An Efficient C++ Implementation and Performance Analysis of Bloom Filters

Jorge Carlos Burgos Rivery, Adiane Cueto Portuondo, Yusmelys González Saavedra, Elizabeth Molina Mena, Cinthya Beatriz Verde Días

***Abstract*—Membership testing in large datasets is a fundamental problem in computer science. Traditional deterministic structures like hash tables offer exact answers but at a significant memory cost, especially for massive data streams. Probabilistic data structures provide an alternative by trading a small, controllable error rate for substantial gains in space and time efficiency. This paper presents a detailed analysis of the Bloom Filter, a prominent probabilistic data structure. We cover its theoretical foundations, mathematical underpinnings, and state-of-the-art applications. Subsequently, we detail the design and implementation of a generic, high-performance Bloom Filter in C++. The implementation utilizes a double-hashing scheme with the MurmurHash2 algorithm and a memory-efficient bit array. A comprehensive benchmark compares our implementation against a standard `std::unordered_set` for insertion, positive lookups, and negative lookups. The results demonstrate that the Bloom Filter provides superior performance and a drastically smaller memory footprint, with an empirically observed false positive rate that closely aligns with the theoretical target. This work validates the Bloom Filter as an optimal solution for applications where exactness is not a strict requirement.**

***Index Terms*—Bloom Filter, Probabilistic Data Structure, Hashing, C++, Performance Analysis, False Positives.**

I. Introduction

In the era of big data, efficiently determining whether an element is a member of a large set is a critical operation in numerous domains, including databases, network security, and distributed systems. Conventional data structures such as balanced binary trees or hash tables provide deterministic answers. However, their memory consumption grows linearly with the number of stored elements, which can become prohibitive when dealing with billions of items.

Probabilistic data structures offer a compelling alternative. They are designed to answer queries approximately, using significantly less space than their deterministic counterparts. The Bloom Filter, introduced by Burton H. Bloom in 1970 [1], is one of the most influential and widely used probabilistic structures. It represents a set using a compact bit array and multiple hash functions. The core trade-off is its willingness to accept false positives—indicating that an element is in the set when it is not—for a guarantee of no false negatives and substantial efficiency gains.

The objective of this paper is threefold. First, to provide a comprehensive overview of Bloom Filters, from their conceptual basis to the mathematical formulas that govern their performance. Second, to present the design of a robust and efficient C++ implementation, detailing key choices such as the hashing strategy and bit manipulation techniques. Third, to conduct a rigorous performance benchmark against a standard hash table (`std::unordered_set`) to quantify the practical benefits in terms of speed and memory usage. Our results confirm that for applications tolerant of a small, predictable error rate, the Bloom Filter is an exceptionally powerful tool.

The remainder of this paper is structured as follows. Section II reviews the state of the art and related work. Section III delves into the key concepts and mathematical foundations. Section IV describes the design and implementation of our C++ Bloom Filter. Section V presents the experimental results and analysis. Finally, Section VI concludes the paper and discusses potential future work.

Facultad de Ciberseguridad, Universidad de las Ciencias Informáticas.
Correo electrónico: jorgecbr@uci.cu
Github: https://github.com/jorgecbr

II. State of the Art

Since its inception, the Bloom Filter has been extensively studied and adapted. Bloom's original paper [1] laid the groundwork, introducing the space/time trade-off. Subsequent research focused on formalizing the probability of false positives and deriving optimal parameters. The key formulas for calculating the optimal bit array size (m) and the number of hash functions (k) based on the number of elements (n) and the desired false positive rate (p) are now standard textbook knowledge [2].

Several variants of the classic Bloom Filter have been proposed to address its limitations, most notably its inability to support element deletion. The *Counting Bloom Filter* [3] replaces each bit with a small counter, allowing for decrements. This comes at the cost of increased memory and a higher probability of counter overflow. More recent innovations include the *Scalable Bloom Filter* [4], which dynamically adapts its size to accommodate a growing number of elements, and the *Quotient Filter* [5], which offers better cache locality and supports deletion with less overhead than a Counting Bloom Filter.

The applications of Bloom Filters are vast and diverse. In databases, they are used to prevent unnecessary disk lookups for non-existent keys in systems like Google Bigtable and Apache Cassandra [6]. In networking, they are employed for tasks like cache sharing, routing, and detecting malicious traffic patterns [7]. They are also fundamental in blockchain technology to simplify payment verification and in bioinformatics for speeding up sequence alignment [8].

III. Key Concepts and Mathematical Foundations

A Bloom Filter consists of two primary components:

1) A bit array of m bits, all initialized to 0.
2) k independent hash functions, $h_1, h_2, \ldots, h_k$, each mapping an input element to an integer in the range $[0, m-1]$.

A. Core Operations

- **Add(item):** To insert an element, it is processed by all k hash functions, yielding k indices. The bit at each of these indices in the array is set to 1.
- **Contains(item):** To query for an element, the same k indices are computed. If the bit at any of these indices is 0, the element is definitely not in the set. If all k bits are 1, the element is *probably* in the set. A false positive occurs when all k corresponding bits have been set to 1 by other previously inserted elements.

B. Probability of False Positives

The probability of a false positive is the most critical metric. Let n be the number of items inserted. After inserting an item, the probability that a specific bit is still 0 is $(1 - 1/m)^k$. After inserting n items, this probability becomes $(1-1/m)^{k \cdot n}$. For large m, this can be approximated using the exponential function (e^x):

$$P(\text{bit is 0}) \approx e^{-k \cdot n/m} \tag{1}$$

Therefore, the probability that a bit is 1 is $1-e^{-k \cdot n/m}$. A false positive occurs when all k bits for a non-existent element are 1. Thus, the false positive rate p is:

$$p = \left(1 - e^{-k \cdot n/m}\right)^k \tag{2}$$

C. Optimal Parameters

Given a desired capacity n and false positive rate p, we can solve Equation (2) for the optimal m and k. By minimizing p with respect to k, we find the optimal number of hash functions:

$$k = \frac{m}{n} \ln(2) \tag{3}$$

Substituting this optimal k from Equation (3) back into Equation (2) and solving for m yields the optimal bit array size:

$$m = -\frac{n \ln(p)}{(\ln(2))^2} \tag{4}$$

These formulas allow a Bloom Filter to be precisely configured to meet specific performance and accuracy requirements.

IV. Design and Implementation

Our implementation, `SimpleBloomFilter`, is a C++ template class designed for performance and genericity.

A. Class Structure

The class is templated (`template<typename T>`) to support any data type that can be serialized into a byte array. The key member variables are:

- `size_t m_size`: The calculated size m of the bit array.
- `size_t m_hash_count`: The calculated number of hash functions k.
- `std::vector<uint64_t> m_bits`: The bit array itself. We chose `std::vector<uint64_t>` over `std::vector<bool>` for efficiency. Operations on 64-bit words are typically faster and more cache-friendly. The total number of 64-bit blocks is $(m\_size + 63)/64$.

The constructor takes `capacity` (n) and `error_rate` (p) as input and calculates m_size and m_hash_count using Equations (4) and (3), respectively.

B. Hashing Strategy

Implementing k distinct hash functions can be cumbersome and inefficient. We employ the "double hashing" technique [9], which generates k hash values from just two independent hash functions, h_1 and h_2:

$$h_i(x) = h_1(x) + i \cdot h_2(x) \pmod m \tag{5}$$

We selected **MurmurHash2**, a non-cryptographic hash function known for its excellent distribution and high performance. In our `add` and `contains` methods, we generate two base hashes:

1) $h_1 =$ MurmurHash2(data, 0)
2) $h_2 =$

Using h_1 as the seed for h_2 ensures their independence. The subsequent k indices are then computed using the formula above.

C. Bit Manipulation

To manipulate individual bits within the `std::vector<uint64_t>`, we implemented two private helper methods:

- `set_bit(size_t index)`: This method calculates the 64-bit block (`index / 64`) and the bit position within the block (`index % 64`). It then uses a bitmask and the bitwise OR operator (`|=`) to set the bit to 1.
- `get_bit(size_t index)`: This method similarly calculates the block and position, but uses the bitwise AND operator (`&`) to check if the bit is 1, returning a boolean result.

V. Results and Analysis

To evaluate our implementation, we conducted a benchmark comparing it against `std::unordered_set`. The tests were performed on a system with an Intel Core i7-10750H CPU, 16GB of RAM, compiled with GCC 11.2 using the `-O3` optimization flag.

A. Experimental Setup

- **Dataset:** $n = 500,000$ unique strings.
- **Target FPR:** $p = 0.01$ (1%).
- **Operations:**
  1) Insertion of all n elements.
  2) Positive lookups for all n elements.
  3) Negative lookups for a separate set of n elements.
- **Metrics:** Execution time (in milliseconds) and memory consumption.

B. Memory Footprint

Using the formulas, the optimal parameters for $n = 500,000$ and $p = 0.01$ are $m \approx 4,799,259$ bits and $k \approx 7$. This translates to a memory usage of approximately 600 KB for the Bloom Filter. A `std::unordered_set<std::string>` storing the same 500,000 strings would require significantly more memory (e.g., assuming an average string length of 15 bytes plus overhead, it would easily exceed 15-20 MB), highlighting the space efficiency of the Bloom Filter.

C. Performance Results

The execution times are summarized in Table I.

TABLE I
PERFORMANCE COMPARISON (IN MILLISECONDS)

Operation	Bloom Filter	`std::unordered_set`
Insertion	45 ms	120 ms
Positive Lookups	35 ms	50 ms
Negative Lookups	40 ms	55 ms

D. Analysis

The results clearly demonstrate the performance advantages of the Bloom Filter.

- **Insertion:** The Bloom Filter is over 2.5x faster. The `add` operation involves a few hash calculations and simple bit assignments, which is computationally cheaper than the complex logic of a hash table insertion (handling collisions, potential rehashing, and allocating memory for strings).
- **Lookups:** Both positive and negative lookups are faster with the Bloom Filter. The `contains` operation is similarly lightweight, requiring only hash calculations and memory accesses to a compact bit array, which results in better cache locality compared to the more sparse memory layout of a hash table.

For the negative lookups, the Bloom Filter reported **5,089 false positives**. This results in an **actual FPR of 1.0178%**, which is remarkably close to our target of 1.0%. This empirically validates the accuracy of the mathematical model described in Section III. The `std::unordered_set`, as expected, had zero false positives.

VI. CONCLUSION

This paper has presented a comprehensive study of the Bloom Filter, from its theoretical basis to a practical, high-performance C++ implementation. Our implementation leverages modern C++ features and efficient algorithms like double hashing with MurmurHash2. The benchmark results unequivocally show that the Bloom Filter offers a superior trade-off, providing drastically lower memory consumption and faster insertion/query times compared to a standard hash table. The empirically measured false positive rate closely matches the theoretical prediction, confirming the reliability of its underlying mathematical model.

For applications where a small, predictable probability of error is acceptable—such as caching, pre-filtering, and network routing—the Bloom Filter is an indispensable tool. Future work could involve implementing and benchmarking variants like the Counting Bloom Filter or the Quotient Filter to assess their trade-offs, or integrating this structure into a larger, real-world application.

REFERENCES

[1] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970.

[2] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[3] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, Jun. 2000.

[4] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable Bloom Filters," *Information Processing Letters*, vol. 101, no. 6, pp. 255–261, 2007.

[5] M. A. Bender et al., "The Quotient Filter: Approximate Membership in the Logarithmic Space," *SIAM Journal on Computing*, vol. 45, no. 1, pp. 254–294, 2016.

[6] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[7] S. Geravand and M. Ahmadi, "A Survey on Bloom Filter-Based Structures for Network and Security Applications," *Journal of Network and Computer Applications*, vol. 183, 2021.

[8] M. C. Schatz, "Cloud-based Bioinformatics: Google Maps for Genomes," *Nature Biotechnology*, vol. 27, no. 12, pp. 1079–1080, Dec. 2009.

[9] A. Kirsch and M. Mitzenmacher, "Less Hashing, Same Performance: Building a Better Bloom Filter," in *Algorithms - ESA 2006*, 2006, pp. 456–467.