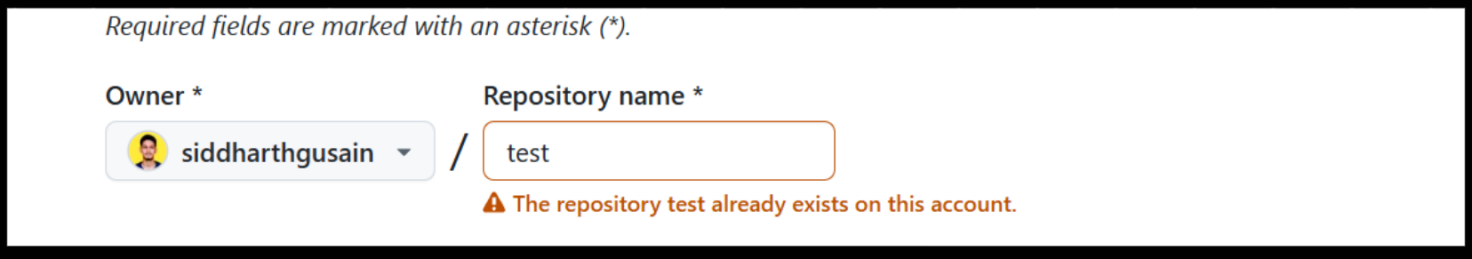
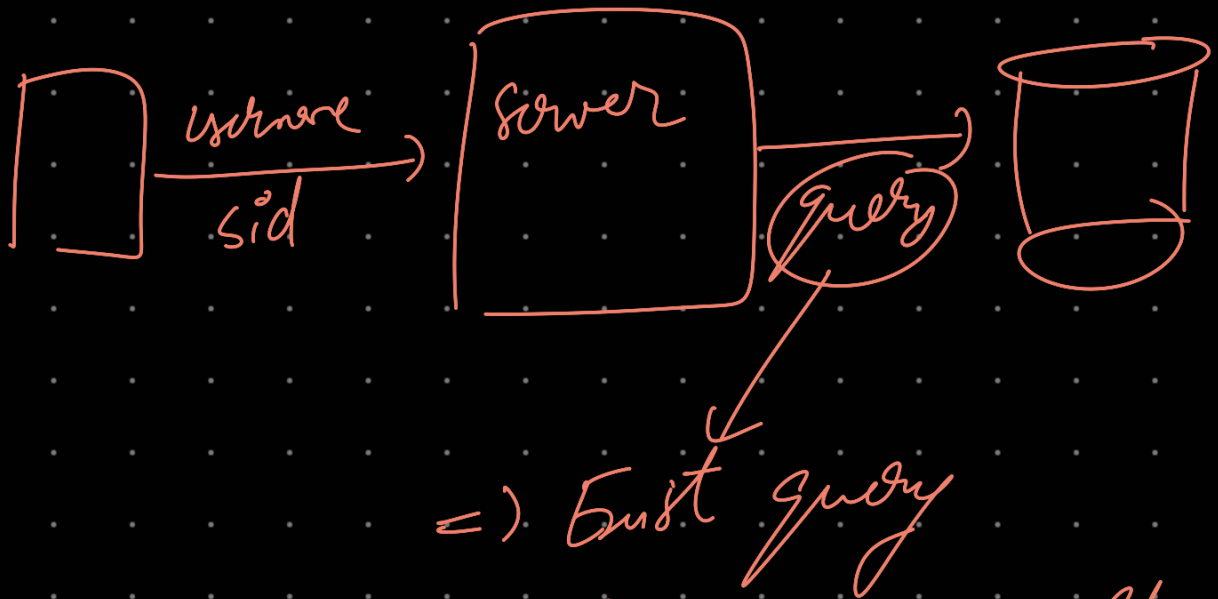
[Bloom filters]

* Let say we want to do existance check.
  E.g -) A username exists.?



E.g -) Github repo name exits
-) Instagram username already taken or not?



=) Exist query

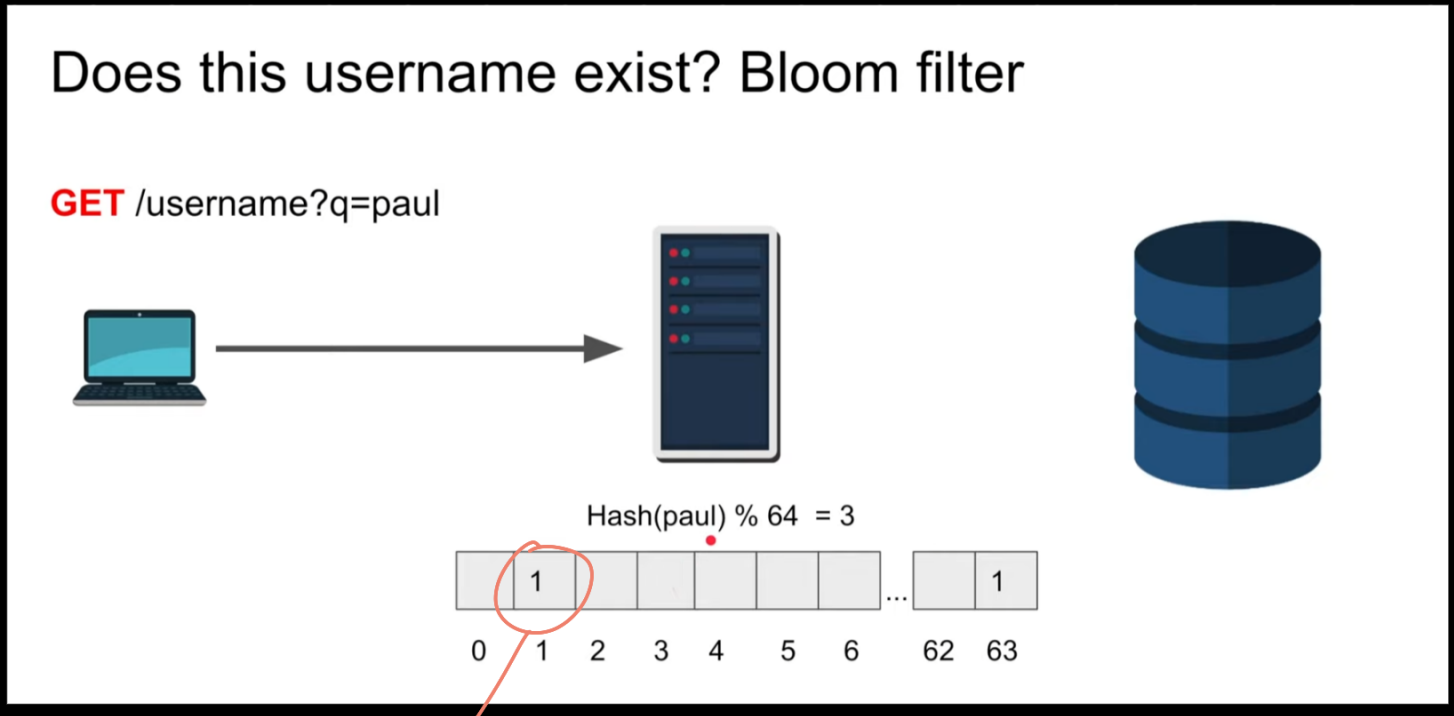
These type of query results could be empty

* In case of million users, using a SQL query which check existence is expensive.

* Lets say there is Indexes → still slow

⇓

To optimize we can use Cache



If, bit = 0 → not present for sure
→ No need to do

DB query

If, bit = 1 ⟶ possibility of existence

As, Multiple username can hashed to same array index, bit = 1 gives possibility of existence.

A **Bloom filter** is a space-efficient probabilistic data structure used to test whether an element is a member of a set. It can provide **two types of results**:

1. **True Positive:** If it says the element is in the set, it *might* actually be in the set.
2. **False Positive:** It may sometimes say an element is in the set when it's not.
3. **True Negative:** If it says the element is *not* in the set, it's guaranteed to be correct.

In short, Bloom filters are:

- **Memory-efficient:** Use less space compared to traditional data structures like hash tables.
- **Fast:** They offer constant-time complexity for insertions and lookups.
- **Probabilistic:** Acceptable for applications where occasional false positives are okay but false negatives are not.

🛑 Limitations of Bloom Filters

1. False Positives

Bloom Filters can produce false positives, meaning they may incorrectly indicate that an element is present in the set when it is not.

> **Example:** Consider a scenario where a non-existent key is checked against a Bloom Filter. If all the hash functions map to bits that are already set to **1**, the filter falsely signals the presence of the key.

Such false positives can lead to unnecessary processing or incorrect assumptions about data.

For instance, in a database system, this might trigger unnecessary cache lookups or wasted attempts to fetch data that doesn't actually exist.

The likelihood of false positives can be reduced by choosing an optimal size for the bit array and an appropriate number of hash functions, but they can never be completely eliminated.

2. No Support for Deletions

Standard Bloom Filters do not support element deletions. Once a bit is set to 1 by adding an element, it cannot be unset because other elements may also rely on that bit.

This limitation makes Bloom Filters unsuitable for dynamic sets where elements are frequently added and removed.

Variants like the **Counting Bloom Filter** can allow deletions by using counters instead of bits, but this requires more memory.

3. Limited to Set Membership Queries

Bloom Filters are specifically designed to answer set membership queries. They do not provide information about the actual elements in the set, nor do they support complex queries or operations beyond basic membership checks.

Example: If you need to know the details of an element (e.g., full information about a user ID), you would need another data structure in addition to the Bloom Filter.

4. Not Suitable for Exact Set Membership

Bloom Filters are probabilistic, meaning they cannot provide a definite "yes" answer (only a "probably yes" or "definitely no").

For applications requiring exact membership information, Bloom Filters are not suitable. Other data structures like hash tables or balanced trees should be used instead.

5. Vulnerable to Hash Collisions

Hash collisions are more likely as the number of elements in the Bloom Filter grows. Multiple elements can end up setting or relying on the same bits, increasing false positives.

As hash collisions accumulate, the filter's effectiveness decreases. With a high load factor, the filter may perform poorly and become unreliable.

The use of additional hash functions can help reduce collisions, but increasing the number of hash functions also increases the complexity and the memory requirements.

1. **Web Caching:** Check if a URL is already in cache.
2. **Databases:** Efficiently filter queries (e.g., check if a key exists in a distributed database).
3. **Networking:** Avoid duplicate processing of packets.
4. **Blockchain:** Bitcoin uses Bloom filters for lightweight client queries.
5. **Spell Checking:** Quickly check if a word is in the dictionary.