds of millions or billions... u may even want this count over a period of a month, or a few months.
- Sites such as Facebook Microsoft, Amazon, etc. get a lot of visitors daily / monthly.

Cardinality estimation

But what do we really want when doing something like a unique visitor count.

- Its used for analytics.
 - Popularity (or not) of page content.
 - Bragging rights.
 - Perhaps helps in deciding ad display costs / potential returns for ad customers/
 - etc.
- For these use cases, is it really necessary to have an accurate count.
 - An accurate count would cost:
 - Lot of memory
 - Lot of compute hours
 - Wait many hours for a query run
 - Not be able to iterate frequently enough on what if scenarios.

Cardinality estimation

- Lets say I have 505,344,576 unique visitors in a week to my site.
- For my analysis purposes, do I need that accurate count, or would I be ok with knowing the number of unique visitors was about 500m, or even 490m(-3%) or 510m (+2%)
- In most cases, an approximate count works.
- And if I know the error margin, that also helps understand the possible range of the actual count.

Cardinality estimation

1. A simplistic way to count (with a huge variance, though).

Lets say we are given a number line with a marking from 0.0 to 1.0

And we need to divide that line evenly into 4 parts.



And now if we take the smallest number and invert that:

$$1 / 0.25 = 4 \quad \leftarrow \text{that's the number of divisions.}$$

If we divide the line into 10 parts, we will have 0.10, 0.20, ... 0.90 and 1.00

Take smallest number and invert:

$$1 / 0.10 = 10 \quad \leftarrow \text{that's the number of divisions.}$$

Cardinality estimation

Now, imagine:

- We have a container
 - It has a lot of numbers in it (> 0 and ≤ 1.0)
 - The numbers are fairly evenly distributed over the range.
 - The same number can occur more than once.
 - and I need to count how many *unique* numbers are present in the container.
-
- Extending the thought from the earlier slide, we could do the following:
 - Linear scan through all the numbers.
 - Keep track of the smallest number you see. Call it *min*
 - Now compute $1/\text{min}$ to get the count of the numbers
 - Example:
 - If the smallest seen was 0.0012, then $1/0.012 = 833$
 - If the smallest seen was 0.00002, then $1/0.012 = 50,000$

Cardinality estimation

So we were able to get a count of how many unique numbers are in the container.

We did this with:

- A linear scan and kept track of the smallest number we saw.
 - Time: $O(N)$
 - Space: $O(1)$
- Then did the division
 - Time: $O(1)$

That seems great, but what's the catch to using this in the real world on real data?

Cardinality estimation

We cannot use this in the real world on real data, at least not in the simplistic way we talked about so far.

Real world data:

- not going to be evenly distributed.
- it could have outliers
- it could also have some instances of bad data.

So if we simply take the smallest number and invert, we could be off in our estimation by a lot.

Cardinality estimation

2. Another “approximate counting” idea.

Lets look at the bit pattern of 3 bit numbers.

If u look at a bit pattern for a number (in binary):

- What is the probability of its LSB being 0?

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Cardinality estimation

2. Another “approximate counting” idea.

Lets look at the bit pattern of 3 bit numbers.

If u look at a bit pattern for a number (in binary):

- What is the probability of its LSB being 0?
 - $\frac{1}{2}$
- What is the probability of the bit to immediate left of LSB being 0?

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Cardinality estimation

2. Another “approximate counting” idea.

Lets look at the bit pattern of 3 bit numbers.

If u look at a bit pattern for a number (in binary):

- What is the probability of its LSB being 0?
 - $\frac{1}{2}$
- What is the probability of the bit to immediate left of LSB being 0?
 - Also $\frac{1}{2}$
- What is the probability of LSB and the bit to immediate left of LSB being 0?

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Cardinality estimation

2. Another “approximate counting” idea.

Lets look at the bit pattern of 3 bit numbers.

If u look at a bit pattern for a number (in binary):

- What is the probability of its LSB being 0?
 - $\frac{1}{2}$
- What is the probability of the bit to immediate left of LSB being 0?
 - Also $\frac{1}{2}$
- What is the probability of LSB and the bit to immediate left of LSB being 0?
 - $\frac{1}{2} * \frac{1}{2} = \frac{1}{4}$
- What is the probability of LSB and the two bits to immediate left of LSB being 0?
 - $\frac{1}{2} * \frac{1}{2} * \frac{1}{2} = \frac{1}{8}$

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Cardinality estimation

- What is the probability of LSB and the two bits to immediate left of LSB being 0?
 - $\frac{1}{2} * \frac{1}{2} * \frac{1}{2} = 1/8$

Looking at this another way, we can say:

Given a lot of numbers:

- a number with 3 consecutive 0s would occur once every 8 numbers.
 - a number with 4 consecutive 0s would occur once every 16 numbers.
 - a number with k consecutive 0s would occur once every 2^k numbers.
-
- And turning that around, we can say that, given a lot of numbers:
 - We will do a linear scan thru the numbers and find the longest sequence of consecutive 0s (say, K)
 - Cardinality then would be 2^k
-
- Next: What are the weak points of this approach?

0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

Cardinality estimation

1. Our count estimation is only by powers of 2.
 - As k increases, the error in the estimation would go up a lot.
 1. $2^{20} = 1000,000$
 2. $2^{21} = 2000,000$
 3. $2^{22} = 4000,000$
2. And if there was even one number with a lot of consecutive 0s, that would throw off our estimate by a lot. So, the variance is high.

How can we improve this?

Cardinality estimation

Before we talk about improving, let's say the following:

The input numbers that we have been talking about so far... they actually are hash values of the input numbers rather than the numbers themselves.... I just didn't mention it earlier 😊

So, now our improvement could be:

instead of one hash function to compute hash values of input, we use multiple hash functions.

This means, we will have:

ConsecutiveZeroCount_1 = Consecutive 0s in H_1 (number)

ConsecutiveZeroCount_2 = Consecutive 0s in H_2 (number)

ConsecutiveZeroCount_3 = Consecutive 0s in H_3 (number)

AvgConsecutiveZeroCount =

$(\text{ConsecutiveZeroCount}_1 + \text{ConsecutiveZeroCount}_2 + \text{ConsecutiveZeroCount}_3) / 3.0$

So, count estimation = $2^{\text{AvgConsecutiveZeroCount}}$

Cardinality estimation

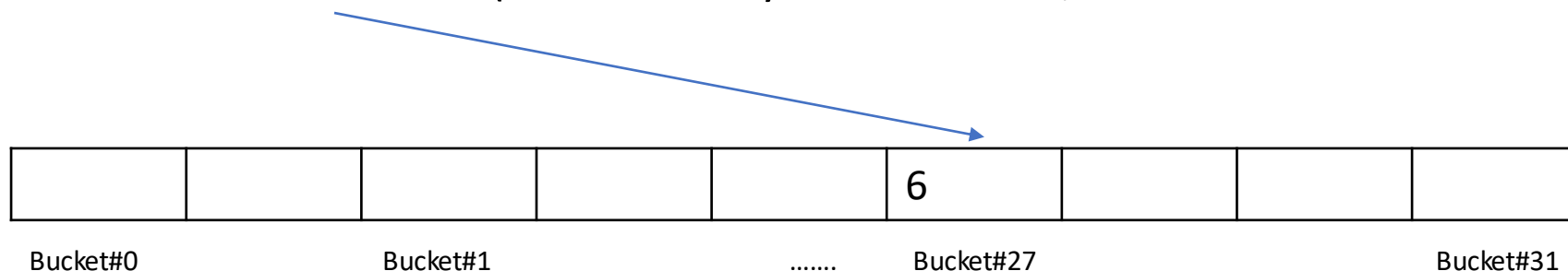
- + we have a better estimate now, compared to using just one hash function.
- - applying multiple hash functions to so many input numbers (think huge data sets) is costly.
- To address the con above, we can look at stochastic averaging.

Cardinality estimation

We will go back to using only one hash function.

But this is what we will do with that one hash value of a number:

- Let size of hash value = B bits
- Look at the left most K bits (say K = 5) of the hash value.
 - Let the hash value be 11011011000000 → decimal 27
 - We use this value of 27 to tell us which bucket number to use (total # of buckets $2^5 = 32$)
- We will use the rightmost B – K bits to check for consecutive zeros
 - So, consecutiveZeroCount = Num consecutive zeroes in rightmost B – K bits
- We will update bucket #27 as follows:
 - bucket #27 = max (value already in bucket #27, consecutiveZeroCount)



Cardinality estimation

When all input data is processed, we take the numbers in all the buckets and find their average.

Average consecutiveZeroCount = (Sum numbers in all buckets) / numberOfBuckets

$$\text{count estimation} = \text{numberOfBuckets} * 2^{\text{AvgConsecutiveZeroCount}}$$

And based on some statistical analysis, there is a constant to correct an upward estimation bias.

So, we have:

$$\text{count estimation} = \text{CONSTANT} * \text{numberOfBuckets} * 2^{\text{AvgConsecutiveZeroCount}}$$

Next: amount of memory used for this.

Cardinality estimation

If numberOfBuckets = 2048

And size of each bucket = 5 bits

- What's the max value we can hold in 5 bits?
- ➔ that's how many 0s we can count in each bucket

Total memory

= $2048 * 5$

= approx. 10,000 bits

= little over 1Kb of memory.

Cardinality estimation

This estimation can be parallelized:

- we can partition the input across multiple machines
- Run this computation on partitioned data.
 - of course, use the same hash function and same number and size of buckets.
- At the end, merge the results as follows:
 - For each bucket X
 - Final bucket #X value = $\max(\text{bucket \#X value across all the partitions})$
- This cardinality estimation algorithm is known as *SuperLogLog*.
- A slight modification of this is *HyperLogLog*.
 - Uses geometric mean instead of arithmetic mean for determining AvgConsecutiveZeroCount