System Design Pattern : Introduction

These patterns refer to common design problems related to distributed systems and their solutions. Knowing these patterns is very important as they can be applied to all types of distributed systems and are very handy, especially in a system design discussion.

Here is the list of patterns we will be discussing:

1. Bloom Filters
2. Consistent Hashing
3. Quorum
4. Leader and Follower
5. Write-ahead Log
6. Segmented Log
7. High-Water mark
8. Lease
9. Heartbeat
10. Gossip Protocol
11. Phi Accrual Failure Detection
12. Split-brain
13. Fencing
14. Checksum
15. Vector Clocks
16. CAP Theorem
17. PACELEC Theorem
18. Hinted Handoff
19. Read Repair
20. Merkle Trees

System Design Pattern : Bloom Filter

Scenario:

Suppose you have a system where you want to implement a filter which will help user to identify that selected Username already exist in system or this name is available to use?

So, for this kind of problem, multiple approach is possible

Approach 01 : Linear Search – Meaning you must search the complete dataset one by one and make sure that this selected user exist or not? If Exist return so otherwise return don't exist.

The problem in this approach that your solution is not at all efficient and as your dataset will grow your new user feel lots of latency in the system. It would degrade the overall experience of the system.

Approach 02: Binary Search – Suppose you have your datasets arranged in some incremental order on the basis alphabets, just like a dictionary. So, in this case you might have easy to use Binary search to explore certain string in complete data sets. This approach is far batter approach than linear search.

This could be one probable solution but still not at par, because in this case also we must access the IO operation , meaning hitting the storage which is not much performant operation.

This approach is much faster than first approach.

Can we do better than this…

Approach 03: Hash Table or Trie Data structure – In this approach, in general or averagely HashMap will search things very fast , kind of constant time, And Trie search elements in O(length Of string to be searched). Both approach are fast and better than earlier two approach. Also, it is providing efficient solution to your problem. But here is one problem they are introducing while implementing this, and that is they need memory space. As you grow in number it will start becoming bulky and more complex if you plan to create some scaling especially horizontal one.

So even though it seems solution but for system architect this is not the perfect solution of this problem.

Can we have any thing better than this for this problem?

Contd to next section

So, as an architect we have to analyse our requirement again.

What we see here in our requirement,

- Need of some system which is surely real time
- **Consistent,** meaning even though system is failing in some scenario , still it should fail false positive not the False negative.

  [meaning, if system is saying that something exist and it won't it is ok for us, but not the otherwise. Meaning, if some thing exist but your service saying don't exist (False Negative)]
- Your **system don't require any IO call for this kind of filtering**. IO operation is completely avoided
- **SPF is ok in this case with proper backup standby system**, but distributed case in this scenario seems too much for us.

A Bloom filter is a probabilistic data structure present in many common applications. Its purpose is answering the question: "is this item in the set?" very fast and not using a lot of space. The answers can be **NO**, or **MAYBE YES.**

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 –

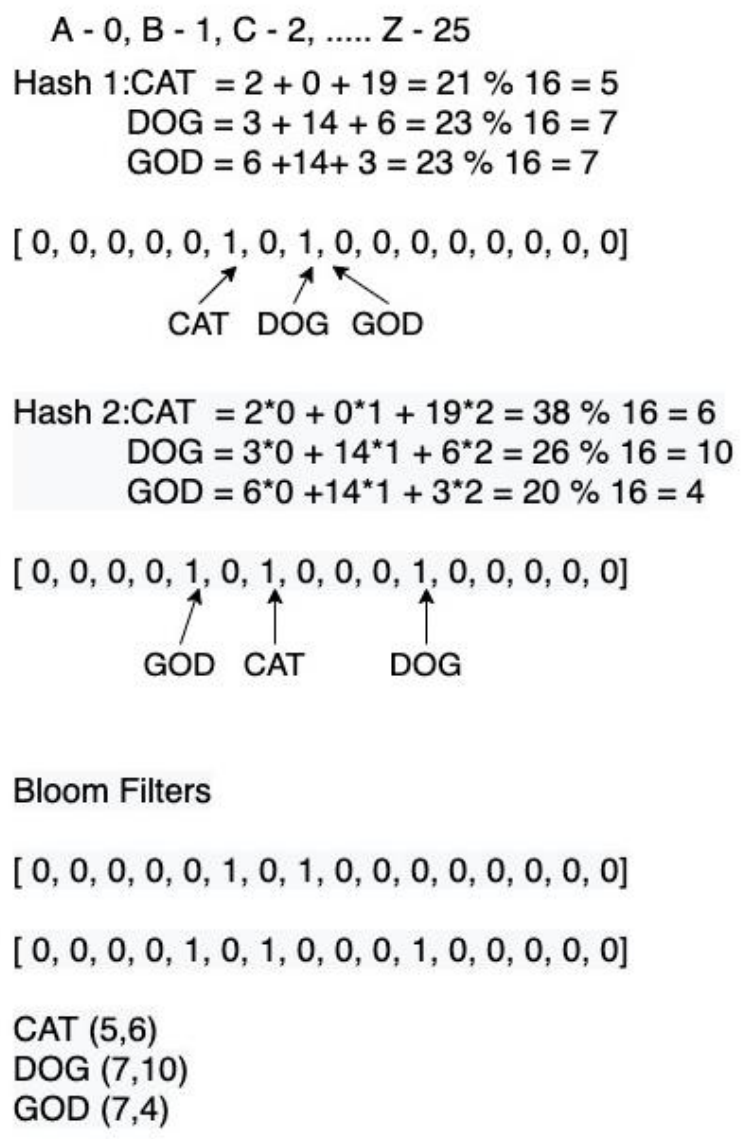
We need **k** number of **hash functions** to calculate the hashes for a given input. When we want to add an item in the filter, the bits at k indices h1(x), h2(x), … hk(x) are set, where indices are calculated using hash functions. Example – Suppose we want to enter "geeks" in the filter, we are using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

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]

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)