# 2/20 - Bloom Filter - Background

**Background**

**Design Problem :**
**Suppose you have a huge dataset stored in your persistent storage ( Disk or DB ). You want quick check, does cityname:delhi present in data set or not?**

Now, you want to deep dive on possible solution.

The First solution , I can see that we can check each element stored in disk or DB to verify that delhi is part of this dataset or not? And this is called Linear Search of your data.

**This approach will work and able to find your problem solution, but this approach is so slow that it is almost not a solution at all.**

**Can we do better?**

My answer is YES.

build an index on each data file and store it in a separate index file. This index can map each record ID to its offset in the data file. Each index file will be sorted on the record ID. Now, if we want to search an ID in this index, the best we can do is a **Binary Search**.

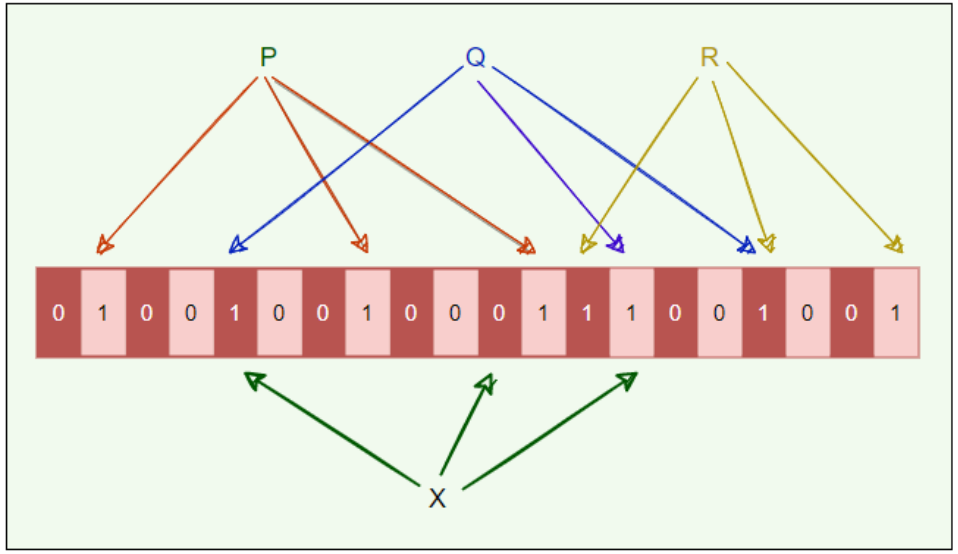
So, it is far better than Linear Search approach, but it also have some disadvantage.

1. You need additional indexing complex system.
2. You still need Disk IO operation to verify the existence. So, this has also penalty over performance.

So, still not the best, right? Can we do better?

YES, we can do better which is of constant time using Bloom Filter. This is even better for HashMap and Trie like system.

So, in principal bloom filter like HashMap but it don't require storage like HashMap, and that is the best part of Bloom Filter. Many standard solution using this Bloom Filter in Distributed system.




A Bloom filter consisting of 20 bits.

# System Design Pattern : Bloom Filter

Some interesting properties of bloom filter

- ✓ Unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements.
- ✓ Adding an element never fails. However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result.
- ✓ Bloom filters never generate false negative result, i.e., telling you that a username doesn't exist when it exists.
- ✓ Deleting elements from filter is not possible because, if we delete a single element by clearing bits at indices generated by k hash functions, it might cause deletion of few other elements. Example – if we delete "geeks" (in given example below) by clearing bit at 1, 4 and 7, we might end up deleting "nerd" also Because bit at index 4 becomes 0 and bloom filter claims that "nerd" is not present.

**Working of Bloom Filter**
An empty bloom filter is a **bit array** of **m** bits, all set to zero, like this –

We need **k** number of **hash functions** to calculate the hashes for a given input. When we want to add an item in the filter, the bits at k indices h1(x), h2(x), … hk(x) are set, where indices are calculated using hash functions.
Example – Suppose we want to enter "geeks" in the filter, we are using 3 hash functions and a bit array of length 16, all set to 0 initially.

h1("CAT") % 16 = 5
h2("CAT") % 16 = 6

h1("DOG") % 16 = 7
h2("DOG") % 16 = 10

h1("GOD") % 16 = 7
h2("GOD") % 16 = 4

For Implementation, please check this link
https://github.com/abmishra1234/4AM_Club_Coding/blob/main/Bloom_Filter.cpp

A - 0, B - 1, C - 2, ..... Z - 25

Hash 1:CAT = 2 + 0 + 19 = 21 % 16 = 5
DOG = 3 + 14 + 6 = 23 % 16 = 7
GOD = 6 +14+ 3 = 23 % 16 = 7

[ 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

CAT DOG GOD

Hash 2:CAT = 2*0 + 0*1 + 19*2 = 38 % 16 = 6
DOG = 3*0 + 14*1 + 6*2 = 26 % 16 = 10
GOD = 6*0 +14*1 + 3*2 = 20 % 16 = 4

[ 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0]

GOD CAT DOG

Bloom Filters

[ 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0]

[ 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0]

CAT (5,6)
DOG (7,10)
GOD (7,4)