

Jihočeská univerzita  
v Českých Budějovicích  
University of South Bohemia  
in České Budějovice

**University of South Bohemia**

**Faculty of Science**

Artificial Intelligence and Data Science, M.Sc.

**DESIGNING A BLOOM FILTER FOR A TARGET FALSE POSITIVE RATE**

Submitted for the course **Information Theory**

**Professor:** Kašťovský Jan prof. RNDr. Ph.D.

Submitted by: María Isabel Sánchez-O'Mullony Martínez

Student ID: B24763

Date: November 26, 2025

Supervisor: Kašťovský Jan prof. RNDr. Ph.D.

## **Declaration**

I declare that I have written this report by myself and have only used the sources and aids mentioned, and that I have marked direct and indirect citations as such. This report has not been submitted prior for any other examination.

I agree that the results of this study work / report may be used free of charge for research and lecturing purposes.

## **Contents**

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Algorithm Description</b>	<b>4</b>
<b>3</b>	<b>Implementation</b>	<b>5</b>
	<b>References</b>	<b>6</b>

## **List of Abbreviations**

BF Bloom Filter

FPR False-Positive Rate

# 1 Introduction

A Bloom filter (BF) is a simple, space-efficient, randomized data structure for concisely representing a static data set, in order to support approximate membership queries [1]. It is mainly a spaced optimized version of hashing where we may have false positives. The idea is to not store the actual key rather store only hash values. It is mainly a probabilistic and space optimized hashing where less than 10 bits per key are required for a 1% false positive probability and is not dependent on the size of individual keys [2].

The structure has some interesting properties: It offers a compact probabilistic way to represent a set that can result in false positives (claiming an element to be part of the set when it is not), but never in false negatives (reporting an inserted element to be absent from the set). A Bloom filter of a fixed size can represent a set with an arbitrary large number of elements. Adding an element never fails, however, we need to take into account the increasing probability of false positive [2, 3].

The tradeoff is that in exchange for providing a sometimes incorrect false positive answer, a bloom filter consumes a lot less memory than other data structures, like a hash table, and it is much faster. Another key point, is that is not possible to remove an item from the Bloom filter.

The objective of this project is to design and implement a Bloom filter that automatically chooses the optimal size ( $m$ ) and number of hash functions ( $k$ ) to achieve a target false-positive rate (FPR) ( $p$ ), given an expected number of elements ( $n$ ).

# 2 Algorithm Description

Bloom Filter has two operations, insert data and key lookup. It uses a bit array and hashing techniques to store the existence of a string.

A BF for representing a set  $S = \{x_1, \dots, x_n\}$  of  $n$  items is described by a vector of  $m$  bits, initially all set to 0. The key idea is to use  $k$  hash functions,  $h_i(x)$ ,  $1 \leq i \leq k$  to map items  $x \in S$  to random numbers uniform in the range  $1, \dots, m$ , where the hash functions are assumed to be uniform [3].

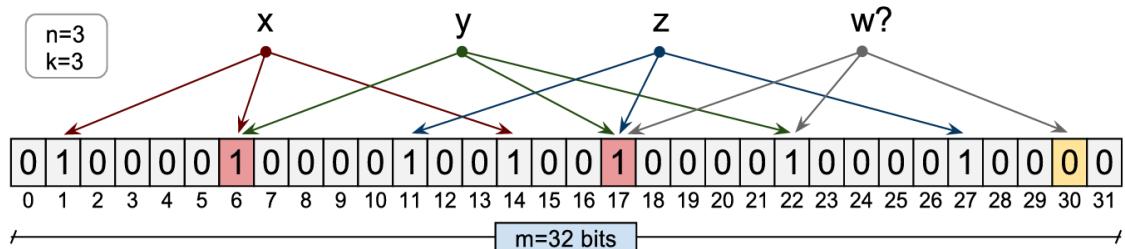


Figure 1: Overview of a Bloom Filter [3].

To insert data in the BF, we first need a big bit array initialized to 0 and multiple hash functions which will take the data and convert it into a unique number that points to a specific switch in the array. When we add the new item into the filter we feed its name through the hash functions and each function points to a switch in the array, turning the switches to 1 [4].

To search for an item, the hash function is used again, checking if the corresponding switches in the array are 1. If there is even one that is 0, then the item is not part of the set.

However, there is the possibility that even though all the switches are 1, the item might not be part of the set, this is a false positive.

### 3 Implementation

The implementation was written in Python.

To design a Bloom filter for a specific FPR, you primarily compute  $m$  and  $k$  using the formulas above based on your expected  $n$  and target  $p$ . Properly tuning these parameters ensures your Bloom filter maintains your desired false positive rate.

1. Use the calculated  $m$  and  $k$  parameters.
2. Select hash functions that are independent and distribute uniformly over the bit array.
3. Be aware that increasing  $m$  (more bits) reduces FPR but increases memory consumption.
4. Adjust parameters if your element estimate  $n$  or FPR requirement changes.

## References

- [1] D. Guo, J. Wu, H. Chen, and X. Luo, "Theory and network applications of dynamic bloom filters," in *Proceedings IEEE INFOCOM 2006. 25TH IEEE International Conference on Computer Communications*. IEEE, 2006, pp. 1–12.
- [2] GeeksforGeeks. (2025) Bloom filters - introduction and implementation. [Online]. Available: <https://www.geeksforgeeks.org/python/bloom-filters-introduction-and-python-implementation/>
- [3] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2011.
- [4] ByteMonk, "Bloom filters, hashtable, system design," YouTube video, May 2024, accessed: 2025-11-26. [Online]. Available: <https://www.youtube.com/watch?v=GT0En1dGntY>