What are Bloom Filters and Where are they Used?

Explained with Real-World Examples

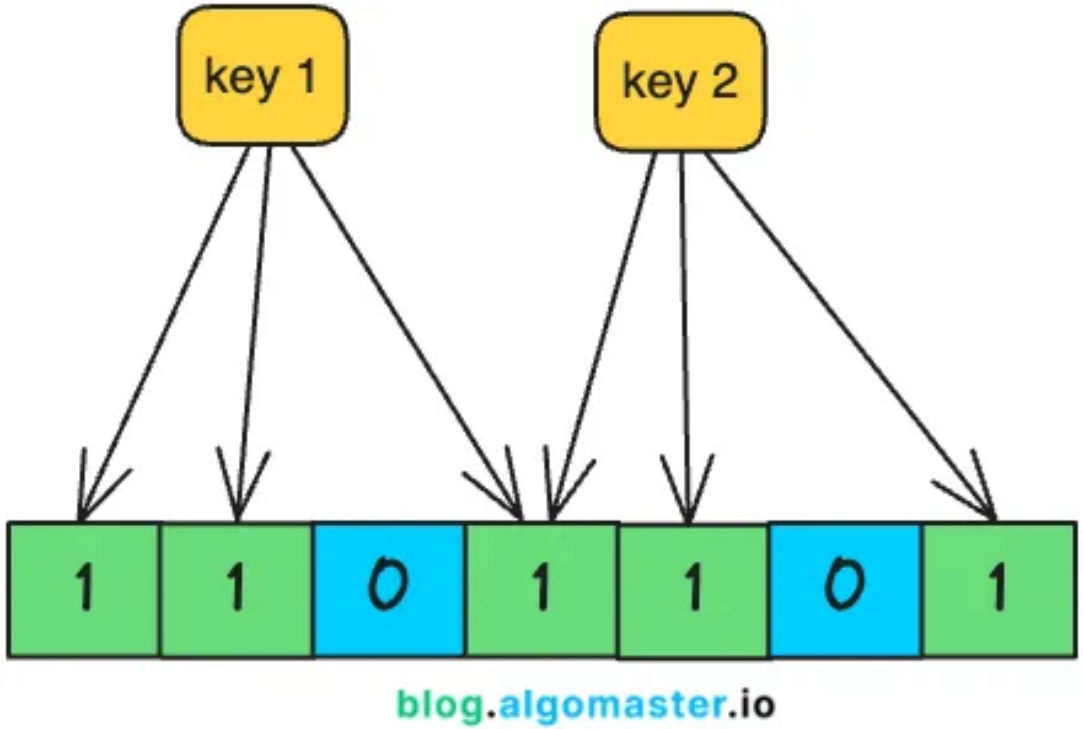
ASHISH PRATAP SINGH

NOV 24, 2024

Have you ever wondered how **Netflix** knows which shows you've already watched? Or how **Amazon** avoids showing you products you've already purchased?

Using a traditional data structure like a **hash table** for these checks could consume significant amount of **memory**, especially with millions of users and items.

Instead, many systems rely on a more efficient data structure—a **Bloom Filter**.



In this article, we will learn what a bloom filter is, how it works, how to implement it in code, it's real-world applications and limitations.

If you're enjoying this newsletter and want to get even more value, consider becoming a [paid subscriber](#).

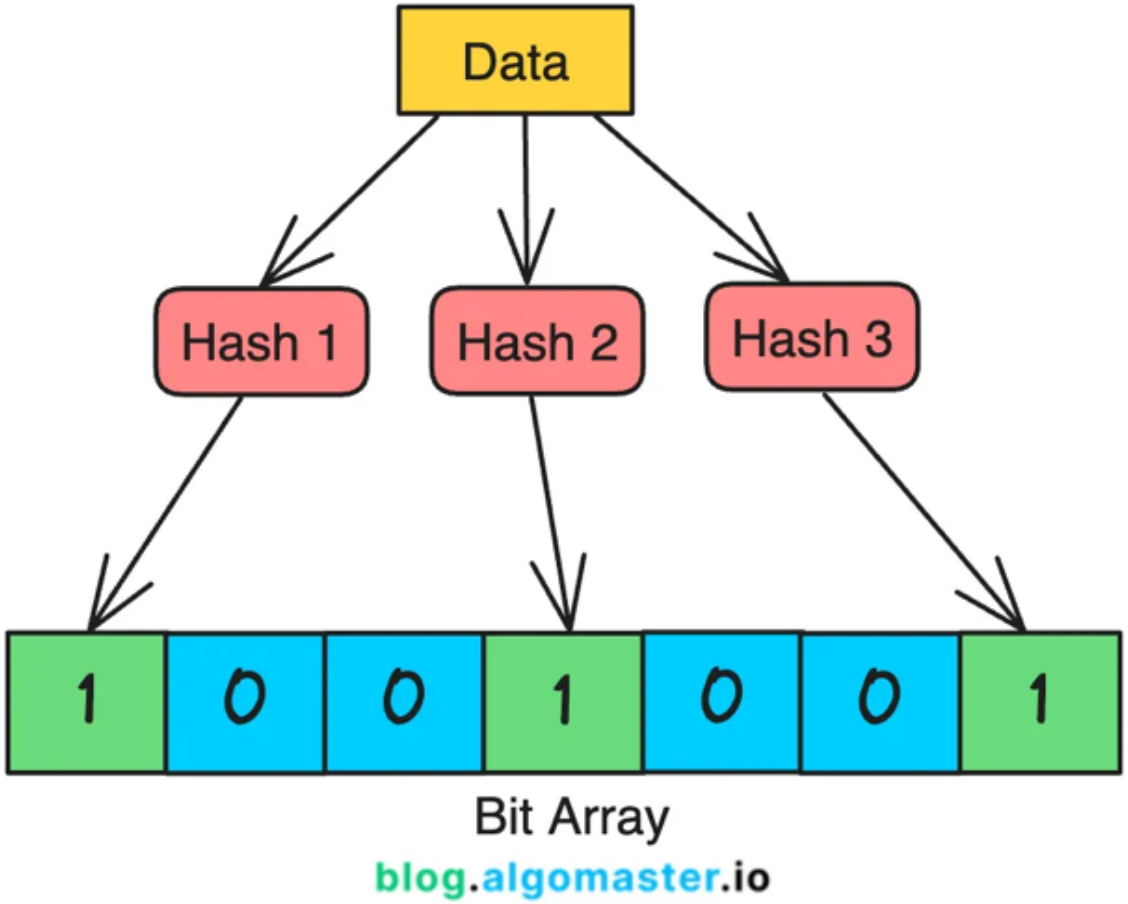
As a paid subscriber, you'll unlock all **premium articles** and gain full access to all premium courses on algomaster.io.

🤔 What is a Bloom Filter?

A **Bloom Filter** is a **probabilistic data structure** that allows you to quickly check whether an element might be in a set.

It’s useful in scenarios where you need **fast lookups** and don’t want to use a large amount of memory, but you’re okay with occasional **false positives**.

🧩 Key Components of a Bloom Filter:



1. **Bit Array**: The Bloom Filter consists of a bit array of a fixed size, initialized to all zeros. This array represents whether certain elements are in the set.
2. **Hash Functions**: To add or check an element, a Bloom Filter uses multiple hash functions. Each hash function maps an element to an index in the bit array.

⚙️ How Does a Bloom Filter Work?

A Bloom filter works by using multiple hash functions to map each element in the set to a bit array.

1. Initialization:

- A Bloom filter starts with an empty bit array of size `m` (all bits are initially set to 0).
- It also requires `k` independent hash functions, each of which maps an element to one of the `m` positions in the bit array.

2. Inserting an Element:

- To insert an element into the Bloom filter, you pass it through each of the `k` hash functions to get `k` positions in the bit array.
- The bits at these positions are set to 1.

3. Checking for Membership:

- To check if an element is in the set, you again pass it through the `k` hash functions to get `k` positions.
- If all the bits at these positions are set to 1, the element is considered to be in the set (though there's a chance it might be a false positive).

- If any bit at these positions is 0, the element is definitely not in the set.

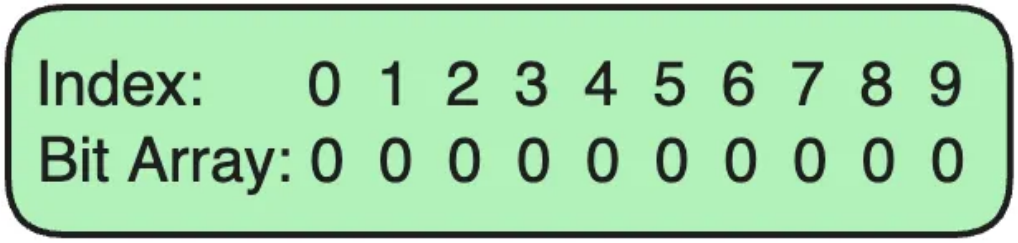
🔎 Example: Using a Bloom Filter for URL Checking

Imagine you're building a web crawler that needs to keep track of which URLs it has already visited.

Instead of storing every URL (which would require a lot of memory), you decide to use a Bloom Filter.

Step 1: Set Up the Bloom Filter

- **Initialize a Bit Array**: Let's assume our Bloom Filter uses a bit array of size **10**, initially all set to 0.



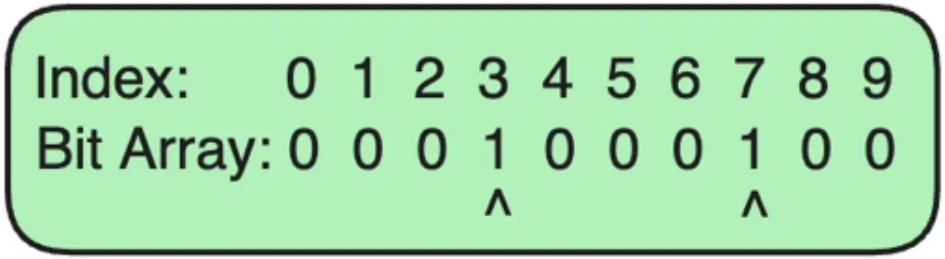
- **Choose Hash Functions**: We'll use two hash functions in this example. These hash functions take an input (like a URL) and output an index in the bit array.

Step 2: Adding a URL to the Bloom Filter

Suppose we want to add the URL `example.com` to our Bloom Filter.

1. **Hash Function 1** generates an index of 3 for `example.com`.
2. **Hash Function 2** generates an index of 7 for `example.com`.

We set the bits at indices 3 and 7 in the bit array to 1.

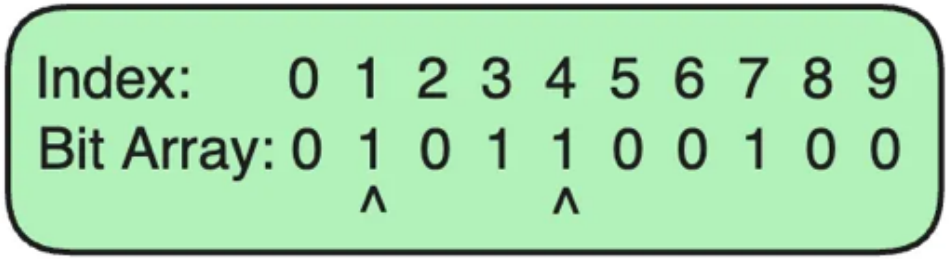


Step 3: Adding Another URL

Now, let's add another URL, `algomaster.io`.

1. **Hash Function 1** generates an index of 1 for `algomaster.io`.
2. **Hash Function 2** generates an index of 4 for `algomaster.io`.

We set the bits at indices 1 and 4 in the bit array to 1.

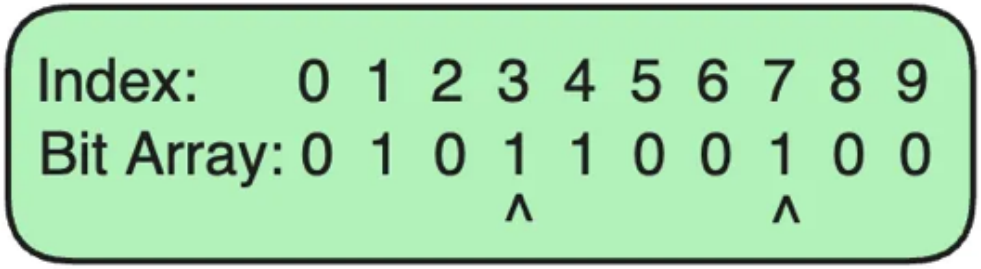


Step 4: Checking for a URL in the Bloom Filter

Suppose we want to check if `example.com` is already in the Bloom Filter.

1. **Hash Function 1** generates index 3 for `example.com`.
2. **Hash Function 2** generates index 7 for `example.com`.

Since both bits at indices 3 and 7 are set to 1, we can say that `example.com` **is probably in the set** (there's a small chance of a false positive).

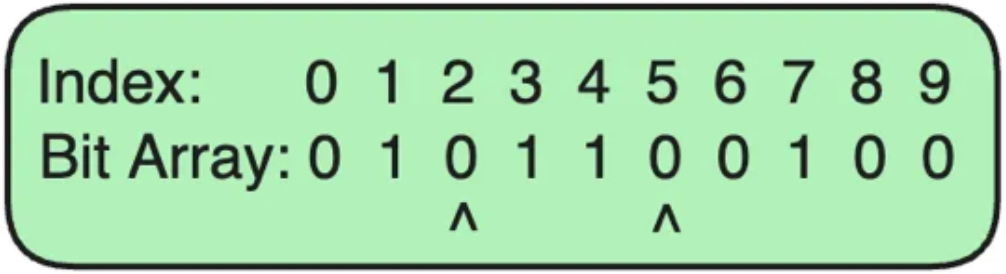


Step 5: Checking for a Non-Existent URL

Now, let's check if `nonexistent.com` is in the Bloom Filter.

1. **Hash Function 1** generates index 2 for `nonexistent.com`.
2. **Hash Function 2** generates index 5 for `nonexistent.com`.

Since the bits at indices 2 and 5 are both 0, we can confidently say that `nonexistent.com` **is not in the set.**



💻 Code Implementation (Java)

```
public class BloomFilter {
    private final BitSet bitArray;
    private final int size;
    private final Function<String, Integer>[] hashFunctions;
```

```
    public BloomFilter(int size, Function<String, Integer>... hashFunctions) {
        this.size = size;
        this.bitArray = new BitSet(size);
        this.hashFunctions = hashFunctions;
    }

    // Method to add an element to the Bloom Filter
    public void add(String item) {
        for (Function<String, Integer> hashFunction : hashFunctions) {
            int index = Math.abs(hashFunction.apply(item) % size);
            bitArray.set(index);
        }
    }

    // Method to check if an element is possibly in the Bloom Filter
    public boolean mightContain(String item) {
        for (Function<String, Integer> hashFunction : hashFunctions) {
            int index = Math.abs(hashFunction.apply(item) % size);
            if (!bitArray.get(index)) {
                return false; // If any bit is 0, the item is definitely not in the
set
            }
        }
        return true; // All bits are 1, so the item is probably in the set
    }
}
```

Explanation

1. **BitSet**: Java's `BitSet` is used for the bit array to efficiently store and manipulate bits.
2. **Hash Functions**: The code uses two simple hash functions. You can add more complex ones for better distribution.
3. **add(String item)**: This method takes an item, applies each hash function, and sets the corresponding bit in the bit array.
4. **mightContain(String item)**: This method checks if an item might be in the set by testing if all corresponding bits are 1.
    - If any bit is 0, the item is **definitely not in the set**.
    - If all bits are 1, the item is **probably in the set** (with a small chance of a false positive).

🌎 Real-World Applications of Bloom Filters

Bloom Filters are widely used in real-world applications where space efficiency and speed are essential, and occasional false positives are acceptable.

Here are some common scenarios where Bloom Filters are employed:

1. Web Caching

Problem: Web servers often cache frequently accessed pages or resources to improve response times. However, checking the cache for every resource could become costly and slow as the cache grows.

Solution: A Bloom Filter can be used to quickly check if a URL might be in the cache. When a request arrives, the Bloom Filter is checked first. If the Bloom Filter indicates the URL is “probably in the cache,” a cache lookup is performed.

If it indicates the URL is “definitely not in the cache,” the server skips the cache lookup and fetches the resource from the primary storage, saving time and resources.

2. Spam Filtering in Email Systems

Problem: Email systems need to filter out spam emails without constantly checking large spam databases.

Solution: A Bloom Filter can store hashes of known spam email addresses. When a new email arrives, the Bloom Filter checks if the sender's address might be in the spam list.

This allows the email system to quickly determine whether an email is likely to be spam or legitimate.

3. Databases

Problem: Databases, especially distributed ones, often need to check if a key exists before accessing or modifying data. Performing these checks for every key directly in the database can be slow.

Solution: Many databases, such as **Cassandra**, **HBase**, and **Redis**, use Bloom Filters to avoid unnecessary disk lookups for non-existent keys. The Bloom Filter quickly checks if a key might be present. If the Bloom Filter indicates “not present,” it can skip the database lookup.

4. Content Recommendation Systems

Problem: Recommendation systems, such as those used by streaming services, need to avoid recommending content that users have already consumed.

Solution: A Bloom Filter can track the content each user has previously watched or interacted with. When generating new recommendations, the Bloom Filter quickly checks if an item might already have been consumed.

5. Social Network Friend Recommendations

Problem: Social networks like Facebook or LinkedIn recommend friends or

connections to users, but they need to avoid recommending people who are already friends.

Solution: A Bloom Filter is used to store the list of each user's existing connections. Before suggesting new friends, the Bloom Filter can be checked to ensure the user isn't already connected with them.

🛑 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.

✅ Conclusion

To summarize, bloom filters are a powerful tool for space-efficient set membership testing, with a wide range of applications. While they may not be suitable for all applications due to the possibility of false positives, they shine in scenarios where space is at a premium and a small error rate is acceptable.

Thank you for reading!

If you found it valuable, hit a like ❤️ and consider subscribing for more such content every week.

If you have any questions or suggestions, leave a comment.

P.S. If you're enjoying this newsletter and want to get even more value, consider becoming a [paid subscriber](#).

As a paid subscriber, you'll unlock all **premium articles** and gain full access to all [premium courses](#) on [algomaster.io](#).

There are [group discounts](#), [gift options](#), and [referral bonuses](#) available.

Checkout my [Youtube channel](#) for more in-depth content.

Follow me on [LinkedIn](#), [X](#) and [Medium](#) to stay updated.

Checkout my [GitHub repositories](#) for free interview preparation resources.

I hope you have a lovely day!

See you soon,
Ashish

149 Likes · 8 Restacks

← Previous

Next →

Discussion about this post

Comments Restacks

Write a comment...

Nitish Nandwana 27 Nov 2024

Liked by Ashish Pratap Singh

Very good Ashish 👍 Keep it simple 👏👏👏

LIKE (2) REPLY SHARE

Akshay Kumar Pendyala 25 Nov 2024

Liked by Ashish Pratap Singh

This was so awesome!

LIKE (1) REPLY SHARE

4 more comments...