Vietnam National University Ho Chi Minh City

University of Science

Faculty of Information Technology

CHALLENGE 2. BLOOM FILTERS

Course	Data Structures and Algorithms
Class	22CLC02
Lecturers	Văn Chí Nam Lê Thanh Tùng Bùi Huy Thông Trần Thị Thảo Nhi
Participants	**22127357 – Phạm Trần Yến Quyên** **22127391 – Nguyễn Xuân Thành** **22127402 – Bế Lã Anh Thư**

HCMC, 2023

Contents

1 Introduction

1.1 Percentage of done works

Research	100%
Programming	100%

2 Research

In this section, we are going to do research about Bloom Filters.

2.1 Estimation Factor

Estimating the optimal size of the bit array and the number of hash functions for Bloom Filters depends on several factors, here are the key factors to consider:

- **Expected Number of Element (n):** The expected number of elements that plan to store in the Bloom Filters. The larger the number of elements, the larger the bit array will need to be to reduce the probability of collisions and false positives.
- **Desired False Positive Rate (p):** The false positive rate (or *the acceptable positive rate*), denoted as `p` is the probability that a membership test incorrectly identifies an element as a member of the set when it is not. The false positive rate is inversely related to the size of the bit array `m` and the number of hash functions `k`. A lower false positive rate requires a larger bit array and more hash functions.
- **Available Memory (M):** The amount of memory available for the Bloom Filter is a limiting factor. We need to ensure that the chosen bit array size and the number of hash functions can fit within the available memory. If memory is limited, we may need to balance the desired false positive rate and the expected number of elements accordingly.
- **Formula for Optimal Size:**

$$m = -\frac{n\ln(p)}{\ln(2)^2}$$

$$k = \frac{m}{n}\ln(2)$$

Where:

m: the optimal bit array size.

k: the number of hash functions.

p: the false positive rate.

n: the expected number of elements.

2.2 Impact Of Hash Functions

2.2.1 Performance

The performance of a Bloom Filters is influenced by the efficiency of the hash functions in generating unique and evenly distributed indices for the elements. The ideal hash functions should have the following characteristics to maximize performance:

- **Fast Computation**: Hash functions should be computationally efficient, as they will be applied multiple times for each element added or checked in the filter. Faster hash functions lead to quicker insertion and lookup operations so the hash function must be *as fast as possible.*
- **Low Collision Rate**: A collision occurs when two different elements are mapped to the same index in the bit array by a hash function. High collision rates can lead to *increased false positives*, *reducing the effectiveness* of the Bloom Filter. Efficient hash functions should minimize collisions to improve the filter's accuracy.
- **Independence:** The hash functions should be independent of each other, meaning the output of one hash function should not predict the output of another. This property is essential for *reducing the chance of correlated collisions* and *enhancing the filter's accuracy.*

2.2.2 Accuracy

The accuracy of a Bloom Filter is primarily influenced by the number of hash functions used and the size of the bit array. More hash functions and a larger bit array generally lead to better accuracy, but at the cost of increased memory usage. Here's how the hash functions impact accuracy:

- **Reducing False Positives**: False positives occur when a non-member element is incorrectly identified as a member of the set. The number of hash functions directly affects the false positive rate. Using more hash functions distributes the bits in the array more uniformly, reducing the probability of multiple unrelated elements setting the same bits and, therefore, decreasing the false positive rate.
- **False Negatives**: Hash functions do not impact false negatives in a Bloom Filter. A false negative occurs when the filter mistakenly reports that an element is not in the set when it actually is. Bloom Filters guarantee no false negatives because, upon checking, all bits corresponding to an element must be set to 1 for it to be considered in the set.

In conclusion, well-designed hash functions that are *fast*, *minimize collisions*, and *exhibit independence* are crucial for the efficient performance and accuracy of a Bloom Filter. Additionally, choosing an appropriate number of hash functions and bit array size based on the expected number of elements and false positive rate is essential to strike a balance between accuracy and memory usage.

2.3 False Positive Reduction

2.3.1 Do lookup operations on Bloom Filter always give correct results? Indicate the estimation method for false positive

- **No**, lookup operations on a Bloom filter do not always give correct results. The Bloom Filter is a probabilistic data structure, which means it can introduce false positives. A false positive occurs when the filter incorrectly indicates that an element is in the set, even though it is not.
- **Estimation method for false positive rate:**
  - **The false positive rate** (FPR) of a Bloom Filter depends on the size of the bit array (m), the number of hash functions (k), and the number of elements in the set (n). The formula to estimate the false positive rate (FPR) is:

$$FPR \approx (1 - e^{(-kn/m)})^k$$

    It's important to note that *the false positive rate increases as the number of elements in the set (n) and the number of hash functions (k) increase, while keeping the bit array size* *(m) constant.* Similarly, reducing the size of the bit array (m) or the number of hash functions (k) will also increase the false positive rate.

  * Another method to indicate the estimation for false positives is **Empirical Estimation Method**, this method is more practical and direct than theoretical estimation and is particularly useful when dealing with real-world data-sets.

However, it's important to remember that the false positive rate may vary depending on the data-set and the chosen parameters of the Bloom filter. The goal is to strike a balance between memory usage, the expected number of elements, and the acceptable false positive rate based on the specific application requirements.

2.3.2 Does the insertion of elements affect the false positive rate in a Bloom Filter?

- **Yes**, the insertion of elements does affect the false positive rate in a Bloom Filter.
- When elements are inserted into the Bloom Filter, the probability of setting bits in the bit array increases. This can lead to an increased likelihood of false positives.
- As more elements are added to the filter, the probability of collisions among hash functions also increases. Collisions cause multiple elements to share the same bit positions in the array, which contributes to false positives. Therefore, as the number of elements in the set (n) increases, the false positive rate tends to rise.

2.3.3 Are there any known techniques or strategies to reduce the false positive rate in a Bloom Filter?

While Bloom filters are designed to have false positives due to their space-efficient nature, there are strategies to mitigate and reduce the false positive rate:

- **Optimal Parameters**: By carefully selecting the size of the bit array (m) and the number of hash functions (k) based on the expected number of elements (n) and the desired false positive rate, you can minimize the false positive rate while still maintaining space efficiency.
- **Double Hashing**: Using double hashing, where each element undergoes two different hash functions to generate two separate indices in the bit array, can reduce the chance of collisions and subsequently lower the false positive rate.
- **Dynamic Bloom Filters**: Instead of using a fixed-size Bloom filter, dynamic Bloom filters allow resizing the bit array or adding more hash functions as the number of elements in the set increases. This adaptive approach can help maintain a low false positive rate even with changes in the size of the set.
- **Counting Bloom Filters**: Counting Bloom filters allow storing a count value at each bit position instead of just a binary value. This can help reduce false positives when the expected number of occurrences of an element in the set is known.
- **Cascading Bloom Filters**: Using multiple Bloom filters in a cascade, with each subsequent filter checking only those elements that have passed the previous filter, can help reduce false positives.

It's essential to keep in mind that while these techniques can reduce the false positive rate, they may come at the cost of increased memory usage or additional computational overhead. The choice of strategy should be based on the specific requirements and constraints of the use case.

3 Handling Bloom Filter

3.1 Is it possible to estimate the number of elements stored in a Bloom filter?

– **Yes**, it is possible to estimate the number of elements stored in a Bloom filter, but it is important to note that this estimation will not be exact.

– Since multiple elements can produce the same set of bits, false positives are possible. However, as false negatives are not possible; if the Bloom filter returns "definitely not in set," the element is certainly not present.

– Due to the potential for false positives, counting the exact number of elements in a Bloom filter is not directly possible. However, you can estimate the number of elements stored in a Bloom filter using various probabilistic methods.

– One common approach is to use the "**Bloom filter occupancy**" formula:

$$N \approx -m \times ln(1 - (\frac{s}{m}))$$

Where:

m: the number of bits in the Bloom filter (size of the bit array).

s: is the number of set bits (number of 1s) in the Bloom filter.

n: the estimated number of elements in the Bloom filter.

– Keep in mind that this estimation is not always highly accurate, especially if the Bloom filter has a high false positive rate or if it is significantly overloaded. The accuracy of the estimation largely depends on the chosen size of the Bloom filter, the number of hash functions used, and the number of elements that have been added to the filter.

3.1.1 Can a Bloom filter be dynamically resized to accommodate more elements?

– **No**, a standard Bloom filter cannot be dynamically resized to accommodate more elements. A Bloom filter is a space-efficient probabilistic data structure used to test whether an element is a member of a set. It works by using multiple hash functions to map elements to a bit array, and each element is represented by multiple bits.

– When an element is added to the Bloom filter, the corresponding bits in the array are set to 1. Checking for the membership of an element involves checking if all the bits for that element are set to 1. However, since the Bloom filter uses hashing and bit-level operations, it does not support deletions of elements.

– If you want to accommodate more elements or reduce the false positive rate of a Bloom filter, you typically need to create a new Bloom filter with a larger bit array and rehash and insert all the existing elements into the new filter. This process is not trivial and can be resource-intensive, especially if you have a large number of elements already in the original filter.

– **However**, there are ways to support deletions and is dynamic but it requires using more advanced data structures like **Counting Bloom filter**, **Scalable Bloom filter** or **Cuckoo filter**. In that, the structures **Counting Bloom filter** and **Cuckoo filter** provide a way to maintain a count of elements or support efficient deletion operations while still providing the benefits of approximate set membership checking. Whereas, **Counting Bloom filter** and the **Scalable Bloom filter**, allow for resizing and dynamic updates. However, they are more complex and may have higher memory overhead than traditional **Bloom filters**.

4 Comparison Table

	Bloom Filter	**Hash Table**
Space efficiency	Very space-efficient. Requires a fixed size bit array and a few hash functions. Small memory footprint. You could make a **Bloom Filter** with $k = 1$ and it would just be a hash table that ignores collisions. However, you would have a very large m if you wanted to keep your false positive rate low. The space of the actual data structure (what holds the data) is simply O(m).	Less space-efficient. Requires space for storing keys and values, leading to larger memory usage.
Time efficiency	Very fast. Constant-time complexity for insertion, deletion, and querying. If we are using a **Bloom Filter** with m bits and k hash function, insertion and search will both take `O(k)` time.	Fast on average. Constant-time complexity for insertion, deletion, and querying, but could degrade with hash collisions.
Insertion and Deletion possibility	Only supports insertion; does not support deletions of elements. Removing an item could cause *false negatives*.	Supports both insertion and deletion of elements. Elements can be easily added or removed.
Queries' Probabilistic Answer	Probabilistic. Can have *false positives* but no *false negatives*. Querying an element can return "possibly in set" or "definitely not in set."	Deterministic. Returns the exact value associated with a given key or a `NULL`/None value if the key is not present.
Limitations	May produce *false positives*, meaning it might indicate that an element is in the set when it is not. Deletions are not possible.	No `false positives` but has the potential for hash collisions, leading to slower performance. Larger data sets could lead to more collisions. May require resizing and rehashing.

- **When to use Bloom Filters:**
  * When memory efficiency is crucial, and false positives are tolerable (e.g., caching, network packet filtering).
  * When you want to quickly check if an item is probably in a large set.
- **When to use Hash Tables:**
  * When you need exact queries, without probabilistic answers.
  * When you need to perform insertions, deletions, and lookups with high precision.

5 Programming

5.1 Code structure

– The program has 4 sections:

  + Section 1: Containing Structures:

    * Librabries:
    * 2 structs:
      + UserAccount: store `<username>` and `<password>` of users + BloomFilter: The below structure is called Cross-checking Bloom Filter, like Bloom Filter it is a probabilistic data structure used for efficiently testing the membership of an element in a set. The difference lies in that the rate of false positive are significantly reduced thank to the use of Two-tier Bloom Filter. It has functions to insert elements into the filter and compute hash values.

```
        struct BloomFilter {
            std::vector<bool> primaryFilter;
            std::vector<bool> secondaryFilter;
            unsigned long long size;
            int hashCount;

            BloomFilter() : size(1e9 + 7), hashCount(3) {
                primaryFilter = std::vector<bool>(size, 0);
                secondaryFilter = std::vector<bool>(2 * size
                    , 0);
            }

            void insert(const std::string& key) {
                for (int i = 0; i < hashCount; ++i) {
                    unsigned long long primaryHash =
                        hashFunction(key, i) % size;
                    unsigned long long secondaryHash =
                        hashFunction(key, i) % (2 * size);
                    primaryFilter[primaryHash] = 1;
                    secondaryFilter[secondaryHash] = 1;
                }
            }

            unsigned long long hashFunction(const std::
                string& key, int i) {
                unsigned long long hash = 0;
                for (int j = 0; j < key.length(); ++j) {
                    // hash = (hash + (key[j] - 'a' + 1)) *
                        (i + 1);
                    hash = (hash + (int(key[j]) + 1)) * (i +
                        1);
                }
                return hash;
            }
        };
```

∗ In here:

+ `std::vector<bool> primaryFilter`: This vector is used to represent the primary filter of the Bloom Filter. It stores boolean values (either 0 or 1) to indicate whether an element is present in the set or not.
+ `std::vector<bool> secondaryFilter`: This vector is used to represent the secondary filter of the Bloom Filter. It is another layer of filtering to reduce the risk of false positives.
+ `unsigned long long size`: This variable holds the size of the Bloom Filter. It is initialized with a large prime number `1e9 + 7`.
+ `int hashCount`: The number of hash functions used to insert elements into the Bloom filter. It is initialized with the value 3.
+ `BloomFilter()`: This is the default constructor of the Bloom Filter struct. It initializes the primary and secondary filters with all zeros.
+ `void insert(const std::string key)`: This method allows you to insert an element into the Bloom Filter. It uses multiple hash functions to map the element to multiple positions in both the primary and secondary filters, setting the corresponding bits to 1.
+ `unsigned long long hashFunction(const std::string key, int i)`: This is the hash function used in the Bloom Filter. It takes the input key (a string) and an integer `i` (used to differentiate between multiple hash functions). The function calculates a hash value based on the characters of the string and the value of `i`.

+ Section 2: Containing some auxiliary functions that assist to main-functions
+ Section 3: Implementing four main-functions corresponding to four Operations at Item2 of Challenge 2
+ Section 4: Containing main function of program, compile and run program

5.2 Librabries

Beside basic librabries, we also use auxiliary librabry name `<vector>` to store a list of accounts is registered before.

5.3 How does this program work?

– This program uses a varient of Bloom Filter, Cross-checking Bloom Filter to do the main chore of hashing Users data (Username and Password) to do the following tasks:
  - Registration: Verify the validity of the account and store created information in the database.
  - Multiple registrations: Verify the validity to register multiple accounts and store created information in the database.
  - Log-in: Users need to enter correct Username and Password to login.
  - Change Password: Changing already existing accounts password. As well as side functions such as: “Check if Account exist”, “Check if Valid Password”, “Check if Valid Username”.

- In the program, there are your normal variables like string username, string password,etc to store information (in this case User Account). However, there are 4 special variables that is use to power the program most important functions:
  + `vector<UserAccount>` accounts: Vector for storing succesfully registered user account.
  + BloomFilter UserFilter: A specific Bloom Filter to filter inserted Usernames.
  + BloomFilter PassFilter: A specific Bloom Filter to filter inserted Passwords.
  + BloomFilter WeakpassFilter: A specific Bloom Filter to filter inserted WeakPasswords (got from "WeakPass.txt").
- Supporting functions:
  + `bool IsExisted(string key, BloomFilter Typefilter)`: This function checks if an element exists in the given Bloom filter. It uses cross-checking to validate both primary and secondary filters to reduce false positives.
  + `bool IsValidUsername(const string username)`: This function checks if a given username is valid. It must be between 6 to 9 characters long and should not contain any spaces.
  + `bool IsWeakPassword(const string password, BloomFilter WeakpassFilter)`: This function checks if a given password is weak by checking its existence in the weak password Bloom filter.
  + `bool IsValidPassword(const string username, const string password, BloomFilter WeakpassFilter)`: This function checks if a given password is valid. It must be between 11 to 19 characters long, must not be the same as the username, and must contain at least one uppercase letter, one lowercase letter, one digit, and one special character. It also checks if the password is not weak using the IsWeakPassword function.
  + `void ReadWeakPasswordsFromFile(BloomFilter WeakpassFilter, const string filename)`: This function reads weak passwords from a file and inserts them into the WeakpassFilter Bloom filter.
  + `void ReadSignUpFile(vector<UserAccount> accounts, string filename)`: This function reads user account information (username and password) from a file and stores them in the accounts vector of UserAccount structures.
  + `int CheckValidAccount(string username, string password, BloomFilter UserFilter, BloomFilter PassFilter, BloomFilter WeakpassFilter)`: This function checks the validity of a user account. It checks if the username is valid, if it already exists in the UserFilter Bloom filter, and if the password is valid using IsValidPassword. It returns an integer code representing the outcome of the validation.
- Main functions:
  + Registration: `void RegisterAccount(BloomFilter UserFilter, BloomFilter PassFilter, BloomFilter WeakpassFilter, vector<UserAccount> accounts)`: This function handles the registration process. It prompts the user to enter a username and password and checks the validity using CheckValidAccount. If the account is valid, it is added to the UserFilter and PassFilter Bloom filters, and the UserAccount is added to the accounts vector.
  + Multiple registrations: void MultipleRegistration(BloomFilter UserFilter, BloomFilter PassFilter, BloomFilter WeakpassFilter, vector<UserAccount> accounts): This function reads user account information from a file and attempts to register each account using CheckValidAccount. If the registration is successful, the UserAccount is added to the accounts vector, otherwise, it is written to a "Fail.txt" file.

  + Log-in: void Login(BloomFilter UserFilter, BloomFilter PassFilter, vector<UserAccount> accounts): This function handles the login process. It prompts the user to enter a username and password and checks if the username exists in the UserFilter. If the username exists, it checks if the password matches the one associated with the username.
  + Change Password: void ChangePassword(BloomFilter UserFilter, BloomFilter PassFilter, BloomFilter WeakpassFilter, vector<UserAccount> accounts): This function handles the change password process. It prompts the user to enter the old password, a new password, and reconfirm the new password. It checks if the new password is valid using IsValidPassword and if the old password matches the one associated with the username. If the password change is successful, it updates the password in the accounts vector and PassFilter Bloom filter.

– MAIN: int main(): This is the main function that initializes the Bloom filters, reads weak passwords from the "WeakPass.txt" file, and presents a menu using switch case for the user to perform various mentioned account management actions. The main function calls the respective functions based on the user's choice until the user chooses to exit.

6 Why use Cross-checking Bloom Filter ?

– Bloom filters are probabilistic data structures used to efficiently test the membership of an element in a set. They provide constant-time complexity for insertion and membership queries while consuming a relatively small amount of memory. However, they have a drawback: they can generate false positives, meaning they might mistakenly claim that an element is in the set when it's not. This is due to the possibility of hash collisions.

– Cross-Checking Bloom Filters (CCBF) are an extension of standard Bloom filters that aim to reduce the false positive rate. They do this by introducing an additional step to cross-check potential false positives before confirming them. Here's how CCBF works and why it's more accurate than standard Bloom filters, illustrated with statistics:

– Standard Bloom Filter (BF):
  . Assume you have a standard Bloom filter with 'm' bits and 'k' hash functions.
  . Given 'n' elements in the set, the false positive rate (FPR) can be approximated by: $(1 - e^{-kn/m})^k$.

– Cross-Checking Bloom Filter (CCBF):
  . In CCBF, you have two independent Bloom filters, each with 'm' bits and 'k' hash functions.
  . You insert each element into both filters independently.

– False Positive Reduction:
  . To check membership, you query both filters.
  . If one filter returns false, the element is definitely not in the set.
  . If both filters return true, you perform a cross-check using a third independent hash function.
  . If the cross-check hash function returns true, the element is considered a true positive. Otherwise, it's treated as a false positive.

– Now, let's compare the false positive rates of standard Bloom filters and Cross-Checking Bloom Filters:

. Suppose we have a set of 1,000 elements (n = 1,000) and a Bloom filter with 10,000 bits (m = 10,000) and 4 hash functions (k = 4).

. Standard Bloom Filter (BF):

- $FPR \approx (1 - e^{-4*1000/10000})^4 \approx 0.183$
- The false positive rate is approximately $18.3 percent.$

. Cross-Checking Bloom Filter (CCBF):

- Since there are two independent filters, the FPR for each filter is the same as above: $18.3 percent.$
- The probability of a false positive in both filters is approximately $(0.183)^2 \approx 0.034$.
- Let's assume the cross-check hash function has a false positive rate of 0.1.
- The overall FPR for $CCBF \approx 0.034 + (0.034 * 0.1) \approx 0.037$.

- Therefore, the Cross-Checking Bloom Filter achieves an FPR of approximately $3.7 percent$, which is significantly lower than the $18.3 percent FPR of the standard Bloom filter.$

– By introducing the cross-checking step, Cross-Checking Bloom Filters are more accurate in distinguishing true positives from false positives, making them a better choice in scenarios where false positives need to be minimized at the expense of some additional memory usage and computation during membership queries.

7 References

– Stack Over Flow - How many hash functions does my bloom filter need?

– Git Hub - Bloom Filters by Example

– One Stop Data Analysis - Bloom Filter Made Simple: Theory and Code

– COCOMO - An Empirical Estimation Model for Effort

– Naturals Publishing - Reducing False Positives of a Bloom Filter using Cross-Checking Bloom Filters

– Difference between Bloom filters and Hashtable

– A New Analysis of the False-Positive Rate of a Bloom Filter