BLOOM **FILTER**

A Data Structure for Computer Networking, Big Data, Cloud Computing, Internet of Things, Bioinformatics and Beyond

Ripon Patgiri
Sabuzima Nayak
Naresh Babu Muppalaneni

BLOOM FILTER

A Data Structure for Computer Networking, Big Data, Cloud Computing, Internet of Things, Bioinformatics and Beyond

This page intentionally left blank

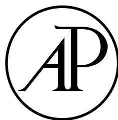
BLOOM FILTER

A Data Structure for Computer Networking, Big Data, Cloud Computing, Internet of Things, Bioinformatics and Beyond

Ripon Patgiri
Department of Computer Science and Engineering
National Institute of Technology
Silchar, India

Sabuzima Nayak
Department of Computer Science and Engineering
National Institute of Technology
Silchar, India

Naresh Babu Muppalaneni
Department of Computer Science and Engineering
National Institute of Technology
Silchar, India



ACADEMIC PRESS
An imprint of Elsevier

Academic Press is an imprint of Elsevier
125 London Wall, London EC2Y 5AS, United Kingdom
525 B Street, Suite 1650, San Diego, CA 92101, United States
50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States
The Boulevard, Langford Lane, Kidlington, Oxford OX5 1GB, United Kingdom

Copyright © 2023 Elsevier Inc. All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or any information storage and retrieval system, without permission in writing from the publisher. Details on how to seek permission, further information about the Publisher's permissions policies and our arrangements with organizations such as the Copyright Clearance Center and the Copyright Licensing Agency, can be found at our website: www.elsevier.com/permissions.

This book and the individual contributions contained in it are protected under copyright by the Publisher (other than as may be noted herein).

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods, professional practices, or medical treatment may become necessary.

Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

ISBN: 978-0-12-823520-1

For information on all Academic Press publications
visit our website at https://www.elsevier.com/books-and-journals

Publisher: Mara E. Conner
Acquisitions Editor: Chris Katsaropoulos
Editorial Project Manager: John Leonard
Production Project Manager: Selvaraj Raviraj
Cover Designer: Christian J. Bilbow

Typeset by VTeX

Contents

Preface vii
Acknowledgments ix

I Bloom Filters

1. Introduction

1.1 Introduction 3
1.2 Organization 3
References 5

2. Bloom Filters: a powerful membership data structure

2.1 Introduction 7
2.2 Bloom Filter 8
2.3 Bit array 10
2.4 Taxonomy of response 11
2.5 Key objectives 12
2.6 Taxonomy of Bloom Filter 13
2.7 Analysis of false positive 17
2.8 Lessons learned 18
2.9 Conclusion 19
Appendix 2.A Source code of Bloom Filter 20
Appendix 2.B Symbols used in the chapter 21
References 21

3. robustBF: a high accuracy and memory efficient 2D Bloom Filter for diverse applications

3.1 Introduction 23
3.2 Preliminaries 24
3.3 robustBF – the proposed system 25
3.4 Experimental results 27
3.5 Analysis 32
3.6 Conclusion 34
References 34

4. Impact of the hash functions in Bloom Filter design

4.1 Introduction 37
4.2 Hash functions 37
4.3 Prime numbers in Bloom Filter 38
4.4 Number of hash functions 38
4.5 Types of hash functions 39
4.6 Comparison of robustBF and libbloom 41
4.7 Conclusion 42
References 43

5. Analysis on Bloom Filter: performance, memory, and false positive probability

5.1 Introduction 45
5.2 False positive probability 45
5.3 Memory 47
5.4 Performance 47
5.5 Conclusion 50
References 50

6. Does not Bloom Filter bloom in membership filtering?

6.1 Introduction 51
6.2 Bloom Filter is not a complete system! 51
6.3 Learned Bloom Filter 53
6.4 Malicious URL filtering using Bloom Filter 55
6.5 Conclusion 56
References 56

7. Standard of Bloom Filter: a review

7.1 Introduction 57
7.2 Literature review on standard Bloom Filter 57
7.3 Other approximation filters 62
7.4 Conclusion 63
References 63

8. Counting Bloom Filter: architecture and applications

8.1 Introduction 65
8.2 Counting Bloom Filter 65
8.3 Variants 69
8.4 Issues 76
8.5 Conclusion 77
References 78

9. Hierarchical Bloom Filter

9.1 Introduction 79
9.2 Variants 79
9.3 Issues 83
9.4 Applications 85
9.5 Conclusion 86
References 86

II
Applications of Bloom Filter in networking

10. Applications of Bloom Filter in networking and communication

10.1 Introduction 89
10.2 Traffic management 89
10.3 Packet management 92
10.4 Routing 94
10.5 Searching 96
10.6 Discussion 97
10.7 Conclusion 98
References 98

11. Bloom Filter for named-data networking

11.1 Introduction 101
11.2 Named data networking 101
11.3 Named data networking packet 102
11.4 Content store 105
11.5 Pending interest table 107
11.6 Forwarding information base 109
11.7 Security 110
11.8 Discussion 112
11.9 Conclusion 113
References 114

12. Enhancement of software-defined networking using Bloom Filter

12.1 Introduction 117
12.2 SDN architecture 117
12.3 Control layer 118
12.4 Data layer 119
12.5 Issues and challenges 126
12.6 Security 126
12.7 Discussion 128
12.8 Conclusion 129
References 129

13. Impact of Bloom Filter in wireless network

13.1 Introduction 131
13.2 Wireless sensor networks 131
13.3 Mobile ad-hoc networks 136
13.4 Internet-of-Things 138
13.5 Discussion 141
13.6 Conclusion 141
References 142

14. Network security using Bloom Filter

14.1 Introduction 145
14.2 DDoS 145
14.3 DoS 148
14.4 Defense against various network attacks 150
14.5 Security 151
14.6 Privacy 153
14.7 Evaluation 155
14.8 Discussion 156
14.9 Conclusion 157
References 157

III
Applications of Bloom Filter in other domains

15. Applications of Bloom Filter in Big data

15.1 Introduction 161
15.2 Data management 161
15.3 Database 165
15.4 MapReduce 169
15.5 Discussion 170
15.6 Conclusion 171
References 172

16. Bloom Filter in cloud computing

16.1 Introduction 175
16.2 Indexing and searching of encrypted data 176
16.3 Cloud data storage management 181
16.4 Discussion 183
16.5 Conclusion 184
References 184

17. Applications of Bloom Filter in biometrics

17.1 Introduction 187
17.2 Biometrics 187
17.3 Cancelable biometrics 190
17.4 Discussion 192
17.5 Conclusion 193
References 194

18. Bloom Filter for bioinformatics

18.1 Introduction 197
18.2 Preprocessing filtering 198
18.3 de Bruijn graph 204
18.4 Searching 206
18.5 DNA assembly technique 207
18.6 Other bioinformatics areas 209
18.7 Discussion 211
18.8 Conclusion 212
References 212

Index 215

Preface

Bloom Filter is a simple data structure for membership filtering. It has a simple architecture and operations with high performance. The input items are mapped to the Bloom Filter rather than storage. This feature of Bloom Filter is both a blessing and a curse. A blessing because it has a low memory footprint and fast operation. A curse because it is not a standalone technique, rather a system/application enhancer. A big data application or an application responsible for processing a huge volume of data can implement Bloom Filter to enhance its processing performance. Bloom Filter with constant time complexity operation processes the incoming item quickly, specifically it has great advantage where the application has more negative responses. In case Bloom Filter returns "false", the item is definitely absent; otherwise if Bloom Filter returns "true", an item may be present in the application.

The simple Bloom Filter is a versatile data structure. It can be implemented in diverse applications. We, the authors, wrote some review papers to highlight the role of Bloom Filter, its issue and challenges in some areas. Then we realized that the research world does not need a review paper on Bloom Filter, but rather a book to illustrate the diverse applications of Bloom Filter. In the book we tried to advocate the superiority of Bloom Filter. The chapters are classified into three parts: (i) precise information about Bloom Filter theory, architecture, variants, issue and challenges, (ii) application in networking, and (ii) other application domains.

Acknowledgments

At the outset, we would like to express our sincere gratitude to the God Almighty for everything.

Ripon Patgiri would like to express his sincere thanks to his mother Umali Patgiri (Doley), wife Rani Patgiri, daughter Sara Patgiri, and son Vivan Patgiri for their constant support and inspiration. Naresh Babu Muppalaneni would like to express his sincere thanks to his wife Prathyusha and daughter Sai Manasa for their continuous support and inspiration.

Sabuzima Nayak wants to express gratitude to her PhD supervisor for giving the opportunity to contribute to this book. She also thanks her elder sister Priyaja Nayak, mother Gitanjali Nayak, and father Akshaya Kumar Nayak for supporting her and appreciating her efforts. This book is special for her because this is her first monograph and the first topic which she started researching from her Master to PhD and will continue further in her career. The writing process of this book has helped her learn many new concepts and theories related to Bloom Filter. In addition, this work kept her engaged during the pandemic time (year 2020) when she was unable to do research due to the lack of resources and helped her remain calm and sane in those chaotic times.

We are hopeful to provide the reader with subsequent versions, including new development of Bloom Filter, review of new proposed Bloom Filter variants, and implementation in new domains.

Finally, the authors would like to express their sincere thanks to the National Institute of Technology Silchar for providing computing resources and libraries.

Ripon Patgiri
Sabuzima Nayak
Naresh Babu Muppalaneni

This page intentionally left blank

PART I

Bloom Filters

This page intentionally left blank

CHAPTER

1

Introduction

1.1 Introduction

Bloom Filter [1] can be applied in diverse domains, namely, Big Data [2], Database [2], Cloud Computing [3], Computer Networking [4,5], IoT [6], Security [7,8], Bioinformatics [9], Biometrics [10], and many other domains. However, Bloom Filter does not store data and, therefore, it is inapplicable in cryptography and hard real-time systems such as those in defense applications. Moreover, Bloom Filter cannot understand patterns and, therefore, it cannot be used to discover patterns. However, recent development suggests that Bloom Filter is applicable in machine learning, too.

Bloom Filter is an approximation data structure that has a tiny memory footprint. The memory size is dependent on the number of inputs, but it does not depend on the size of an individual item. For instance, if the size of a single input is 512 MB, then Bloom Filter takes k bits to store the information for the same, where k is the number of hash functions. The value of k may be in the range of $[5, 20)$. A few bits can represent a large-sized input. On the contrary, if the size of a single input is 4 bits, it still takes the same number of bits to represent this input. Therefore, the size of the Bloom Filter is dependent on the number of inputs but not on the size of individual input items.

1.2 Organization

Chapter 2 explores conventional Bloom Filter. It demonstrates the basic operations of conventional Bloom Filter, namely, insertion and query. Moreover, Chapter 2 illustrates why non-counting Bloom Filter should avoid deletion operation, which introduces the possibility of a false negative which is not desirable. Furthermore, the chapter classifies the Bloom Filter into various categories based on memory allocation, architecture, implementation, and platform. The chapter also presents the analysis of memory, false positive probability, and an optimal number of hash functions.

Chapter 3 presents a powerful Bloom Filter, called robustBF, which is developed based on a two-dimensional bit array, and thus is more economical in terms of memory. Also, it is faster than the state-of-the-art Bloom Filters due to fewer hash function invocations. It can achieve low false positive probability with fewer hash function invocations and a low memory footprint. A detailed demonstration of the performance of robustBF is presented in Chapter 3.

Chapter 4 exposes the significance of the hash functions in Bloom Filter and helps in understanding their use. Moreover, the chapter also highlights the importance of prime numbers in Bloom Filter design. The chapter experimentally demonstrates the performance of various hash functions. It also compares robustBF and libbloom.

Chapter 5 exposes the relationship among performance, memory, and false positive probability. The chapter demonstrates the approximate false positive probability and exact false positive probability of the conventional Bloom Filter. Also, it demonstrates memory requirements for the conventional Bloom Filter.

Chapter 6 exposes the recent developments of Bloom Filter variants and their applications. It exposes the limitation of the Bloom Filter and its applicability in diverse fields. Moreover, it illustrates the learned Bloom Filter. Furthermore, it demonstrates malicious URL detection using Bloom Filter and deep learning algorithms.

Chapter 7 reviews the standard Bloom Filter. It surveys diverse variants of Bloom Filters and compares them with a quantitative approach. It also discusses the architecture of recently developed Bloom Filters. Chapter 8 discusses the counting variants of Bloom Filters, as well as exposes their architectures. Similarly, Chapter 9 surveys hierarchical Bloom Filters and discusses the architectures of the proposed hierarchical Bloom Filters.

Chapter 10 explores the role of Bloom Filter in networks and communication systems. The exponential increase of Internet users is putting a strain on network traffic. With improvements in technology, users want the fastest

Bloom Filter https://doi.org/10.1016/B978-0-12-823520-1.00006-7

Copyright © 2023 Elsevier Inc. All rights reserved.

response. Hence, the telecommunication industry cannot use traffic delays as an excuse. Therefore, Bloom Filter is an excellent solution for traffic management and measuring traffic statistics. The network traffic is due to the movement of huge volumes of packets. However, some of them are erroneous or malicious and are generated intentionally to hinder the smooth flow of network traffic. Bloom Filter helps in packet management by filtering legitimate packets from malicious and erroneous packets. Bloom Filter also helps routers in enhancing the routing process by fast processing of the packets. Bloom Filter further helps in longest prefix matching by performing the query operation in constant time complexity.

Chapter 11 discusses the role of Bloom Filter in a future networking paradigm called a named-data network. The exponential increment of Internet users and the change in the user requirements from the Internet is leading to new network architecture to satisfy the requirement of the new era users. The chapter provides a brief understanding of the architecture and workings of the named-data network, which does not require the sender or receiver address: it uses the content of the request for data retrieval. The named-data network consists of many data structures that improve the scalability, utilization of multiple network interfaces, security, etc. Bloom Filter can be deployed in all data structures to enhance its efficiency and performance. The request packet is called an Interest packet, and the response packet is called a data packet. Bloom Filter helps in the fastest filtering of the packets and in content discovery. The data structures of named-data networks are content to store, pending interest table, and forwarding information base. The content store is the cache storage for future requests with high read and write frequency. Bloom Filter is used for fast queries to maintain high performance. The pending interest table stores the interest packet and its corresponding data packet receiving interface. It has to handle high-speed incoming packets. Bloom Filter filters these packets and reduces the packet volume. The forwarding information base is used to store information regarding the network, such as the next hop. Bloom Filter is implemented in this data structure for an overall reduction of processing time. Bloom Filter has also helped in improving the security of the named-data network.

Chapter 12 highlights the role of Bloom Filter in software-defined networking. The software-defined networking is proposed to separate the hardware and software components of the network to improve performance. The chapter discusses a brief introduction to the architecture of software-defined networking. The networking separates the data and control planes to upgrade the control and management of the network. However, this network design increases the burden of the controller, which is responsible for control and management. Bloom Filter is deployed in the controller to reduce the burden. Moreover, Bloom Filter helps in flow monitoring and revamping security.

Chapter 13 presents the role of Bloom Filter in wireless communication: wireless sensor network, Mobile Ad-hoc NETwork (MANET), Vehicular Ad-hoc NETwork (VANET), and IoT. The wireless network has a complex architecture as the nodes are mobile. The nodes have to continuously broadcast messages to determine the nodes present in the neighborhood, which increases the network traffic. The chapter presents the reviews of the Bloom Filter-based techniques/protocol implemented on wireless sensors whose primary goal is the collection of environmental information. Along with communication, these sensors have to protect against various cyber-attacks such as eavesdropping attacks, spoofing attacks, and message falsification/injection attacks. Hence, communication in wireless sensors also has to focus on authenticity, confidentiality, integrity, and availability. The chapter reviewed many Bloom Filter-based techniques that provide security to communication in wireless sensors. The chapter also focused on MANET, which is communication among mobile and wireless devices located within a short range, and VANET. Some of the MANET applications, such as battlefield communication and search and rescue, require communication in real time without any delay. The chapter highlights the review of Bloom Filter-based techniques used in such applications. Moreover, the chapter includes a review of Bloom Filter-based IoT techniques.

Chapter 14 comprehends the role of Bloom Filter in network security. In recent years the number of Internet users has increased exponentially; hence, much information is passing through the communication network. This data is personal, important, or of national importance. The huge traffic is increasing the complexity of traffic maintenance and implementation of network security. Furthermore, there is an exponential increase in cyber-attacks. The chapter discusses the Distributed Denial-of-Service (DDoS) attack. The chapter also includes interest and data flooding attacks in Content-Centric Networking under DDoS. The chapter provides reviews of Bloom Filter-based techniques to provide security against DDoS. The chapter briefly explains DoS attacks along with a review of Bloom Filter-based security techniques against DoS. Recently the majority of communication network techniques and protocols have implemented Bloom Filter. Hence, there are many attacks on Bloom Filter. The chapter discusses these attacks. The chapter also highlights and includes reviews on Bloom Filter-based techniques proposed for increasing the security and privacy of communication networks. An experiment is also conducted and discussed in the chapter that shows why Bloom Filter is an excellent barrier against adversaries.

Chapter 15 explores the role of Bloom Filter in data management of Big data. Currently, Big data has expanded beyond volume, velocity, and variety. Its dimension has increased, which further complicates the collection, processing, management, and storage of Big data. This chapter discusses Big data and provides reviews of the Bloom Filter-based techniques for data management of Big data. Bloom Filter is implemented in the database for easy functioning. The chapter also includes reviews of these techniques. MapReduce is a data processing framework deployed in the Hadoop cluster for Big data processing. The chapter briefly discusses MapReduce and has included a review of some Bloom Filter-based techniques to enhance the performance of MapReduce.

Chapter 16 highlights the role of Bloom Filter in cloud computing. Cloud computing provides services of software, platform, and infrastructure through the Internet. This enabled users to access the latest software, technology, etc., without installation in the user system or physical purchase. Cloud computing provides a sense of infinity: infinite software, technology, memory, etc. However, the number of users has increased in substantially. Hence, cloud computing is exploring Bloom Filter to enhance its performance and maintain its quality of services. The chapter includes a discussion on indexing and searching of encrypted data in cloud computing along with a review of techniques based on Bloom Filter. Data storage is an attractive feature of cloud computing because of its infinite storage. However, the service provider has to maintain the data efficiently and reduce redundancy for providing high quality service. The chapter presents a brief discussion and review of the Bloom Filter-based techniques on storage management by cloud computing.

Chapter 17 presents the role of Bloom Filter in biometrics. Nowadays biometric characteristics are explored for authentication to prevent forgery. The human characteristics are usually stored as images in the system, for instance, fingerprints and iris. Image processing is a compute-intensive process where a single image matching with stored images in the system requires a huge time. Therefore, Bloom Filter is explored to enhance the performance of biometric processing. This chapter discusses biometrics and includes a review of Bloom Filter-based biometric indexing techniques. The cancelable biometric is the distortion of original data to enhance security. Bloom Filter is also used in cancelable biometrics. The chapter presents a brief explanation and review of techniques based on the Bloom Filter on cancelable biometrics.

Chapter 18 explores the contribution of Bloom Filter for the enhancement of performance in applications of Bioinformatics, which combines various other available technologies for a better understanding of the mysteries of biology. In the field of computer science, domains such as machine learning and data mining are explored in Bioinformatics. The algorithms and techniques proposed for Bioinformatics are data- and compute-intensive. A short-length fragment of a gene is called a read. The reads in genomic data are highly redundant. This characteristic of genomic data motivated the researcher to explore Bloom Filter. This chapter discusses the pre-processing modules of DNA assembly: k-mer counting, error correction, read compression, and error correction. It also includes a review of techniques based on this topic that implement Bloom Filter. The chapter also briefly explains de Bruijn graphs along with the review of Bloom Filter-based techniques. Bloom Filter is also used in the enhancement of searching for genomic sequences in databases. The chapter provides a review of the Bloom Filter-based indexing schemes. Moreover, the chapter includes a review of the DNA assembly algorithms based on the Bloom Filter. The chapter also includes the implementation of Bloom Filter in other Bioinformatics areas.

References

[1] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426, https://doi.org/10.1145/362686.362692.

[2] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2) (2008), https://doi.org/10.1145/1365815.1365816.

[3] A. Singh, S. Garg, K. Kaur, S. Batra, N. Kumar, K.-K.R. Choo, Fuzzy-folded Bloom filter-as-a-service for big data storage in the cloud, IEEE Trans. Ind. Inform. 15 (4) (2019) 2338–2348, https://doi.org/10.1109/TII.2018.2850053.

[4] S. Nayak, R. Patgiri, A. Borah, A survey on the roles of Bloom Filter in implementation of the Named Data Networking, Comput. Netw. 196 (2021) 108232, https://doi.org/10.1016/j.comnet.2021.108232.

[5] R. Patgiri, S. Nayak, N.B. Muppalaneni, Is Bloom filter a bad choice for security and privacy?, in: 2021 International Conference on Information Networking (ICOIN), 2021, pp. 648–653.

[6] A. Singh, S. Garg, S. Batra, N. Kumar, J.J. Rodrigues, Bloom filter based optimization scheme for massive data handling in IoT environment, Future Gener. Comput. Syst. 82 (2018) 440–449, https://doi.org/10.1016/j.future.2017.12.016, https://www.sciencedirect.com/science/article/pii/S0167739X17314516.

[7] R. Patgiri, S. Nayak, S.K. Borgohain, PassDB: a password database with strict privacy protocol using 3D Bloom filter, Inf. Sci. 539 (2020) 157–176, https://doi.org/10.1016/j.ins.2020.05.135.

[8] R. Patgiri, A. Biswas, S. Nayak, deepBF: malicious URL detection using learned Bloom filter and evolutionary deep learning, Comput. Commun. 200 (2023) 30–41, https://doi.org/10.1016/j.comcom.2022.12.027, https://www.sciencedirect.com/science/article/pii/S0140366422004832.

[9] S. Nayak, R. Patgiri, A review on role of Bloom filter on DNA assembly, IEEE Access 7 (2019) 66939–66954, https://doi.org/10.1109/ACCESS.2019.2910180.

[10] C. Rathgeb, F. Breitinger, C. Busch, H. Baier, On application of Bloom filters to iris biometrics, IET Biometrics 3 (4) (2014) 207–218, https://doi.org/10.1049/iet-bmt.2013.0049.

CHAPTER

2

Bloom Filters: a powerful membership data structure

2.1 Introduction

Bloom Filter [1] is a widely used probabilistic data structure for membership filter. It is applied in many research domains, namely, Computer Networking, Cloud Computing, Big Data, Bioinformatics, and IoT. For instance, BigTable [2] enhances its performance dramatically by deploying Bloom Filter to reduce unnecessary HDD accesses. In BigTable, Bloom Filter is queried to access a data item. If Bloom Filter responds negatively, then the queried data is not a member of the set. As we know, HDD access is costlier than RAM access. Hence, Bloom Filter helps in avoiding HDD accesses. Thus, BigTable is able to enhance its performance drastically. Similarly, Bloom Filter is used to prevent DDoS attacks [3]. Therefore, Bloom Filter is useful in various fields. Bloom Filter attracts many researchers from various fields to enhance on-chip memory consumption. Compared to the conventional hash data structure, Bloom Filter is able to reduce on-chip memory consumption and makes it insignificant. However, unlike a conventional hash data structure, Bloom Filter introduces an error. The errors are classified into false positive and false negative.

2.1.1 Motivation

Bloom Filter has seen wide applications in diverse research fields including interdisciplinary research works. It is a simple data structure, yet very powerful to enhance system performance. Bloom Filter plays the role of a membership filtering, but it is capable of adapting to work on various requirements, for instance, deduplication, query processing, database implementation, etc., which can improve the performance of a system dramatically. Also, Bloom Filter uses a very small amount of main memory, and thus, it is very useful in small, as well as very powerful, devices, for instance, IoT devices and Datacenter. Bloom Filter is providing all these advantages with a time complexity of $O(1)$ for all operations, i.e., insertion, query, and delete. This chapter provides a detailed description of conventional Bloom Filter. It also includes an elaborated discussion on its architecture, issues, and taxonomy. Hence, readers are requested to understand every detail of this chapter, which will help in better understanding of the other chapters.

2.1.2 Contribution

The key contributions of this chapter are given below:

- The chapter thoroughly discusses the architecture of Bloom Filter. Also, we highlight the working mechanism of Bloom Filter.
- The chapter exposes the implementation mechanism of Bloom Filter. There are diverse methods of implementing Bloom Filter, however, we use a simple implementation of Bloom Filter using Murmur hash functions [4] and bitmap.
- The chapter emphasizes the taxonomy of Bloom Filter, which is classified based on characteristics of implementation.
- The chapter analyzes the false positive probability.

Copyright © 2023 Elsevier Inc. All rights reserved.

2.1.3 Organization

The chapter is organized as follows. Section 2.2 demonstrates the architecture and various operations using algorithms. Also, it identifies the hash function that is better for Bloom Filter to implement. Bloom Filter uses bitmap which can be created in many ways. However, we have demonstrated that bitmap can be created using integer values, which is given in Section 2.3. Bloom Filter can respond either true or false in a query, and this response is further classified into four categories discussed in Section 2.4. This response creates issues of having false positives and false negatives, and hence, Section 2.5 highlights the key objectives of Bloom Filters. Moreover, this chapter provides detailed classifications of Bloom Filter based on their characteristics which is discussed in Section 2.6. Most importantly, the false positive issue is a grand challenge for Bloom Filter, and its analysis is presented in Section 2.7. Another important feature of this chapter is lessons learned, which is discussed in Section 2.8, to utilize the experience gained by the authors. Finally, the chapter is concluded in Section 2.9.

2.2 Bloom Filter

Bloom Filter [1] is a probabilistic data structure to test a membership of an item in a set [5]. Bloom Filter stores the information of the item set using an insignificant space overhead. The true positive and true negative response enhances the performance of the filtering mechanism. And, false positive and false negative introduces overhead in the Bloom Filter. However, Bloom Filter has a negligible false positive probability. But, this negligible false positive probability cannot be tolerated by some systems because the impact of such a small error can be very devastating for a particular system. Bloom Filter has another issue, namely that of false negative response. It is caused by the deletion of an element from Bloom Filter. However, some variants of Bloom Filter guarantee that there is no false negative.

Definition 2.1. *Bloom Filter is a probabilistic data structure with some error tolerance that returns either true or false.*

2.2.1 Architecture of Bloom Filter

Bloom Filter was first proposed by Burton Howard Bloom in 1970 [5]. Fig. 2.1 depicts the architecture of Bloom Filter. A conventional Bloom Filter $\mathbb{B}[]$ is a bit array of size μ. The array can only assign zeros or ones. The number of hash functions is K. In Fig. 2.1, the hash functions are $\mathcal{H}_1()$, $\mathcal{H}_2()$, and $\mathcal{H}_3()$, and the K value is 3. Initially, the array is initialized to 0.

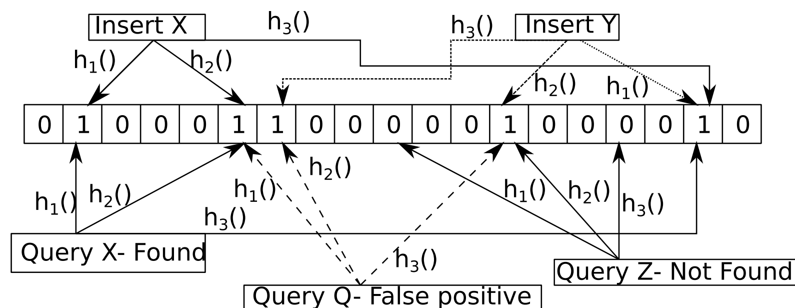


FIGURE 2.1 Architecture of conventional Bloom Filter.

2.2.2 Operations of Bloom Filter

Bloom Filter supports three key operations, namely, insert, lookup, and delete, as shown in Fig. 2.2. However, the delete operation introduces the possibility of a false negative into the Bloom Filter. Therefore, the conventional Bloom Filter does not permit the delete operation. However, other variants of Bloom Filter permit the delete operation without increasing the number of false negatives.

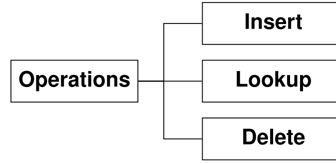


FIGURE 2.2 Operations of Bloom Filter.

2.2.2.1 Insertion

Bloom Filter stores bit information about an input item in a bitmap array. Any input item is hashed into a specific slot of the bitmap array and that slot is set to '1' in the insertion operation. As shown in Fig. 2.1, input item X is hashed by three hash functions (i.e., $\mathcal{H}_1()$, $\mathcal{H}_2()$, and $\mathcal{H}_3()$) and generates three bit locations. These locations are set to 1. Similarly, the input item Y is inserted. Any hash function can be used to insert an item into Bloom Filter, namely, CRC32, MD5, FNV, DJB, etc. The Murmur hash function is the fastest hash algorithm among the string hash functions including FNV1, FNV1a, CRC32, DJB, DJB2a, SuperFastHash, and xxHash. The Murmur hash function has three versions, Murmur, Murmur2, and Murmur3 [4]. Algorithm 2.1 uses Murmur hash function to hash the input items into Bloom Filter. For the input item κ, the string length of κ is $length$ and seed values S_1, S_2, and S_3 are passed to Murmur hash function. The seed values are prime numbers. The Murmur hash function returns a ten-digit number. The ten-digit number is hashed using μ where μ is the size of a Bloom array. The insertion operation performance of Bloom Filter depends on the number of hash functions. Therefore, the time complexity of the insertion operation is $O(K)$ where K is the number of hash functions. Also K is constant, therefore, the time complexity of the insertion operation is $O(1)$. The performance of Bloom Filter depends on the Murmur hash function and the number of modulus operations in Algorithm 2.1. We must avoid multiple calls to the hash function to enhance the performance of the Bloom Filter.

Algorithm 2.1 Insertion of an input item κ into Bloom Filter $\mathbb{B}$ using three hash functions.

1: **procedure** INSERT($\mathbb{B}[]$, κ, S_1, S_2, S_3) ▷ S_1, S_2, S_3 are seeds used to create different hash functions
2: $i_1 = \text{MURMUR}(\kappa, length, S_1) \% \mu$ ▷ μ is the size of $\mathbb{B}[]$ which is a prime number.
3: $i_2 = \text{MURMUR}(\kappa, length, S_2) \% \mu$ ▷ % is modulus operator
4: $i_3 = \text{MURMUR}(\kappa, length, S_3) \% \mu$ ▷ i is index of the bit array $\mathbb{B}[]$
5: $\mathbb{B}[i_1] \leftarrow 1$ ▷ Key is hashed into Bloom Filter array $\mathbb{B}[]$ to insert the input item κ
6: $\mathbb{B}[i_2] \leftarrow 1$
7: $\mathbb{B}[i_3] \leftarrow 1$
8: **end procedure**

2.2.2.2 Lookup

As shown in Fig. 2.1, a lookup operation is invoked for an item X. Algorithm 2.2 demonstrate a lookup operation by invoking three Murmur hash functions [4]. Similar to the insertion operation, κ is hashed by K hash functions. The bit value of the location generated by the hash function is checked. If all bits are set to 1, then Bloom Filter returns $True$, otherwise it returns $False$. The AND-ing operations are performed to examine whether the item κ is a member of the Bloom Filter or not, i.e., $\mathbb{B}[v_1]$ AND $\mathbb{B}[v_2]$ AND $\mathbb{B}[v_3]$. If any slot contains 0 value, then the item is not a member of the Bloom Filter and the Bloom Filter returns $False$. Similar to the insertion operation, the time complexity of a lookup operation depends on the number of hash functions. Therefore, the time complexity of lookup operation is $O(K)$ where K is the number of hash functions. As K is constant, the time complexity of the lookup operation is $O(1)$.

2.2.2.3 Deletion

Algorithm 2.3 performs a deletion operation on non-counting Bloom Filter. Before deleting an item, a lookup operation is performed. If the item exists in the Bloom Filter, then the algorithm resets the bit values to 0. The bit locations are obtained as discussed for the insertion and lookup operations. Bloom Filter may reset the bit location to '0' in the case of an absent item, if the lookup operation is not performed before the deletion operation. To elaborate,

Algorithm 2.2 Lookup an item κ in Bloom Filter using three hash functions.

```
1:  procedure LOOKUP(𝔹[], κ, S1, S2, S3)
2:      i1 = MURMUR(κ, length, S1) % μ
3:      i2 = MURMUR(κ, length, S2) % μ
4:      i3 = MURMUR(κ, length, S3) % μ
5:      if 𝔹[i1] AND 𝔹[i2] AND 𝔹[i3] then        ▷ AND-ing operation is perform to check the bit values of 𝔹[].
6:          return True
7:      else
8:          return False
9:      end if
10: end procedure
```

let us assume $\mathbb{B}[i_1] = 1$, $\mathbb{B}[i_2] = 0$, and $\mathbb{B}[i_3] = 1$. In this case, $\mathbb{B}[i_1]$ and $\mathbb{B}[i_3]$ are reset to zero unnecessarily, albeit the item was not actually present in the Bloom Filer. This causes a false negative issue. Thus, the lookup operation is performed before removal of an item from Bloom Filter. However, this process is not necessarily free from false negatives due to collision probability in the deletion operation. For instance, let us assume $\mathbb{B}[i_1] = 1$, $\mathbb{B}[i_2] = 1$, and $\mathbb{B}[i_3] = 1$, and suppose the item was not actually inserted into Bloom Filter. The slots are exhibiting value '1' due to the false positives. In the deletion operation, the slots are reset to '0' assuming that the item was inserted into the Bloom Filter while it was not the case. Thus, conventional Bloom Filter exhibits a false negative if and only if it supports the deletion operation. Therefore, the deletion operation is not permitted in the conventional Bloom Filter. To permit the deletion operation, counting Bloom Filter was introduced. The time complexity of the deletion operation is similar to that of the insertion and lookup operations, i.e., $O(1)$.

Algorithm 2.3 Deletion of an item κ from Bloom Filter using three hash functions.

```
1:  procedure DELETE(𝔹[], κ, S1, S2, S3)
2:      i1 = MURMUR(κ, length, S1) % μ
3:      i2 = MURMUR(κ, length, S2) % μ
4:      i3 = MURMUR(κ, length, S3) % μ
5:      if 𝔹[i1] AND 𝔹[i2] AND 𝔹[i3] then
6:          𝔹[i1] ← 0          ▷ Key is hashed into Bloom Filter array 𝔹[] to reset to 0 for the input item κ
7:          𝔹[i2] ← 0
8:          𝔹[i3] ← 0
9:      else
10:         False
11:     end if
12: end procedure
```

2.3 Bit array

Bit Array is an array of bits where every cell of the array occupies a 1-bit. Bit Array can store either 0 or 1. There is no library of Bit Array. Therefore, we have to create own Bit Array to manipulate Bloom Filter. The most crucial part of Bloom Filter is creating a Bit Array. If we represent a Bit Array using a character array, say **char**[], then each cell occupies one byte. Bloom Filter is used to reduce memory consumption. Therefore, this process becomes very costly in terms of memory. The most simple way to create a Bit Array is by using **unsigned long int**[]. Let us create Bit Array using **unsigned long int**[]. In this case, each cell of the array occupies 64 bits. Therefore, each bit is used to store information about an input item in Bloom Filter. Let $\mathbb{B}_\mu$ be the Bit Array of size μ where μ is a prime number. Now, we would like to insert κ into $\mathbb{B}_\mu$. Therefore, we need to call a hash function $\mathcal{H}()$, and so, $h \leftarrow \mathcal{H}(\kappa)$ where h is the hash value returned by the hash function $\mathcal{H}(\kappa)$. Now, $i = h \% \mu$ and $\rho \leftarrow h\%63$ where i is the index of **unsigned long int** array and ρ is the position of the bit. Then, Eq. (2.1) is invoked to insert κ. Eq. (2.2) is invoked to perform

the lookup operation of κ:

$$\mathbb{B}_i \leftarrow \mathbb{B}_i \text{ OR } (1 \ll \rho), \tag{2.1}$$

$$Flag \leftarrow \mathbb{B}_i \text{ AND } (1 \ll \rho). \tag{2.2}$$

Together Eqs. (2.1) and (2.2) minimize costly operations and use fast operations, like bitwise operators. However, the % (modulus operator) is very costly and it cannot be avoided due to hashing data structure. Further details are provided in Chapter 5.

2.4 Taxonomy of response

Bloom Filter returns two types of response, namely, $True$ and $False$. $True$ means that the queried item is present in Bloom Filter. $False$ shows that the queried item is absent from Bloom Filter. Based on some situations, the $True$ and $False$ response is further classified as shown in Fig. 2.3. $True$ is classified into True Positive and True Negative. Similarly, $False$ is classified into False Positive and False Negative. Let $\mathbb{S}$ be a set, $\kappa = \kappa_1, \kappa_2, \kappa_3, \ldots, \kappa_\mu$ be the input items, where $\kappa \in \mathbb{S}$, and suppose μ is the total number of elements [6,7]. Let κ_i be a random query element where $i = 1, 2, 3, 4, \ldots$, and $\mathbb{B}$ be the Bloom Filter, and therefore, true positive, false positive, true negative, and false negative are defined as follows.

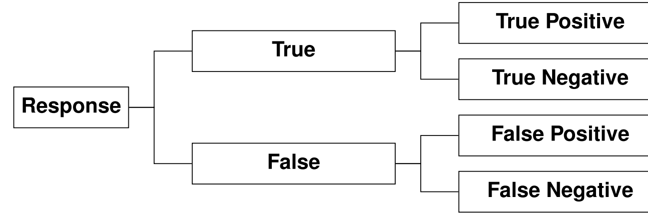


FIGURE 2.3 Taxonomy of Response.

Definition 2.2 (True Positive). *If $\kappa_i \in \mathbb{B}$ and $\kappa_i \in \mathbb{S}$, then the response of Bloom Filter is a true positive.*

Definition 2.3 (True Negative). *If $\kappa_i \notin \mathbb{B}$ and $\kappa_i \notin \mathbb{S}$, then the response of Bloom Filter is a true negative.*

Definition 2.4 (False Positive). *If $\kappa_i \in \mathbb{B}$ and $\kappa_i \notin \mathbb{S}$, then the response of Bloom Filter is a false positive.*

Definition 2.5 (False Negative). *If $\kappa_i \notin \mathbb{B}$ and $\kappa_i \in \mathbb{S}$, then the response of Bloom Filter is a false negative.*

2.4.1 True positive

The True Positive response enhances the performance of the Bloom Filter. Suppose that in Fig. 2.1 the items X and Y are inserted into $\mathbb{B}[]$. In the figure, the lookup operation for item X is an example of true positive. Upon the lookup operation of X, it is hashed. The hashed locations have value 1. Therefore, X is present in Bloom Filter. To illustrate more clearly the true positive, let us take an example of the COVID-19 test. The person is suffering from COVID-19. He/she is tested for COVID-19 and the result is given as positive. Then, the result is a true positive.

2.4.2 True negative

The True Negative response eliminates unnecessary searching overhead. Suppose that before searching for data in the system the Bloom Filter is searched. Then, if Bloom Filter gives a $False$ response, it indicates that data is absent from the system. Therefore, the data is not searched in the system. In Fig. 2.1, the lookup operation for item Z is an example of a true negative; Z is hashed and the slots have value 0. Therefore, $False$ response is returned. Suppose a person tests for COVID-19 where the person is not infected with the coronavirus. If the result is negative, then it is a true negative.

2.4.3 False positive

The False Positive is an overhead in Bloom Filter. Indeed, suppose the Bloom Filter returns $True$. After a query for an absent item, the system will search for the item. As the item is not inserted into the system, it will return $False$ after searching the whole system. Therefore, due to a wrong response, the system has to waste time searching for an absent data item. Suppose, in Fig. 2.1, only items X and Y are inserted into $\mathbb{B}[]$. Upon the lookup operation of Q, it is hashed. The $Q_{\mathcal{H}_1()}$ (slot location of $\mathcal{H}_1(Q)$) is the same as $X_{\mathcal{H}_2()}$. Similarly, $Q_{\mathcal{H}_2()}$ and $Q_{\mathcal{H}_3()}$ are the same as $Y_{\mathcal{H}_3()}$ and $Y_{\mathcal{H}_2()}$, respectively. During the insertion of items X and Y, the slots are assigned to 1. Hence, during the lookup for item Q, the Bloom Filter returns $True$. Similar situations give false positive responses. Moreover, when the Bloom Filter becomes saturated, the False Positive probability increases. Let us assume that a person tests for COVID-19 where he/she is not suffering from COVID-19. But the result shows positive. In this case, the result is a false positive.

2.4.4 False negative

The False Negative is more dangerous compared to False Positive. A false negative indicates that an item is absent, whereas the item is present in Bloom Filter. Due to a false negative, many domains do not implement Bloom Filter, for instance, password systems. However, PassDB [8] is a password database that has implemented Bloom Filter. A false negative occurs due to the delete operation. Due to a delete request, the slot is reset to 0. However, when an item has a common slot with the deleted item, then, as one slot became 0, it returns $False$. Suppose a person wishes to test for COVID-19 and he/she is actually suffering from COVID-19. But the result shows negative. In this case, the result is a false negative.

Theorem 2.1. *The delete operation causes false negatives in the conventional Bloom Filter.*

Proof. A false negative is a situation where the item is already inserted, but upon query the Bloom Filter returns $False$. Let us refer to Fig. 2.1 to prove this theorem. In the figure items X and Y are both inserted. As shown in the figure, both $X_{\mathcal{H}_3()}$ (slot location of $\mathcal{H}_3(X)$) and $Y_{\mathcal{H}_1()}$ (slot location of $\mathcal{H}_1(Y)$) have the same hash value. Now, delete item Y. It means that all the slot values obtained by the three hash functions are reset to 0. Thus, $Y_{\mathcal{H}_1()}$ slot is also reset to 0. Then, if the X item is queried, then Bloom Filter finds one slot as 0. Hence, Bloom Filter concludes that the item is not inserted. In such situations Bloom Filters give incorrect response. Thus, the delete operation causes false negatives in the conventional Bloom Filter. ☐

2.5 Key objectives

The prime objectives of developing a new variant of Bloom Filter are depicted in Fig. 2.4, which are elaborated below. The key objectives are to (a) reduce the number of false positives, (b) reduce the number of false negatives, (c) increase scalability, and (d) maximize the performance of Bloom Filter.

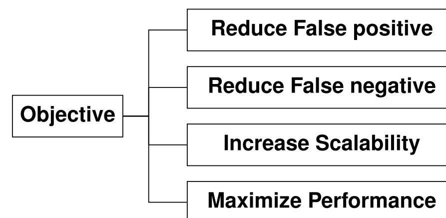


FIGURE 2.4 Key objectives of Bloom Filters.

2.5.1 Reduction of the number of false positives

Section 2.7 provides detailed mathematical analysis of the false positive probability. In this section, we will discuss why we should reduce the false positive probability. For example, the deduplication process filters out the duplicate

items using Bloom Filter. But, due to a false positive, a unique item is filtered out. Therefore, the deduplication process needs a Bloom Filter with high accuracy, which lowers the false positive rate. In this case, a unique item can be added again in the deduplication process, albeit there is a false negative. However, the duplicate item is added again due to the false negative, which is not an error, but it is an overhead. For example, BigTable [2] uses Bloom Filter to increase its performance. A query is submitted to the BigTable. First, BigTable examines the data existence of the queried item in the Bloom Filter. If Bloom Filter returns a false positive, then BigTable further query tries to access the HDD, which causes a significant overhead. If BigTable encounters many false positive results, then it will slow down, because BigTable uses extra lookup time in Bloom Filter. Now, let us assume that there is no false positive in the Bloom Filter. Then, there will be true positives and true negatives, which will further improve the BigTable performance drastically. Therefore, reduction of the number of false positives is very important in designing a new variant of Bloom Filter. Currently, a high-accuracy Bloom Filter is developed, called HFil [9], which gives very low false positive probability.

2.5.2 Reduction of the number of false negatives

Similar to the reduction of the number of false positives, the reduction of the number of false negatives is also important in designing a new Bloom Filter. Due to false negatives, many applications have not adapted Bloom Filter. For instance, identity management systems do not use Bloom Filter because in such systems a user is unable to access its own account due to false negatives after some period of time. However, recent research shows that Bloom Filter is also used in identity management systems, for instance, PassDB [8] deploys a high-accuracy Bloom Filter to reduce the number of false positives and does not permit the deletion operation to avoid false negatives. The counting Bloom Filter is able to reduce the number of false negatives significantly [10]. However, the counting Bloom Filter suffers from a high false positive probability. Therefore, the counting Bloom Filter is used only when a false negative creates an error.

2.5.3 Increment of scalability

In the Big data era, millions of data items are filtered by Bloom Filter. Therefore, scalability becomes a prominent research topic for Bloom Filter. There are numerous ways to enhance the scalability of the Bloom Filter. First, the dynamic Bloom Filter adjusts the Bloom Filter sizes dynamically. However, the memory reallocation causes a segmentation fault. Therefore, hierarchical Bloom Filter is the best option to increase the scalability of the Bloom Filter. Nevertheless, the hierarchical Bloom Filter is slower than other variants of Bloom Filter. A chained approach is used in scaleBF [11]. If Bloom Filter is saturated, then we add another Bloom Filter to increase the scalability. However, scaleBF disapproves a linked list structure. It implements the chain hash data structure to reduce the time complexity.

2.5.4 Maximization of performance

False positive and false negative are two main issues of Bloom Filter. Therefore, there are numerous variants of Bloom Filter to reduce the numbers of false positives and false negatives. However, the performance of Bloom Filter is affected when reducing the numbers of false positives and false negatives. Moreover, increasing scalability is also a challenge because the applications of Bloom Filter demand large-scale Bloom Filter, for instance, deduplication. There is a trade-off between the number of false positives and performance. A few Bloom Filters sacrifice performance to have fewer false positives, for example, Cuckoo Filter [12]. Also, the space consumption increases when reducing the number of false positives.

2.6 Taxonomy of Bloom Filter

Fig. 2.5 clearly classifies the types of Bloom Filter depending on the characteristics. Bloom Filter poses different characteristics depending on architecture, memory allocation, implementation, and platform.

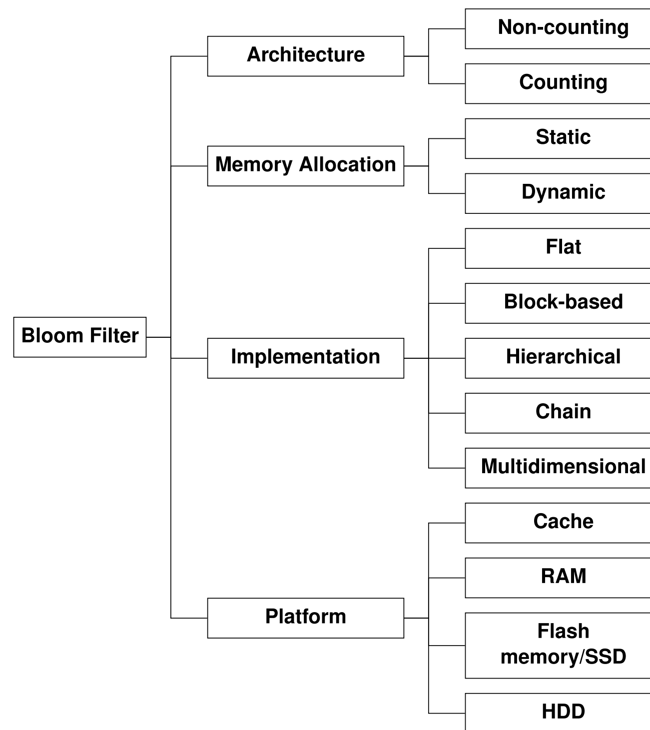


FIGURE 2.5 Taxonomy of Bloom Filters.

2.6.1 Architecture

Bloom Filter is classified into two key categories, particularly, counting Bloom Filter and non-counting Bloom Filter, depending on the architecture of the Bloom Filter.

2.6.1.1 Counting Bloom Filter

Counting Bloom Filter typically counts the number of inputs and increment the counter upon insertion of an item. When the same input is inserted, the counter is incremented. When the same item is deleted, then the counter is decremented. However, if the counter size is static, the number of false negatives increases due to counter overflow. For example, let us assume $X = 15$ where X is the value of the counter for certain states. The counter is at the maximum value. After one more insertion of same item, the counter cannot be incremented and counter overflow occurs. Suppose the total number of times the item is inserted is 20. Then $X = 0$ after decrementing 15 times, i.e., after the delete operation is executed 15 times on the same item. Then, upon query for the same item, counting Bloom Filter will return $False$, i.e., the item is absent. However, the item is inserted 20 times in total and, after deleting 15 times, there are additional 5 times the item is present. Thus, the counting Bloom Filter has fewer false negatives, but more false positives. Counting Bloom Filter is discussed more in detail in Chapter 4.

2.6.1.2 Non-counting Bloom Filter

Non-counting Bloom Filter typically does not have any counters to count the inputs. Conventional Bloom Filter is an example of a non-counting Bloom Filter. Unlike counting Bloom Filter, the non-counting Bloom Filter sets the bit to '1' at the insertion of an item, and resets the bit to '0' at the deletion of an item. In a query, non-counting Bloom Filter responds $True$ if the corresponding bits are set to '1', otherwise, it returns $False$. The non-counting Bloom Filter may be extended to hierarchical, multidimensional, or blocked-base Bloom Filter. However, the non-counting Bloom Filter has more false negatives if it allows the deletion operation. Therefore, a non-counting Bloom Filter usually restricts the deletion operation.

2.6.2 Memory allocation

We classify the Bloom Filter into two types based on the memory allocation strategy, specifically, static Bloom Filter and dynamic Bloom Filter.

2.6.2.1 Static Bloom Filter

The static Bloom Filter can allocate memory, statically or dynamically. However, the static Bloom Filter allocates memory once, and does not allow increasing the allocated memory size. Conventional Bloom Filter and conventional counting Bloom Filter both are examples of a static Bloom Filter, which does not allow altering the parameters of the Bloom Filter. Therefore, its architecture and implementation are simple. However, another instance of a static Bloom Filter can be created on demand to meet the requirement of scalability. However, when Bloom Filter becomes saturated, the false positive probability increases because most of the bits of the array are set to 1. Therefore, as a static Bloom Filter does not increase its size, eventually it will be saturated and will give false positive responses.

2.6.2.2 Dynamic Bloom Filter

Dynamic Bloom Filter allocates memory as per the requirement of the current situation. Later, the allocated memory is increased to readjust current requirements. Dynamic Bloom Filter grows in size over a time period and adjusts automatically without interference of the programmer. However, the dynamic Bloom Filter requires adjustment of the Bloom Filter to keep the same rules of insertion, lookup, and deletion. These requirements make designing dynamic Bloom Filter more complex than static Bloom Filter. In some implementations, resizing the Bloom Filter requires the reconstruction of the whole Bloom Filter.

Theorem 2.2. *Static Bloom Filters are faster than dynamic Bloom Filters.*

Proof. A static Bloom Filter allocates memory using dynamic memory allocation techniques, but having it allocated the filter does not change the memory size for a lifetime, whereas a dynamic Bloom Filter readjusts the memory on a time-to-time basis. Thus, a dynamic Bloom Filter is scalable as opposed to a static Bloom Filter. Due to readjustment, the dynamic Bloom Filter needs to alter all parameters, and hence, there is a readjustment cost. Moreover, the old Bloom Filter must copy to the new Bloom Filter, which takes time. Moreover, the dynamic Bloom Filter is more complex than the static Bloom Filter. Due to readjustment, the dynamic Bloom Filter is slower than the static Bloom Filter. □

Theorem 2.3. *Dynamic Bloom Filter encounters reallocation of memory.*

Proof. A dynamic Bloom Filter readjusts the memory size if the filter is half full or reached a threshold. New memory occupies a larger space than that of the earlier Bloom Filter. The old data from Bloom Filter are mapped into the new Bloom Filter, while a static Bloom Filter creates a new Bloom Filter of the same size and both are linked to each other. However, the dynamic Bloom Filter requires a larger memory than its predecessor. Therefore, if we repeat this process again and again, then the operating system will throw a segmentation fault error. However, it will happen only in the large-scale filtering. Thus, over a time period, the dynamic Bloom Filter may encounter an error. □

2.6.3 Implementation

Based on implementation, we classify Bloom Filter into five categories, namely, flat Bloom Filter, block-based Bloom Filter, hierarchical Bloom Filter, chained Bloom Filter, and multidimensional Bloom Filter.

2.6.3.1 Flat Bloom Filter

Flat Bloom Filter uses an array of bits. Standard or conventional Bloom Filter is an example of a flat Bloom Filter. The bit array contains information about membership of large datasets. It does not have any special arrangement of bit arrays. Therefore, it is called a flat Bloom Filter. However, the flat Bloom Filter can be utilized to derive new variants of a Bloom Filter. A flat Bloom Filter is a simple form of a membership filter, yet very powerful. It is a highly adaptable data structure.

2.6.3.2 Block-based Bloom Filter

Block-based Bloom Filter creates multiple blocks on the given bit array to reduce both collision and false positive probability. However, it is similar to a flat Bloom Filter. A block-based Bloom Filter logically arranges several blocks,

and items are placed on the basis of blocks. Thus, the block-based Bloom Filter is able to reduce the number of false positives. One simple example is a counting Bloom Filter which uses one slot to store a bit and an adjacent slot to store a counter. A block-based Bloom Filter is not free from false negatives if it allows the deletion operation.

2.6.3.3 Hierarchical Bloom Filter

A hierarchical Bloom Filter consists of multiple Bloom Filters and forms a tree-like structure. Each node of the hierarchical Bloom Filter is embedded with a conventional Bloom Filter. The hierarchical Bloom Filters greatly help in very large-scale membership filtering. For instance, B-Tree can be used to implement a hierarchical Bloom Filter. However, the lookup and insertion cost increases. The main goal of the hierarchical Bloom Filter is to increase scalability.

2.6.3.4 Chained Bloom Filter

A chained Bloom Filter creates a chain of multiple Bloom Filters to increase scalability, i.e., a chained Bloom Filter creates a chain of Bloom Filters. A new Bloom Filter is linked to a chained Bloom Filter to increase the scalability. The chained Bloom Filter is categorized into two subcategories, namely, hash-based chained Bloom Filter and linked-list-based chained Bloom Filter. The linked-list-based chained Bloom Filter increases time complexity of insertion and lookup. However, the hash-based chained Bloom Filter uses open-hashing scheme to reduce the lookup and insertion time complexity. Bloom Filter must provide detection of the fullness of the Bloom Filter in both cases to create another Bloom Filter and be linked with the chained Bloom Filter. For example, scaleBF is an example of a hash-based chained Bloom Filter [11].

2.6.3.5 Multidimensional Bloom Filter

On the contrary, a multidimensional Bloom Filter is implemented using a multidimensional Bloom Filter array, for instance, a 3D array. The multidimensional Bloom Filter is similar to a flat Bloom Filter except for its dimension. rDBF is the first multidimensional Bloom Filter [6]. It uses **unsigned long int** in cells of the arrays (2D, 3D, 4D, 5D, etc.) and the numbers are manipulated using bitwise operators. Thus, the rDBF outperforms many Bloom Filters. As we know, bitwise operators are the fastest operators. Also, the performance depends on the number of hash functions. Thus, rDBF avoids many function calls and uses only single Murmur hashing function calls. Moreover, rDBF optimizes the number of modulus operators. Hence, its performance rises significantly.

2.6.4 Platform

Finally, we categorize the Bloom Filter into four key categories based on platform, namely, cache, RAM, Flash/SSD, and HDD.

2.6.4.1 Cache-based Bloom Filter

A cache-based Bloom Filter is designed to improve the caching performance. The block size of a Bloom Filter is ought to be small such that the block size fits in the cache memory. The cache-aware Bloom Filter is designed to enhance the cache performance. However, the cache-aware Bloom Filter can also be implemented in RAM by considering the cache hit and miss probabilities.

2.6.4.2 RAM-based Bloom Filter

On the other hand, RAM-based Bloom Filters are the most popular. Such filters are stored in RAM, which further increases the efficiency of Bloom Filters since their access is faster when stored in RAM. Therefore, the insertion, lookup, and delete operations can easily be performed with $O(1)$ time complexity. Most of the Bloom Filters are implemented in RAM except for a few, since Bloom Filter is a data structure. However, a RAM-based Bloom Filter does not ensure consistency and persistence. If a machine fails, then the Bloom Filter will be wiped out. Therefore, the Bloom Filter is periodically buffered in permanent storage devices, for instance, SSD or HDD.

2.6.4.3 Flash/SSD-based Bloom Filter

Bloom Filter is implemented in RAM. Therefore, it is not persistent. Thus, Flash/SSD is used to increase the scalability and fault tolerance of Bloom Filter implementation. However, the Flash/SSD memory cannot be used similarly as RAM. In a random write operation, Bloom Filter waits for some time to update the bit information in Flash/SSD devices because Flash/SSD is slower than RAM. Hence, a single bit update becomes costly in terms

of latency. This process is known as lazy update. A frequent update operation may degrade the performance of a system. Moreover, the SSD is used to increase the scalability of a Bloom Filter. There are many separate Bloom Filters which are created and all are not required. Similar to demand paging, a required Bloom Filter is brought into RAM and another Bloom Filter will not be loaded into RAM.

2.6.4.4 HDD-based Bloom Filter

However, HDD is also used to backup the filter to implement the persistence of a Bloom Filter. If the system shuts down, then the Bloom Filter can be reconstructed from the backup which is stored in HDD. However, the HDD is slower than SSD/Flash. Both HDD and Flash/SSD can adapt lazy update methods.

2.7 Analysis of false positive

The false positive is an overhead that affects the performance of a Bloom Filter as shown in Fig. 2.6, which discloses the relation between the number of hash functions and the probability of bits to be 1. Let us assume that μ is the number of bits in the array. The probability of a specific bit to be 1 is $\frac{1}{\mu}$. The probability of a specific bit to be 0 is

$$1 - \frac{1}{\mu}. \tag{2.3}$$

Let K be the number of hash functions. The probability of that a specific bit remain 0 is [13,5]

$$\left(1 - \frac{1}{\mu}\right)^K. \tag{2.4}$$

If there are η elements inserted into the Bloom Filter array, then the probability of that specific bit to be 0 is

$$\left(1 - \frac{1}{\mu}\right)^{\eta K}. \tag{2.5}$$

Now, the probability of that bit to be 1 is

$$1 - \left(1 - \frac{1}{\mu}\right)^{\eta K}. \tag{2.6}$$

The probability of all bits to be 1 is

$$\left(1 - \left(1 - \frac{1}{\mu}\right)^{\eta K}\right)^K \approx \left(1 - e^{-\eta K/\mu}\right)^K. \tag{2.7}$$

The probability of false positive increases with an increase in the number of entries, η. However, it is lowed by enhancing the value of μ. What is the optimal number of hash functions K? The optimal number of hash functions K is

$$K = \frac{\mu}{\eta} \ln 2. \tag{2.8}$$

Let P be the desired false positive probability. Then

$$P = \left(1 - e^{-(\frac{\mu}{\eta} \ln 2\eta)/\mu}\right)^{\frac{\mu}{\eta} \ln 2}, \tag{2.9}$$

$$\ln p = -\frac{\mu}{\eta} (\ln 2)^2, \tag{2.10}$$

$$\mu = -\frac{\eta \ln p}{(\ln 2)^2}, \tag{2.11}$$

$$\frac{\mu}{\eta} = -\frac{\eta \log_2 p}{\ln 2} \approx -1.44 \log_2 p. \tag{2.12}$$

Therefore, the number of optimal hash functions required is

$$K = -1.44 \log_2 p. \tag{2.13}$$

F. Grandi [5] presented an analysis of false positive probability (FPP) through γ-transformations. Let X be the random variable corresponding to the total number of set bits in the array of a Bloom Filter, then

$$E[X] = \mu \left(1 - \left(1 - \frac{1}{\mu}\right)\right)^{\eta K}. \tag{2.14}$$

Let us condition the random variable $X = x$ to determine the false positive probability, then

$$Pr(FPP|X = x) = \left(\frac{x}{\mu}\right)^{K}. \tag{2.15}$$

The total probability theorem gives the approximate false positive probability as follows:

$$FPP = \sum_{x=0}^{\mu} Pr(FP|X = x)\ Pr(X = x) \tag{2.16}$$

$$= \sum_{x=0}^{\mu} \left(\frac{x}{\mu}\right)^{K} f(x), \tag{2.17}$$

where $f(x)$ is a probability mass function. F. Grandi [5] applied γ-transformation to calculate the value of $f(x)$, and then the exact false positive probability is

$$FPP = \sum_{x=0}^{\mu} \left(\frac{x}{\mu}\right)^{K} \binom{\mu}{x} \sum_{j=0}^{x} (-1)^{j} \binom{x}{j} \left(\frac{x-j}{\mu}\right)^{\eta K}. \tag{2.18}$$

Eq. (2.18) represents the approximate false positive probability of a conventional Bloom Filter.

Fig. 2.6 depicts the behavior of the false positive probability with respect to the load factor and the number of hash functions.

Fig. 2.7 depicts the number of optimal hash functions required in a load factor $\alpha \leftarrow \frac{m}{n}$. The straight line depicts the optimal number of hash functions. The Y-axis represents value of K. Concluding from Fig. 2.7, the optimal number of hash functions must be less than the load factor. Otherwise, the false positive probability increases.

2.8 Lessons learned

Bloom Filter is a variant of hash data structures. Therefore, it depends on prime numbers while hashing. Otherwise, collisions may arise, which result in larger false positive probabilities. Therefore, the Bloom Filter size requires prime numbers. Moreover, the Murmur hash function is the fastest non-cryptographic string hash function. Apparently, the Murmur hash function can enhance the performance of a Bloom Filter. Also, enhancing the Murmur hash function can enhance the performance of a Bloom Filter drastically. The source code of the Murmur hash function is available in [4]. Therefore, it is required to enhance the performance of the Murmur hash function. In addition, Bloom Filter depends on the number of hash functions. If the total number of hash functions is very high then it will degrade the performance of the Bloom Filter. If the total number of hash functions is very low, then it will increase the false positive probability. Therefore, the total number of hash functions is chosen very carefully. Also, the false positive probability depends on the number of bits available for the incoming input items. Over a time period, the Bloom Filter will become saturated if it does not allow the deletion operation and readjustment of memory size. Therefore, the Bloom Filter requires auto-readjustment of memory size depending on the number of inputs.

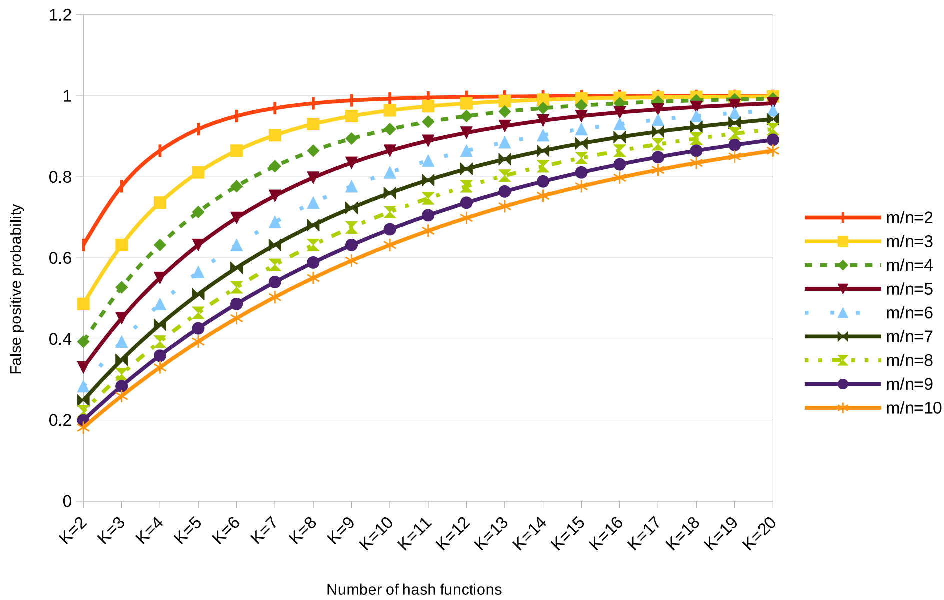


FIGURE 2.6 False positive probability.

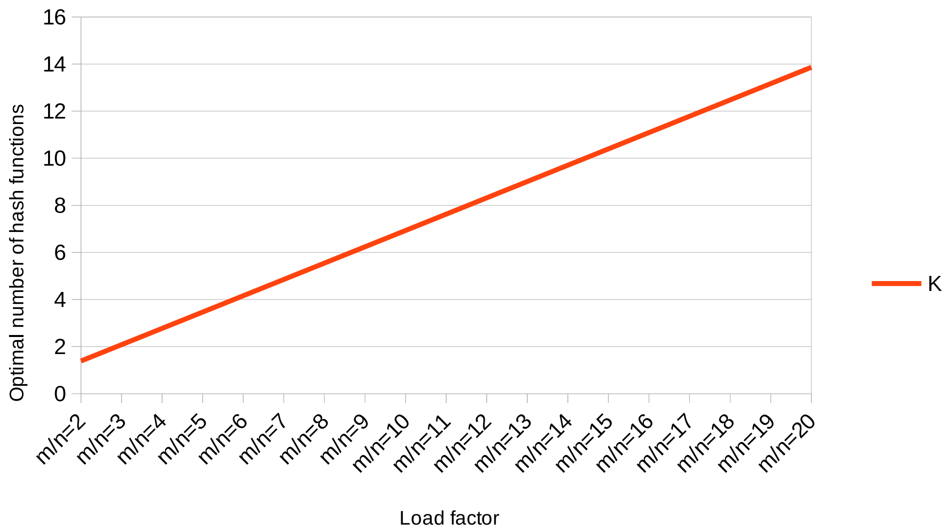


FIGURE 2.7 Optimal number of hash functions.

2.9 Conclusion

Bloom Filter is a famous data structure to deploy in various fields. It is utilized to enhance systems' performance drastically using a tiny amount of main memory. Moreover, it can accommodate a massive amount data using a small amount of memory. Bloom Filter is known as a quick set-membership filter to lookup an item, whether an item is a member of a set or not. Bloom Filter returns either $True$ or $False$ in a query for an item. A true result can be either a true positive or a false positive. Similarly, a false result can be either a true negative or a false negative. A true negative and a true positive result enhance the query performance of a system dramatically while a false positive and a false negative affect the performance of a system. Therefore, the false positive and false negative are the overheads of a Bloom Filter. False negatives occur only when a Bloom Filter allows the deletion operation. On the contrary, a counting Bloom Filter is able to reduce the number of false negatives significantly, but the false positives is a key barrier of the counting Bloom Filter. Moreover, Bloom Filter is deployed in diverse applications,

and therefore, it has several variants. The key objectives of a Bloom Filter are to reduce the probabilities of false negatives and false positives. In addition, researchers have already proposed numerous variants of a Bloom Filter to deal with scalability and accuracy. Besides, Bloom Filter meets copious applications, and thus, researchers have performed extensive experiments to develop various Bloom Filters and to adapt them in several applications. Diverse application requirements vary, and thus, tuning of the Bloom Filter parameters is required. We have classified Bloom Filters based on features. Also, we have discussed the architecture of Bloom Filters and their various operations. Moreover, we have presented key objectives of Bloom Filters and an analysis of the false positive probability.

Appendix 2.A Source code of Bloom Filter

The codes for a simple Bloom Filter in C programming language are given below. You may follow the link for Murmur hash function https://sites.google.com/site/murmurhash/ and for bit array https://www.codeproject.com/Tips/53433/BitArray-in-C

```
1000 #include<stdio.h>
     #include"murmur.h"
1002 #include"bitarray.h" //programmer must define bitmap in this header file.

1004 //long S1=PrimeNumber1, S2=PrimeNumber2, S3=PrimeNumber3;

1006 void insert(bitmap B[], char key[], long S1, long S2, long S3)
     {
1008     int m=sizeof(B); // m is a prime number.
         int length=strlen(key);
1010     int v1=Murmur(key, length, S1) % m;
         int v2=Murmur(key, length, S2) % m;
1012     int v3=Murmur(key, length, S3) % m;
         B[v1]=1;
1014     B[v2]=1;
         B[v3]=1;
1016 }
```

Code 2.1: Insertion operation of a key into Bloom Filter.

```
1000 #include<stdio.h>
     #include"murmur.h"
1002 #include"binaryarray.h"

1004 //long S1=PrimeNumber1, S2=PrimeNumber2, S3=PrimeNumber3;

1006 void Lookup(bitmap B[], char key[], long S1, long S2, long S3)
     {
1008     int m=sizeof(B); // m is a prime number.
         int length=strlen(key);
1010     int v1=Murmur(key, length, S1) % m;
         int v2=Murmur(key, length, S2) % m;
1012     int v3=Murmur(key, length, S3) % m;
         if(B[v1] && B[v2] && B[v3])
1014         printf("The %s is a member of the Bloom Filter");
         else
1016         printf("The %s is not a member of the Bloom Filter");
     }
```

Code 2.2: Lookup operation of a key in Bloom Filter.

```
1000 #include<stdio.h>
     #include"murmur.h"
1002 #include"binaryarray.h"

1004 //long S1=PrimeNumber1, S2=PrimeNumber2, S3=PrimeNumber3;

1006 void Delete(bitmap B[], char key[], long S1, long S2, long S3)
     {
1008     int m=sizeof(B); // m is a prime number.
         int length=strlen(key);
1010     int v1=Murmur(key, length, S1) % m;
         int v2=Murmur(key, length, S2) % m;
1012     int v3=Murmur(key, length, S3) % m;
         if(B[v1] && B[v2] && B[v3])
1014     {
             B[v1]=0;
1016         B[v2]=0;
             B[v3]=0;
1018     }
         else
1020         printf("The %s is not a member of the Bloom Filter");
     }
```

Code 2.3: Deletion operation of a key from Bloom Filter.

Appendix 2.B Symbols used in the chapter

Keywords	Definition
K	Number of hash functions
κ	Input item
μ	Size of Bloom Filter which is a prime number
η	Number of input items
$\mathbb{B}[]$	Bloom Filter array
$\mathcal{H}()$	Hash function
ρ	Bit position in a cell of unsigned long int array
%	Modulus operator
$\mathbb{B}_\mu$	Bloom Filter array of size μ
$Flag$	Boolean store which is either $True$ or $False$

References

[1] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.
[2] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2) (2008) 4:1–4:26, https://doi.org/10.1145/1365815.1365816.
[3] R. Patgiri, S. Nayak, S.K. Borgohain, Preventing DDoS using Bloom filter: a survey, EAI Endorsed Trans. Scalable Inf. Syst. 5 (19) (2018), https://doi.org/10.4108/eai.19-6-2018.155865.
[4] A. Appleby, Murmur hash, https://sites.google.com/site/murmurhash/. (Accessed March 2020).
[5] F. Grandi, On the analysis of Bloom filters, Inf. Process. Lett. 129 (Supplement C) (2018) 35–39, https://doi.org/10.1016/j.ipl.2017.09.004.
[6] R. Patgiri, S. Nayak, S.K. Borgohain, rDBF: an r-dimensional Bloom filter for massive scale membership query, J. Netw. Comput. Appl. 136 (2019) 100–113, https://doi.org/10.1016/j.jnca.2019.03.004.
[7] R. Patgiri, S. Nayak, S.K. Borgohain, Role of Bloom filter in big data research: a survey, Int. J. Adv. Comput. Sci. Appl. 9 (11) (2018), https://doi.org/10.14569/IJACSA.2018.091193.
[8] R. Patgiri, S. Nayak, S.K. Borgohain, Passdb: a password database using 3D Bloom filter, in: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2019, pp. 1147–1154.
[9] R. Patgiri, Hfil: a high accuracy Bloom filter, in: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2019, pp. 2169–2174.

[10] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area web cache sharing protocol, IEEE/ACM Trans. Netw. 8 (3) (2000) 281–293, https://doi.org/10.1109/90.851975.

[11] R. Patgiri, S. Nayak, S.K. Borgohain, scalebf: a high scalable membership filter using 3D Bloom filter, Int. J. Adv. Comput. Sci. Appl. 9 (12) (2018), https://doi.org/10.14569/IJACSA.2018.091277.

[12] B. Fan, D.G. Andersen, M. Kaminsky, M.D. Mitzenmacher, Cuckoo filter: practically better than Bloom, in: Proceedings of the 10th ACM Intl. Conf. on Emerging Networking Experiments and Technologies, CoNEXT '14, 2014, pp. 75–88.

[13] S. Tarkoma, C.E. Rothenberg, E. Lagerspetz, Theory and practice of Bloom filters for distributed systems, IEEE Commun. Surv. Tutor. 14 (1) (2012) 131–155, https://doi.org/10.1109/SURV.2011.031611.00024.

CHAPTER

3

robustBF: a high accuracy and memory efficient 2D Bloom Filter for diverse applications

3.1 Introduction

Bloom Filter [1] is a membership filter that is capable of filtering a large number of data items with a tiny amount of memory. Therefore, Bloom Filter is suitable for diverse applications and applied in various domains, namely, Computer Networking [2,3], Network Security [4] and Privacy [5], IoT [6], Big Data [7], Cloud Computing [8], Biometrics [9], and Bioinformatics [10]. There are different variants of Bloom Filter available [11–13]; however, these filters are unable to provide high accuracy using tiny memory footprint without compromising query performance. Moreover, the false positive probability is an issue for Bloom Filter, and it cannot be reduced without increasing the memory size. There are many fast filters available, for instance, Morton Filter [14] and XOR Filter [15]. Morton Filter is faster than Cuckoo Filter. Morton Filter extends Cuckoo Filter and implements a compressed Cuckoo Filter, but sacrifices the memory footprint.

A multi-threaded filter is useful in deduplicating the extensive data where multiple threads can work on multiple dedicated processors. It is also suitable in an application where the filter becomes a complete system to deal with the problem. However, most of the time Bloom Filter is used to enhance system's performance, and it is not an independent system. Therefore, multiple threads may not be the right choice for many applications, for instance, security, privacy, routing, database, etc., where Bloom Filter acts just as an enhancer. Bloom Filter supports the insert and query operations. Counting Bloom Filter (CBF) supports the delete operations [16,17]. Most Bloom Filters do not support the delete operation because it introduces a false negative issue. To get away from false negative issues, conventional Bloom Filter avoids deletion operations. CBF provides the delete operation without any false negative issue. On the contrary, CBF has an issue of the counter overflow. Moreover, CBF has a higher false positive probability and larger memory footprint than conventional Bloom Filter.

In this chapter, we propose a novel Bloom Filter, called robustBF, to reduce both the false positive probability and the memory footprint. robustBF is a 2D Bloom Filter which modifies the Murmur hash function for better performance. The outcomes of the proposed system are outlined as follows:

- robustBF is a fast Bloom Filter, and it is faster than standard Bloom Filter (SBF) [18] and counting Bloom Filter (CBF) [16]. robustBF is 2.038× and 2.48× faster in insertion of 10M data than SBF and CBF, respectively.
- robustBF consumes on average 10.40× and 44.01× less memory than SBF and CBF, respectively.
- robustBF exhibits a false positive probability of almost zero in the desired false positive probability setting of 0.001 with lower memory consumption. Thus, the accuracy of robustBF is almost 100%.

We compare robustBF with SBF and CBF. We observe that robustBF exhibits lower false positive probability and has a smaller memory footprint than the other filter's variants. To the best of our knowledge, robustBF is the only Bloom Filter that can increase its accuracy and lower the memory footprint without compromising the filter's performance. Moreover, we also compare robustBF with Cuckoo Filter (CF). Experimental results show that CF is faster than robustBF in the insertion and lookup operations. But CF consumes huge memory compared to robustBF. Moreover, it exhibits the worst false positive probability in the disjoint set and random set lookup process.

This chapter is organized as follows: Section 3.2 briefly establishes preliminary concepts about Bloom Filter. Section 3.3 illustrates the proposed algorithm. Section 3.4 demonstrates the experimental results and compares the state-of-the-art algorithms. Section 3.5 analyzes the proposed algorithm. Section 3.6 concludes the chapter.

 https://doi.org/10.1016/B978-0-12-823520-1.00008-0

Copyright © 2023 Elsevier Inc. All rights reserved.

3.2 Preliminaries

Bloom Filter was introduced by Burton Howard Bloom in 1970 [1]. Bloom Filter has the potential to improve many systems. Bloom Filter is not a complete system, and it is just an enhancer of a system. Therefore, it is applied in many domains, including Networking, Security, Big Data, Bioinformatics, etc., to enhance system's performance. The capability of Bloom Filter is limited to $True$ or $False$. Let $\mathbb{B}$ be the Bloom Filter of m bits. Let $\mathcal{S}$ be the successfully inserted set into the Bloom Filter $\mathbb{B}$ to define true positives, false positives, false negatives, and true negatives. Importantly, we do not verify any item of the set $\mathcal{S}$ in real implementation. Let U be the universe where $\mathcal{S} \subset$ U and n be the total number of keys inserted into $\mathbb{B}$ using κ independent hash functions. Let x be a random query, then we can define true positives, false positives, false negatives, and true negatives as shown in Definitions 3.1, 3.2, 3.3, and 3.4, respectively.

Definition 3.1. *If $x \in \mathbb{B}$ and $x \in \mathcal{S}$, then the result of Bloom Filter is called a true positive.*

Definition 3.2. *If $x \in \mathbb{B}$ and $x \notin \mathcal{S}$, then the result of Bloom Filter is called a false positive.*

Definition 3.3. *If $x \notin \mathbb{B}$ and $x \in \mathcal{S}$, then the result of Bloom Filter is called a false negative.*

Definition 3.4. *If $x \notin \mathbb{B}$ and $x \notin \mathcal{S}$, then the result of Bloom Filter is called a true negative.*

The queried item x can either belong to $\mathcal{S}$ or not. For instance, it can be either $x \in \mathcal{S}$ or $x \notin \mathcal{S}$. We denote the originally inserted set as $\mathcal{S}$. Therefore, we can find whether $\mathcal{S}$ contains the queried item x or not, but we do not query any item from $\mathcal{S}$ in real implementation. For convenience, suppose x is queried in $\mathcal{S}$ to define true positives, false positives, false negatives, and true negatives. Notably, the conventional Bloom Filter does not have false negative issues.

3.2.1 Operations

There are two key operations of Bloom Filter, namely, the insert and lookup (query) operations. Algorithm 3.1 presents a key to Bloom Filter's insertion process using three hash functions. The same process is applied in the lookup of a key, which is shown in Algorithm 3.2. Filter $\mathbb{B}$ and x are the conventional Bloom Filter and input item, respectively. Then η_1, η_2, and η_3 are the three seed values. Also m is the length of the conventional Bloom Filter array.

Algorithm 3.1 Insertion of an input item x into Bloom Filter $\mathbb{B}$ using three hash functions.

```
1: procedure Insert(𝔹, x, η1, η2, η3)
2:     i1 = Murmur(x, length, η1) % m
3:     i2 = Murmur(x, length, η2) % m
4:     i3 = Murmur(x, length, η3) % m
5:     𝔹[i1] ← 1
6:     𝔹[i2] ← 1
7:     𝔹[i3] ← 1
8: end procedure
```

Algorithm 3.2 Lookup an item x in Bloom Filter using three hash functions.

```
1: procedure Lookup(𝔹[], x, η1, η2, η3)
2:     i1 = Murmur(x, length, η1) % m
3:     i2 = Murmur(x, length, η2) % m
4:     i3 = Murmur(x, length, η3) % m
5:     if 𝔹[i1] AND 𝔹[i2] AND 𝔹[i3] then
6:         return True
7:     else
8:         return False
9:     end if
10: end procedure
```

3.3 robustBF – the proposed system

Definition 3.5. *Consider a set $\mathcal{S} = \{x_1, x_2, x_3, \ldots, x_n\}$ and a multidimensional bit array $\mathbb{B}_{d_1,d_2,d_3,\ldots,d_i}$ where n is the total number of elements in the set $\mathcal{S}$ and d_i is the dimension. A multidimensional Bloom Filter $\mathbb{B}$ maps the elements of set $\mathcal{S}$ into the multidimensional bit array $\mathbb{B}_{d_1,d_2,d_3,\ldots,d_i}$ using κ distinct hash functions similar to the conventional Bloom Filter.*

We propose a novel Bloom Filter based on a 2D array as defined in Definition 3.5, called robustBF. We modify the existing hash function and embed the modified hash function to construct a new Bloom Filter. robustBF modifies the hashing technique to increase its performance. The performance of robustBF depends on the Murmur hash functions, which introduces some biases and a number of hash function calls. This bias helps in improving the accuracy, and hence, reduces the false positive probability. robustBF is designed so as to provide a high accuracy and memory-efficient Bloom Filter. The detailed architecture of the 2D Bloom Filter is depicted in Fig. 3.1. Each cell of 2D Bloom Filter of Fig. 3.1 contains β bits; for instance, 64 bits.

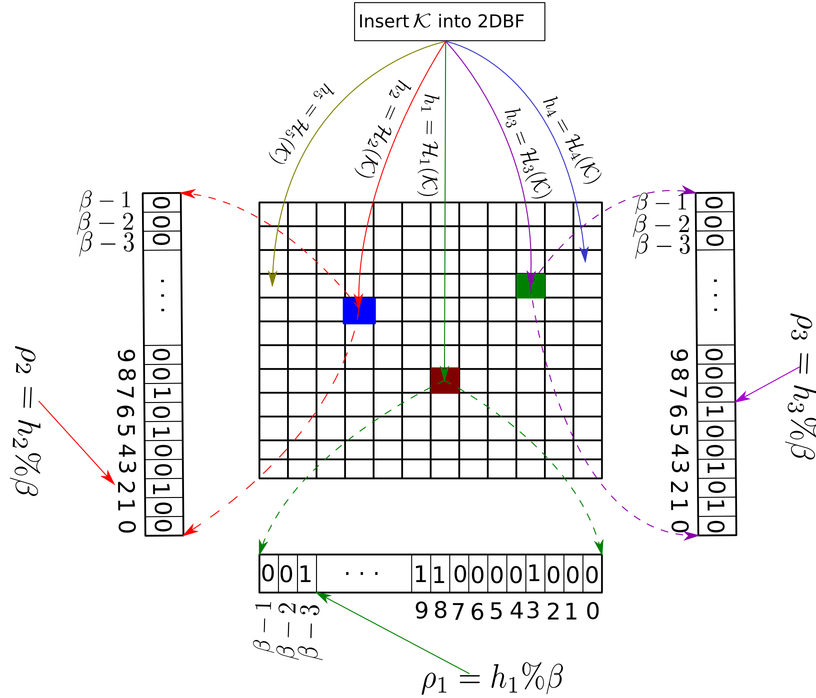


FIGURE 3.1 Insertion and lookup process in 2D Bloom Filter.

3.3.1 Selection of a hash function

The performance of the Bloom Filter depends on the hash functions. Many hash functions are available, namely, xxHash, SuperFastHash, FastHash, FNV, and Murmur hash functions. Murmur hash function is the finest hash function among all non-cryptographic string hash functions [19]. There are also cryptographic string hash functions, namely, SHA1, SHA2, and MD5. These string hash functions are the best for security purposes. However, these cryptographic string hash functions are unable to reduce the false positive probability [20]. Moreover, such functions degrade the performance of the Bloom Filter, which is proven in Patgiri et al. [20]. Thus, we modified the Murmur hash functions and evaluated the best modification among the modified hash functions. A Murmur hash function depends on the bit mixture and scanning of the input, and thus, we altered the Murmur hash function and termed it as H1, H2, H3, H4, H5, H6, and H7. The modified Murmur hash function H1 reads 3 bytes of data at a time. Similarly, the modified Murmur hash function H2 reads 4 bytes at a time. Thus, H3, H4, H5, H6, H7, H8, and H9 read 5, 6, 7, 8, 9, 10, and 11 bytes of data at a time, respectively. This shows a huge difference in changing the reading of data at a time in Murmur hash functions. The experimental results show that H4 exhibits the best performance.

Algorithm 3.3 Insertion operation into 2DBF. The symbols %, |, and $\ll$ are the modulus, bitwise-OR, and bitwise shift-left operators, respectively.

```
1: procedure SET(𝔹_{X,Y}, h)
2:     i = h%X
3:     j = h%Y
4:     ρ = h%β
5:     𝔹_{i,j} = 𝔹_{i,j} | (1 ≪ ρ)
6: end procedure
```

Algorithm 3.4 Insertion of an item x using κ distinct hash functions.

```
1: procedure INSERTANITEM(𝔹_{X,Y}, x)
2:     l = LENGTH(x)
3:     Init seed value η
4:     for i = 1 to κ do
5:         h_i = MURMUR2(x, l, η)
6:         η = h_i
7:         SET(𝔹_{X,Y}, h_i)
8:     end for
9: end procedure
```

3.3.2 Insertion

Initially, an item is hashed into 2DBF using κ distinct hash functions as shown in Algorithm 3.4. The item is mapped into κ cells of a 2D array, and the bits are set to 1 using Algorithm 3.3. Let $\mathbb{B}_{X,Y}$ be the 2D Bloom Filter where X and Y are the dimensions, and these are prime numbers. Let $\mathcal{H}()$ be the modified Murmur hash function, and x is an item to be inserted. Let $\mathbb{B}_{i,j}$ be a particular cell in $\mathbb{B}_{X,Y}$ filter. To insert, Eq. (3.1) is invoked,

$$\mathbb{B}_{i,j} \leftarrow \mathbb{B}_{i,j} \mid \mathbb{P}, \tag{3.1}$$

where $h = \mathcal{H}(x)$, $i = h\%X$, $j = h\%Y$, $\rho = h\%\beta$, and $\mathbb{P} = (1 \ll \rho)$, where %, |, β and $\ll$ are the modulus operator, bitwise-OR operator, size of a cell, and bitwise left shift operator, respectively. The number β should also be a prime. Let us assume that **unsigned long int** occupies 64 bits, then the nearest prime number is 61. As we know, that prime number exhibits a good distribution property in hashing. Therefore, β should be 61, but it cannot be greater than the size of a cell in the Bloom Filter.

Algorithm 3.5 Lookup operation of input item x in 2DBF. The symbol %, |, &, and $\gg$ are the modulus, bitwise-AND, bitwise shift-left, and bitwise shift-right operators, respectively.

```
1: procedure TEST(𝔹_{X,Y}, h)
2:     i = h%X
3:     j = h%Y
4:     ρ = h%β
5:     return ((𝔹_{i,j} & (1 ≪ ρ)) ≫ ρ)
6: end procedure
```

3.3.3 Lookup

Algorithm 3.5 extracts the bit information from a cell of a 2D array and returns the bit information to calling function (Algorithm 3.6). Algorithm 3.6 declares whether the item x is a member of robustBF or not. For the lookup operation (query operation) of an item x, Eq. (3.2) is invoked,

$$F = (\mathbb{B}_{i,j} \,\&\, \mathbb{P}) \gg \rho. \tag{3.2}$$

Algorithm 3.6 Lookup an item x into $\mathbb{B}$ using κ distinct hash functions.

1: **procedure** LOOKUP($\mathbb{B}_{X,Y},\ x$)
2: $l = $ STRINGLENGTH(x)
3: $\mathcal{F} = true$
4: **for** $i = 1\ to\ \kappa$ **do**
5: $h_i = $ MURMUR2($x,\ l,\ \eta$)
6: $\eta = h_i$
7: $\mathcal{F} = \mathcal{F}$ & TEST($\mathbb{B}_{X,Y},\ h_i$)
8: **end for**
9: **return** $\mathcal{F}$
10: **end procedure**

If $\mathcal{F}$ is 1, then the key is a member of 2DBF, and otherwise it is not a member of 2DBF. Eqs. (3.1) and (3.2) depend on the hash function $\mathcal{H}()$.

3.4 Experimental results

We evaluate the accuracy, false positive probability, query and insertion performance, and memory consumption of robustBF. We have conducted our experiment on a desktop PC with the configuration of Intel Core i7-7700 CPU @ 3.60 GHz × 8, 8 GB RAM, 1 TB HDD, Ubuntu 18.04.5 LTS, and GCC-7.5.0 which are shown in Table 3.1.

TABLE 3.1 Configuration details of experimental environment.

Name	Description
CPU	Intel Core i7-7700 CPU @ 3.60 GHz × 8
RAM	8 GB
OS	Ubuntu 18.04.5 LTS
HDD	1TB
Language	GCC-7.5.0

3.4.1 Test cases

We have created four different test cases to validate the accuracy, performance, memory efficiency, and false positive rate. The four test cases are Same Set, Mixed Set, Disjoint Set, and Random Set which are defined in Definitions 3.6, 3.7, 3.8, and 3.9, respectively [20]. These test cases are used to evaluate the strengths and weaknesses of Bloom Filter in every aspect. Let $\mathcal{S} = \{s_1, s_2, s_3, \ldots, s_m\}$ be the input set for the robustBF.

Definition 3.6. *Let $\mathcal{Q}$ be a set queried where $\mathcal{Q} = \mathcal{S}$, then the set $\mathcal{Q}$ is called Same Set.*

Definition 3.7. *Let $\mathcal{Q} = \{Q^1, Q^2\}$ be a query set where $Q^1 = \{Q^1_1,\ Q^1_2, Q^1_3, \ldots\}$ and $Q^1 = \{Q^2_1,\ Q^2_2, Q^2_3, \ldots\}$ such that $Q^1 \subset \mathcal{S}$ and $Q^2 \cap \mathcal{S} = \emptyset$, then the set $\mathcal{Q}$ is called a Mixed Set.*

Definition 3.8. *Let $\mathcal{Q}$ be a query set where $\mathcal{Q} \cap \mathcal{S} = \emptyset$, then the set $\mathcal{Q}$ is called a Random Set.*

Definition 3.9. *Let $\mathcal{Q}$ be a randomly generated query set, then the set $\mathcal{Q}$ is called a Disjoint Set.*

3.4.2 Experiments

This section presents the accuracy of Bloom Filters and its memory requirements. We compare robustBF with standard Bloom Filter (SBF) [18] where SBF is a standard Bloom Filter to benchmark. The robustBF, SBF, and CBF are single-threaded Bloom Filters. Single-threaded Bloom Filters are useful as enhancers of a system, while a multi-threaded Bloom Filter is a complete system. robustBF outperforms SBF and CBF in every aspect, presented in this section.

3.4.3 Hash function experiments

robustBF is a two-dimensional Bloom Filter which depends on the Murmur hash function. The insertion performances of H1, H2, H3, H4, H5, H6, H7, H8, and H9 are demonstrated in Fig. 3.2. A total of 10M data is inserted into robustBF. The performance of robustBF is the lowest in H1 and the highest in H9. However, the insertion performance is not as important as the lookup operation.

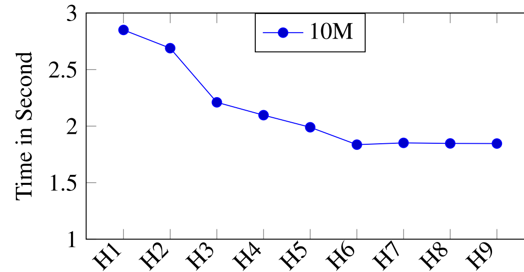


FIGURE 3.2 Insertion time of modified Murmur hash functions H1, H2, H3, H4, H5, H6, H7, H8, and H9. Lower is better.

The Bloom Filter is designed to provide higher lookup performance with a tiny amount of memory usage. robustBF is evaluated based on its performance on Same Set, Mixed Set, Disjoint Set, and Random Set to reveal the strength and weakness. robustBF is evaluated with modified Murmur hash functions H1, H2, H3, H4, H5, H6, H7, H8, and H9. robustBF exhibits a similar lookup performance of 10M queries using H3, H4, H5, H6, H7, H8, and H9. Therefore, we select a modified Murmur hash function from H3, H4, H5, H6, H7, H8, and H9. However, it is challenging to select the best performer because their performance differs in different test cases. See Fig. 3.3.

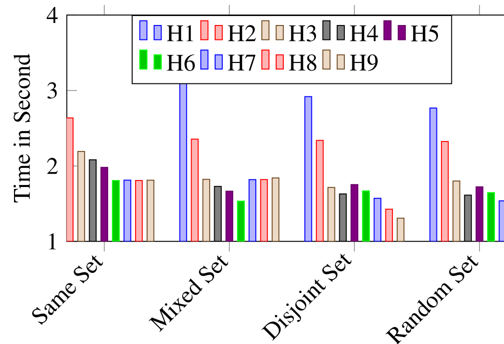


FIGURE 3.3 Lookup time of modified Murmur hash functions H1, H2, H3, H4, H5, H6, H7, H8, and H9. Lower is better.

robustBF measures its accuracy as $\mathcal{A} = 1 - \mathcal{FPP}$ since there are no false negatives. Fig. 3.4 demonstrates the accuracy of robustBF using modified Murmur hash functions H1, H2, H3, H4, H5, H6, H7, H8, and H9. As per our observation, H4 has the highest accuracy and lowest false positive probability in all test cases. The false positive probability is demonstrated in Fig. 3.5. The H6, H7, H8, and H9 functions exhibit the highest performance; however, the false positive probability of these hash functions is poor. Therefore, we are simply strike out the modified Murmur hash function H6, H7, H8, and H9. The remaining hash functions are H3, H4, and H5 to compare their performance.

robustBF performance is measured using H3, H4, and H5 by inserting 10M, 20M, 30M, 40M, and 50M of data items. robustBF performance is similar for H4 and H5; however, its performance is lower using H3 as depicted in Fig. 3.6. We observed that H4 is the best performer among the modified Murmur hash functions. However, H4 is not the best performer in insertion and lookup, but it is best with respect to false positive probability. Therefore, we conclude that H4 is the best choice for robustBF. The rest of the experiment of robustBF is performed using the H4 function because H4 exhibits its best performance regarding the false positive probability, which is crucial for Bloom Filter.

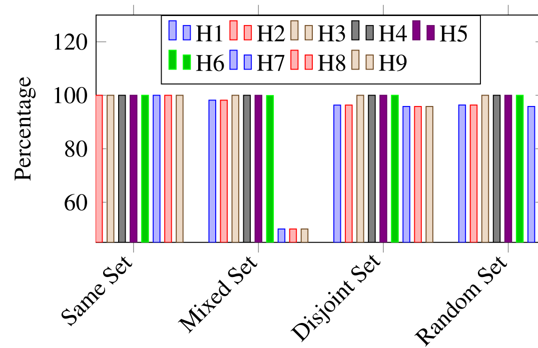


FIGURE 3.4 Accuracy of modified murmur hash functions H1, H2, H3, H4, H5, H6, H7, H8, and H9. Higher is better.

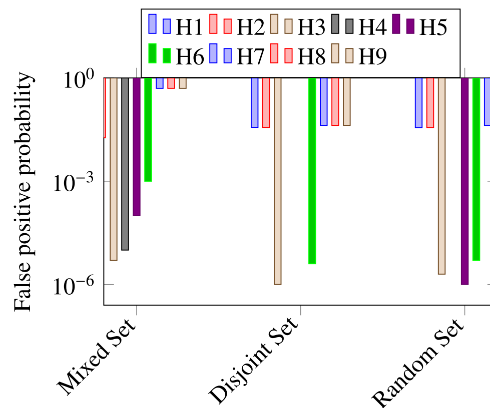


FIGURE 3.5 False positive probability of lookup of modified Murmur hash functions H1, H2, H3, H4, H5, H6, H7, H8, and H9. Lower is better.

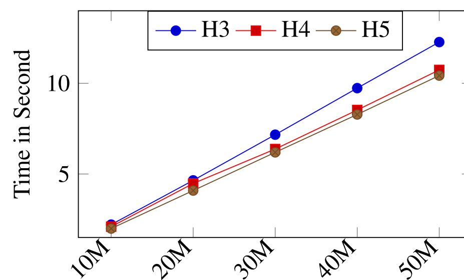


FIGURE 3.6 Insertion time of robustBF of 10M, 20M, 30M, 40M, and 50M by H3, H4, and H5 Murmur hash functions. Lower is better.

3.4.4 Comparison of robustBF with other filters

The comparison of insertion time of robustBF and SBF is demonstrated in Fig. 3.7. robustBF, SBF, and CBF perform 5.426 million operations per second (MOPS), 2.182 MOPS, and 1.79 MOPS on average, respectively. robustBF is faster than SBF in insertion.

Figs. 3.8 and 3.9 demonstrate the lookup time taken by robustBF and SBF in different test cases, namely, Same Set (SS), Mixed Set (MS), Disjoint Set (DS), and Random Set (RS). robustBF is faster than SBF and CBF in every aspect. robustBF performs 5.463 MOPS, 6.373 MOPS, 6.369 MOPS, and 6.423 MOPS on the Same Set, Mixed Set, Disjoint Set, and Random Set, respectively. On the other hand, SBF performs 2.554 MOPS, 3.178 MOPS, 3.667 MOPS, and 3.727 MOPS on the Same Set, Mixed Set, Disjoint Set, and Random Set on average, respectively. Similarly, CBF performs 1.994 MOPS, 2.642 MOPS, 3.347 MOPS, and 3.384 MOPS on the Same Set, Mixed Set, Disjoint Set, and Random Set on average, respectively.

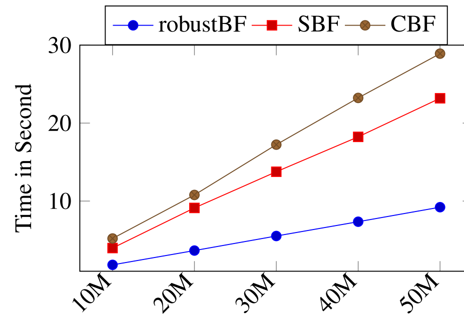


FIGURE 3.7 Insertion time of robustBF, SBF, and CBF. Lower is better.

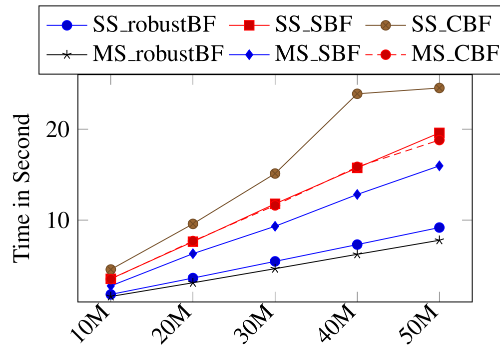


FIGURE 3.8 Lookup time of robustBF, SBF, and CBF in Same Set (SS) and Mixed Set (MS). Lower is better.

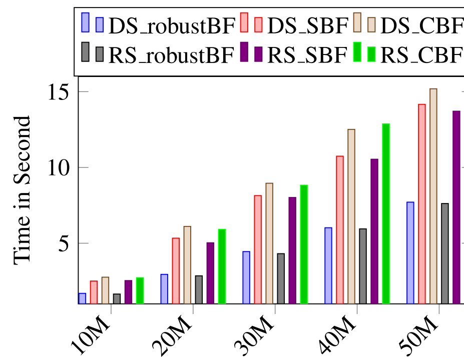


FIGURE 3.9 Lookup time of robustBF, SBF, and CBF in Disjoint Set (DS) and Random Set (RS). Lower is better.

TABLE 3.2 False positive probability of robustBF and SBF in various test cases. The symbol 0* denotes 0.000001. Lower is better.

Dataset	MS_robustBF	MS_SBF	MS_CBF	DS_robustBF	DS_SBF	DS_CBF	RS_robustBF	RS_SBF	RS_CBF
10M	0.000010	0.001028	0.001027	0	0.001012	0.000996	0*	0.000989	0.000998
20M	0.000010	0.001009	0.000996	0*	0.001002	0.000999	0	0.00098	0.001002
30M	0.000007	0.000998	0.000999	0	0.001003	0.000996	0	0.000981	0.000976
40M	0.000003	0.001019	0.001001	0	0.001022	0.001003	0	0.001	0.000977
50M	0.000002	0.001013	0.000994	0	0.001008	0.000999	0	0.000976	0.000986

Table 3.2 shows the false positive probability of robustBF and SBF in different test cases with 10 to 100 million of datasets. robustBF and SBF do not exhibit any false positive probability on the Same Set. However, there is a false positive probability in other test cases. robustBF exhibits the highest false positive probability on the Mixed Set,

which is much lower than that of SBF. It is an almost zero false positive probability. However, the configuration of SBF and CBF is 0.001, and hence, this shows a constantly similar false positive probability.

TABLE 3.3 Accuracy of robustBF, SBF, and CBF on Same Set (SS), Mixed Set (MS), Disjoint Set (DS), and Random Set (RS). The symbol 99* denotes 99.9999. Higher is better.

Interval	MS_robustBF	MS_SBF	MS_CBF	DS_robustBF	DS_SBF	DS_CBF	RS_robustBF	RS_SBF	RS_CBF
10M	99.999	99.8972	99.8973	100	99.8988	99.9004	99*	99.9011	99.9002
20M	99.999	99.8991	99.9004	99*	99.8998	99.9001	100	99.902	99.8998
30M	99.9993	99.9002	99.9001	100	99.8997	99.9004	100	99.9019	99.9024
40M	99.9997	99.8981	99.8999	100	99.8978	99.8997	100	99.9	99.9023
50M	99.9998	99.8987	99.9006	100	99.8992	99.9001	100	99.9024	99.9014

Table 3.3 demonstrates the accuracy of robustBF and SBF. robustBF, SBF, and CBF all exhibit 100% accuracy on the Same Set. The accuracy of robustBF is much higher than that of SBF and CBF. The lowest accuracy of robustBF is 99.999% in 20M and 10M datasets with the Mixed Set test cases.

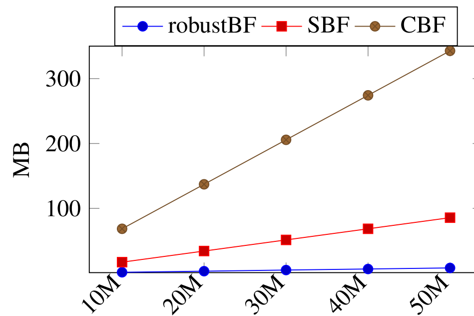


FIGURE 3.10 Memory consumption of robustBF, SBF, and CBF. Lower is better.

Interestingly, robustBF uses a tiny amount of memory. SBF uses more memory than robustBF, as shown in Fig. 3.10. robustBF, SBF, and CBF require 1.382, 14.378, and 57.511 bits of memory per element, respectively. Therefore, robustBF consumes 10.40× and 44.01× less memory than SBF and CBF, respectively. Therefore, robustBF is better than SBF and CBF in memory consumption.

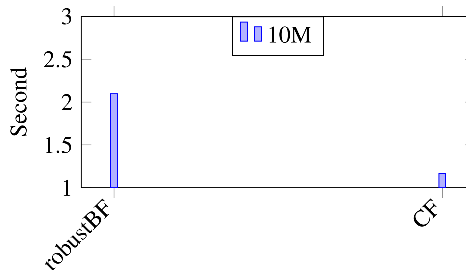


FIGURE 3.11 Comparison of insertion time of 10M data into robustBF and CF in seconds. Lower is better.

Now, let us compare robustBF with Cuckoo Filter [21] which is not a Bloom Filter. Cuckoo Filter is a filter based on cuckoo hashing. Fig. 3.11 demonstrates the insertion time taken by robustBF and CF, which is the fastest filter in the insertion of 10M data into the filters. Here, robustBF takes the most time, which is much slower than CF.

Similar to insertion, robustBF is slower in lookup than CF, as shown in Fig. 3.12. The CF is the fastest filter to perform the lookup of 10M data.

The false positive probability of robustBF is almost zero, which is demonstrated in Table 3.4. But CF exhibits good false positive probability on the Same Set and Mixed Set, whereas it shows the worst performance on Disjoint Set and Random Set due to the kicking process.

Fig. 3.13 demonstrates the memory consumption in 10M data case by robustBF and CF. robustBF and CF occupy 1.55 and 24 MB for 10M data, respectively.

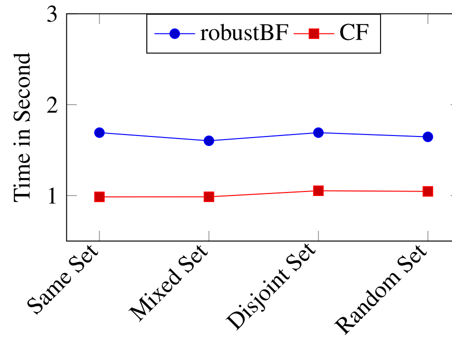


FIGURE 3.12 Comparison of lookup time of 10M in robustBF and CF in seconds. Lower is better.

TABLE 3.4 Comparison of false positive probability in lookup of 10M in robustBF and CF. Lower is better.

Test cases	robustBF	CF
Same Set	0	0
Mixed Set	1E-05	0.0005592
Disjoint Set	0	0.5749
Random Set	0	0.995351

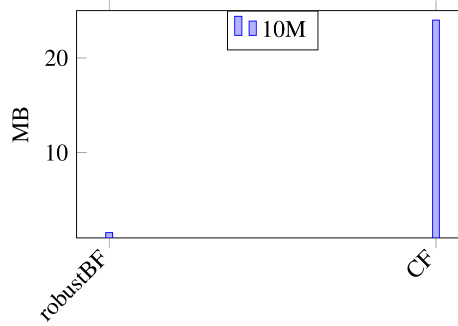


FIGURE 3.13 Comparison of memory consumption on 10M data by robustBF and CF. Lower is better.

3.5 Analysis

There are different variants of Bloom Filter available [11], for instance, learned Bloom Filter [22,23]. Learned Bloom Filters are designed to reduce the false positive probability using Machine Learning techniques, which are costly in terms of computation and space complexity. These are designed for different purposes. There are also faster filters than Bloom Filters; namely, counting-Quotient Filter [24], Morton Filter [14], and XOR filter [15]; however, these filters do not provide high accuracy using a low memory footprint similar to our robustBF.

Let m be the number of bits of the Bloom Filter $\mathbb{B}$ and the probability of a particular bit to be not set to 1 is $(1 - \frac{1}{m})$. If there are κ hash functions and n total items to be inserted into the Bloom Filter $\mathbb{B}$, then the probability of a bit not set to 1 is $\left(1 - \frac{1}{m}\right)^{n\kappa}$. As we know,

$$\lim_{m \to \infty} \left(1 - \frac{1}{m}\right)^m = e^{-1}. \tag{3.3}$$

Using Eq. (3.3), we get

$$\left(1 - \frac{1}{m}\right)^{n\kappa} \approx e^{-\kappa n/m}. \tag{3.4}$$

Therefore, the probability that a particular bit is set to 1 is

$$1-\left(1-\frac{1}{m}\right)^{n\kappa}. \tag{3.5}$$

Thus, the desired false positive probability is

$$\varepsilon=\left(1-\left(1-\frac{1}{m}\right)^{n\kappa}\right)\approx\left(1-e^{-\kappa n/m}\right)^{\kappa}. \tag{3.6}$$

The number of hash functions κ cannot be too large or too small. Bloom Filter requires an optimal number of hash functions. The assumption of the optimal number of hash function is $\kappa=\frac{m}{n}\ln 2$. The optimal number of hash functions is an assumption and it works well with conventional Bloom Filter. It gives the optimal number of hash functions. Solving Eq. (3.6) and using the assumption on the number of hash functions, we get

$$\varepsilon=\left(1-e^{-(\frac{m}{n}\ln 2)n/m}\right)^{\frac{m}{n}\ln 2}. \tag{3.7}$$

By simplifying Eq. (3.7), we get

$$\ln\varepsilon=-\frac{m}{n}(\ln 2)^2 \implies m=-\frac{n\ln\varepsilon}{(\ln 2)^2}. \tag{3.8}$$

The memory requirement is that m depends on the desired false positive probability ε and number of input items n. robustBF uses approximately half the desired memory calculated by Eq. (3.8).

3.5.1 Memory requirements of robustBF

robustBF is a 2D Bloom Filter (2DBF), and thus, the memory requirement not only depends on n and ε but also on the dimension of the array. Let X and Y be the dimensions of robustBF. The dimensions $X\neq Y$ and these are prime numbers. If X and Y are not prime numbers, then the number of false positives increases. We need to calculate X and Y from m. Let P_i be the array of prime numbers and $q=\frac{m}{2\beta}$ where the β is the number of bits per cell in robustBF. Now, we calculate $t=\sqrt{q}$ and call the function $i=$ SELECTPRIME(t) as shown in Algorithm 3.7.

Algorithm 3.7 Index calculation of prime number array P_i.

```
1: procedure SELECTPRIME(t)
2:     for i = 1 to TotalNumberOfPrime do
3:         if P_i > t then return i;
4:         end if
5:     end for
6: end procedure
```

The dimensions are $X=P_{\frac{i}{2}+3}$ and $Y=P_{\frac{i}{2}-3}$. Thus, robustBF allocates an **unsigned long int** 2D array, i.e., the total number of memory bit is $X\times Y\times\beta$ where β is the bit size of **unsigned long int**. Also, robustBF uses less than a half of the number of optimal hash functions calculated because it performs the modulus operation using dimensions X and Y to place an item into the filter. Therefore, it occupies the lowest memory as compared to the state-of-the-art filters, to the best of our knowledge.

3.5.2 Comparison of robustBF with other filters

There are numerous faster membership filters available, other than robustBF. For instance, counting Quotient Filter (CBF) [24], Morton Filter (MF) [14], XOR Filter (XF) [15]. Morton and XOR filters are the fastest filters. However, there is a trade-off among speed, memory, and false positive probability. Also, there is a compressed Bloom Filter, which compromises performance and accuracy with memory [27]. Various membership filters are available, and a few membership filters are compared in Table 3.5. Many membership filters do not provide high accuracy and low

false positive probability with a tiny amount of memory, while robustBF can fulfill the high accuracy requirement with a low memory footprint. CBF has the highest false positive probability and consumes the largest memory [16].

TABLE 3.5 Comparison of a few membership filters. Rating scale from 1 to 10. Rating 1 is the worst, and 10 is the best.

Filter	False positives	Accuracy	Memory Consumption	Query Speed
SBF [18]	7	7	6	6
CBF [16]	4	4	3	5
Cuckoo Filter [21]	4	4	5	9
CQF [24]	6	6	7	9
Morton Filter [14]	6	6	7	10
XOR Filter [15]	6	6	7	10
VI-CBF [25]	6	6	5	9
TCBF [26]	7	7	7	9
robustBF	10	10	10	8

Typically, CBF is slow with a high memory footprint and false positive probability, but it has no false negatives. If we compare the architecture of the existing membership filters with CBF, then we get Table 3.5 which demonstrates the overall comparison among SBF, CBF, CF, CQF, MF, XF, VI-CBF, TCBF, and robustBF using rating points from 1 to 10. We have already compared SBF, CBF, CF, and robustBF. For this comparison, we can draw the performance of the filters intuitively.

Rating 1 represents the worst, and rating 10 is the best. SBF is the standard, and the overall rating is 6.5. The MF and XF are the fastest, and their ratings are 10 for both. The rating of robustBF and CF is almost the same in execution time, while robustBF outperforms in other features. Moreover, SBF, CF, MF, and XF are the same in false positive probability and accuracy. SBF, CF, MF, XF, VI-CBF, and TCBF have similar memory footprints. robustBF outperforms all other filters with respect to false positive probability, accuracy, and memory footprint, except for execution time. However, the execution performance of robustBF is highly competitive.

3.6 Conclusion

In this chapter, we presented a novel Bloom Filter, called robustBF. We have demonstrated that robustBF is able to reduce the false positive probability to almost zero, which is desired. Also, robustBF consumes approximately $10\times$ and $44\times$ less memory than SBF and CBF, respectively, with the same settings. We have also demonstrated its accuracy, which is satisfactorily high. The lookup and insertion performance of robustBF is higher than those of SBF and CBF. No Bloom Filter can increase its accuracy by lowering the memory footprint significantly, to the best of our knowledge. There are many faster filters available other than robustBF. The key objective of robustBF is to improve accuracy and lower memory footprint without compromising the filter's performance. Therefore, the robustBF performance is compared with SBF and CBF. We agree that there are many faster filters available than robustBF, for instance, Morton Filter [14] and XOR Filter [15]; but these filters use more memory per item. Therefore, robustBF can improve various applications' performance using a tiny amount of memory in Computer Networking, Security and Privacy, Blockchain, IoT, Big Data, Cloud Computing, Biometrics, and Bioinformatics.

References

[1] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Commun. ACM 13 (7) (1970) 422–426.

[2] J.H. Mun, H. Lim, New approach for efficient IP address lookup using a Bloom filter in tree-based algorithms, IEEE Trans. Comput. 65 (5) (2016) 1558–1565, https://doi.org/10.1109/TC.2015.2444850.

[3] H. Lee, A. Nakao, Improving Bloom filter forwarding architectures, IEEE Commun. Lett. 18 (10) (2014) 1715–1718, https://doi.org/10.1109/LCOMM.2014.2355199.

[4] R. Patgiri, S. Nayak, S.K. Borgohain, Preventing DDoS using Bloom filter: a survey, EAI Endorsed Trans. Scalable Inf. Syst. 5 (19) (2018) 1–9, https://doi.org/10.4108/eai.19-6-2018.155865.

[5] R. Patgiri, S. Nayak, S.K. Borgohain, Passdb: a password database with strict privacy protocol using 3D Bloom filter, Inf. Sci. 539 (2020) 157–176, https://doi.org/10.1016/j.ins.2020.05.135.

[6] A. Singh, S. Garg, S. Batra, N. Kumar, J.J. Rodrigues, Bloom filter based optimization scheme for massive data handling in IoT environment, Future Gener. Comput. Syst. 82 (2018) 440–449, https://doi.org/10.1016/j.future.2017.12.016.

[7] R. Patgiri, S. Nayak, S.K. Borgohain, Role of Bloom filter in big data research: a survey, Int. J. Adv. Comput. Sci. Appl. 9 (11) (2019) 655–661, https://doi.org/10.14569/IJACSA.2018.091193.

[8] A. Singh, S. Garg, K. Kaur, S. Batra, N. Kumar, K.R. Choo, Fuzzy-folded Bloom filter-as-a-service for big data storage in the cloud, IEEE Trans. Ind. Inform. 15 (4) (2019) 2338–2348, https://doi.org/10.1109/TII.2018.2850053.

[9] C. Rathgeb, F. Breitinger, H. Baier, C. Busch, Towards Bloom filter-based indexing of iris biometric data, in: 2015 International Conference on Biometrics (ICB), Phuket, Thailand, IEEE, 2015, pp. 422–429.

[10] S. Nayak, R. Patgiri, A review on role of Bloom filter on dna assembly, IEEE Access 7 (2019) 66939–66954, https://doi.org/10.1109/ACCESS.2019.2910180.

[11] L. Luo, D. Guo, R.T.B. Ma, O. Rottenstreich, X. Luo, Optimizing Bloom filter: challenges, solutions, and comparisons, IEEE Commun. Surv. Tutor. 21 (2) (2019) 1912–1949, https://doi.org/10.1109/COMST.2018.2889329.

[12] H. Lim, J. Lee, C. Yim, Complement Bloom filter for identifying true positiveness of a Bloom filter, IEEE Commun. Lett. 19 (11) (2015) 1905–1908, https://doi.org/10.1109/LCOMM.2015.2478462.

[13] P. Reviriego, J. Martínez, S. Pontarelli, Cfbf: reducing the insertion time of cuckoo filters with an integrated Bloom filter, IEEE Commun. Lett. 23 (10) (2019) 1857–1861, https://doi.org/10.1109/LCOMM.2019.2930508.

[14] A.D. Breslow, N.S. Jayasena, Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity, Proc. VLDB Endow. 11 (9) (2018) 1041–1055, https://doi.org/10.14778/3213880.3213884.

[15] T.M. Graf, D. Lemire, Xor filters: faster and smaller than Bloom and cuckoo filters, ACM J. Exp. Algorithmics 25 (2020), https://doi.org/10.1145/3376122.

[16] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area web cache sharing protocol, IEEE/ACM Trans. Netw. 8 (3) (2000) 281–293, https://doi.org/10.1109/90.851975.

[17] S. Nayak, R. Patgiri, countbf: a general-purpose high accuracy and space efficient counting Bloom filter, in: 2021 17th International Conference on Network and Service Management (CNSM), Izmir, Turkey, IEEE, 2021, pp. 355–359.

[18] A. Kirsch, M. Mitzenmacher, Less hashing, same performance: building a better Bloom filter, Random Struct. Algorithms 33 (2) (2008) 187–218.

[19] A. Appleby, Murmurhash, Retrieved on September 2020 from https://sites.google.com/site/murmurhash/, 2020.

[20] R. Patgiri, S. Nayak, N.B. Muppalaneni, Is Bloom filter a bad choice for security and privacy?, in: 2021 International Conference on Information Networking (ICOIN), Jeju Island, Korea (South), IEEE, 2021, pp. 648–653.

[21] B. Fan, D.G. Andersen, M. Kaminsky, M.D. Mitzenmacher, Cuckoo filter: practically better than Bloom, in: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies, CoNEXT '14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 75–88.

[22] M. Mitzenmacher, A model for learned Bloom filters and optimizing by sandwiching, in: S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), Advances in Neural Information Processing Systems, vol. 31, Curran Associates, Inc., Montréal, Canada, 2018, pp. 464–473, https://proceedings.neurips.cc/paper/2018/file/0f49c89d1e7298bb9930789c8ed59d48-Paper.pdf.

[23] Z. Dai, A. Shrivastava, Adaptive learned Bloom filter (Ada-BF): efficient utilization of the classifier with application to real-time information filtering on the web, in: Advances in Neural Information Processing Systems, Virtual, Curran Associates, Inc., 2020, pp. 464–473, https://proceedings.neurips.cc/paper/2020/hash/86b94dae7c6517ec1ac767fd2c136580-Abstract.html.

[24] P. Pandey, M.A. Bender, R. Johnson, R. Patro, A general-purpose counting filter: making every bit count, in: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD'17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 775–787.

[25] O. Rottenstreich, Y. Kanizo, I. Keslassy, The variable-increment counting Bloom filter, in: 2012 Proceedings IEEE INFOCOM, Orlando, FL, USA, IEEE, 2012, pp. 1880–1888.

[26] P. Reviriego, O. Rottenstreich, The tandem counting Bloom filter – it takes two counters to tango, IEEE/ACM Trans. Netw. 27 (6) (2019) 2252–2265, https://doi.org/10.1109/TNET.2019.2944954.

[27] M. Mitzenmacher, Compressed Bloom filters, IEEE/ACM Trans. Netw. 10 (5) (2002) 604–612, https://doi.org/10.1109/TNET.2002.803864.

This page intentionally left blank

CHAPTER

4

Impact of the hash functions in Bloom Filter design

4.1 Introduction

Bloom Filter is applied in diverse applications due to its tiny memory footprint. Moreover, different membership filter variants are available. But they consume comparatively large memory due to different design architecture. Memory footprint can be reduced by selecting or designing a good hash function. A good hash function reduces the memory footprint, along with the false positive probability. In addition, it dramatically enhances the query performance of the Bloom Filter.

There are various hash functions available, and we categorize them into two major categories, namely cryptographic and non-cryptographic hash functions. Cryptographic hash functions are more secure than non-cryptographic hash functions, which are designed with respect to adversaries. Therefore, they produce a large-size hash value. Moreover, they do not support a seed value. But cryptographic hash functions are not suitable for Bloom Filter. This chapter demonstrates why the cryptographic string hash function is unsuitable for Bloom Filter. Also, we need to select the best non-cryptographic hash function to develop a new Bloom Filter.

This chapter exposes how the hash functions impact the Bloom Filter, which is strongly dependent on the hash functions. To place an item, it requires k hash functions. The items are placed in k places in the bit array of the Bloom Filter. Alternatively, an item requires k bit positions in the bit array of the Bloom Filter. Additionally, we experimentally demonstrate the performance, false positive probability, and memory requirement of robustBF using different hash functions. Furthermore, we compare libbloom and robustBF. Moreover, we expose the major requirement for designing Bloom Filter.

This chapter is organized as follows: Section 4.2 discusses different hash functions. Section 4.3 discusses the importance of prime numbers in Bloom Filter and hash functions. Section 4.4 demonstrates the relation between the number of hash functions and the false positive probability in libbloom [1]. Section 4.5 demonstrates various types of hash functions experimentally. Also, Section 4.6 compares robustBF and libbloom. Finally, Section 4.7 concludes the chapter.

4.2 Hash functions

There are numerous string hash functions, which are broadly categorized into two key categories, namely, cryptographic and non-cryptographic string hash functions. Examples of cryptographic hash functions are MD5, SHA1, SHA2, and SHA3; non-cryptographic hash functions are FastHash, XXHash, Murmur, SuperFast, etc. Choosing a correct hash function is essential for designing an efficient Bloom Filter, which is often confusing. The secure hash function increases the execution time, which is significantly disadvantageous for designing a Bloom Filter. Moreover, the hash function for Bloom Filter requires a seed value to distribute the input item into different slots in the bit array. Therefore, the secure hash function is unsuitable for Bloom Filter because it does not support seed value.

For more clarification, we take an example of SHA256 by assuming a seed value. SHA256 generates a hash value which is 256-bit long. A modulus operation is performed with the generated hash value. Let us assume that the bit array of the Bloom Filter is of length 10000, and the secure hash value is h_v. To place an item in a single-bit location, it performs $h_v\%10000$. Clearly, it shows that a 256-bit hash value is a waste because the modulus operation truncates entire bits. It does not require a large hash value. Normally, the Bloom Filter size is a few million bits.

 https://doi.org/10.1016/B978-0-12-823520-1.00009-2

Copyright © 2023 Elsevier Inc. All rights reserved.

Moreover, the cryptographic hash functions are designed for large-sized bits. Therefore, the 256-bit size becomes enormous for a few million bits. Additionally, a large bit-sized hash function requires more time compared to a lower bit-sized hash function. Intuitively, we can conclude that the non-cryptographic hash function is faster than any other cryptographic hash function because the cryptographic hash function produces more bits than the non-cryptographic hash function. Notably, a large-sized hash value is not essential to reduce the number of false positives and increase the performance of the Bloom Filter. If the bit array size of Bloom Filter is approximately equal to 2^{256}, then the SHA256 hash function can evenly distribute the input items in the Bloom Filter array. Practically, a 32-bit hash function is enough for the implementation of the Bloom Filter for the current requirement. The rest of the section briefly discusses various non-cryptographic string hash functions.

4.2.0.1 FastHash

FastHash [2] is a simple non-cryptographic string hash function. By default, FastHash produces 64 bits of hash code. For 32 bits of hash code, it deducts 32-bit code from 64-bit hash code. It is similar to the Murmur hash function.

4.2.0.2 xxHash

xxHash [3] is a non-cryptographic hash algorithm developed by Yann Collet. It optimizes all operations to execute faster. It partitions the input items into four independent streams. The responsibility of each stream is to execute a block of 4 bytes per step. Each stream stores a temporary state. In the final step, all four states are combined to obtain a single state. The most important advantage of xxHash is that its code generator gets many opportunities to reorder opcodes to prevent delay.

4.2.0.3 Murmur

Murmur was designed by Austin Appleby in 2008 [4]. The name is constructed using two basic operations, namely, multiply (MU) and rotate (R). These two operations are repeatedly used to construct the hash value. It is a non-cryptographic hash function which helps in common hash-based queries. Various versions are also developed to improve performance, the latest version is MurmurHash3 [5].

4.3 Prime numbers in Bloom Filter

Prime numbers are the most important numbers in hashing techniques. If a table size is a prime number, then it evenly distributes the items in all slots in the hash table [6]. Therefore, a prime number is always chosen as a hash-table size. It significantly reduces the collision probability in hashing. The prime number also plays a vital role in Bloom Filter. However, the conventional Bloom Filter allocates memory based on $m = -\frac{n \ln \varepsilon}{(\ln\ 2)^2}$ where m is not necessarily a prime number. Therefore, it suffers from a higher false positive probability. Hence, it requires more hash functions to reduce the false positive probability. Notably, if m is a prime number, fewer hash functions can achieve the desired false positive probability while significantly enhancing the performance.

4.4 Number of hash functions

We conduct an experiment to reveal the relation between the number of hash functions and the false positive probability in the standard Bloom Filter, libbloom [1,7]. We input 10 million integers and query another 10 million integers. We fixed the false positive probability to 0.001, and the number of hash functions ranged from 1 to 80. Fig. 4.1 demonstrates the relation between the number of hash functions and the false positive probability. It illustrates that fewer hash functions have a high false positive probability, whereas an increase in the number of hash functions also increases the false positive probability. More hash functions should reduce the false positive probability; instead, they increase the false positive probability because more hash functions fill more 1s in the bit array of the Bloom Filter. Therefore, the number of hash functions should be optimal. This requires calculating an optimal number of hash functions to maintain a low false positive probability in the Bloom Filter. In Fig. 4.1, the false positive probability is very high if the number of hash functions is less or more than 10. The optimal number of hash functions is 10 for 10 million inputs in libbloom. Moreover, the number of hash functions directly impacts the filter's performance.

Alternatively, more function calls decrease the performance of the filter. Therefore, the conventional Bloom Filter calculates the number of hash functions as $k = \frac{m}{n} \ln 2$ where m is the bit size, and n is the number of input items.

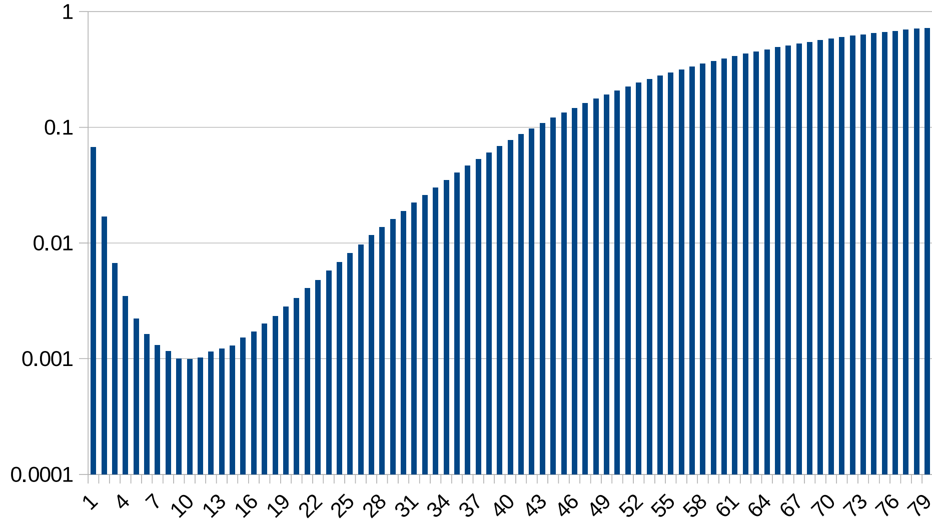


FIGURE 4.1 Relation between the number of hash functions and the false positive probability.

4.5 Types of hash functions

The key objective of this experimentation is to understand the behavior of hash functions. We demonstrate how a large-sized hash function adversely affects Bloom Filter. We conducted experiments on robustBF (source code of robustBF can be found at https://github.com/patgiri/robustBF). The code is written in C programming language. Our experimental environment is given in Table 4.1.

TABLE 4.1 Experimental environment.

Configuration	Description
CPU	Intel(R) Core(TM) i7-7700 CPU @ 3.60 GHz
RAM	8 GB
Operating System	Ubuntu 18.04.6 LTS
Programming Language	GNU C 7.5.0

There are many types of hash functions, and we select a non-cryptographic string hash function for experimentation, producing a non-secure hash value. However, we exclude SuperFast hash function in the experimentation due to a lack of seed value.

In our experiment, we use Same Set, Mixed Set, Disjoint Set, and Random Set for the query. Let $X = \{x_1, x_2, x_3, \ldots\}$ be a set of input items. Let $Q = \{q_1, q_2, q_3, \ldots\}$ be the set of query set, then the Same Set, Mixed Set, Disjoint Set, and Random Set can be defined as shown in Definitions 4.1, 4.2, 4.3, and 4.4, respectively.

Definition 4.1. *If the input and query items satisfy the condition that $X \cap Q = X$ or $X \cap Q = Q$, then we term it as a Same Set, i.e., the input and query items do not differ.*

Definition 4.2. *If the input and query items satisfy the condition that $X \cap Q = \emptyset$, then we term it as a Disjoint Set, i.e., the input and query sets are disjoint.*

Definition 4.3. *Let us divide the query set into two sets $Q = \{Q_1, Q_2\}$. We term the query set as Mixed Set if either (a) or (b) holds, where*

(a) $X \cap Q_1 = Q_1$ *and* $X \cap Q_2 = \emptyset$,
(b) $X \cap Q_1 = \emptyset$ *and* $X \cap Q_2 = Q_2$.

Definition 4.4. *If the input item set X is inserted into Bloom Filter and query item set Q is queried randomly in the Bloom Filter, then the query set is termed as a Random Set.*

TABLE 4.2 Time taken by the hash functions in various operations for 10 million inputs in robustBF, in seconds. Lower is better.

Process	FastHash	XXHash	Murmur2	Murmur64
Insertion	1.417302	1.538096	1.157248	1.311276
Same Set	1.389863	1.536134	1.156138	1.305726
Mixed Set	1.360552	1.591233	1.089147	1.300329
Disjoint Set	1.5812	1.861303	1.292238	1.529559
Random Set	1.54792	1.827468	1.25729	1.484729

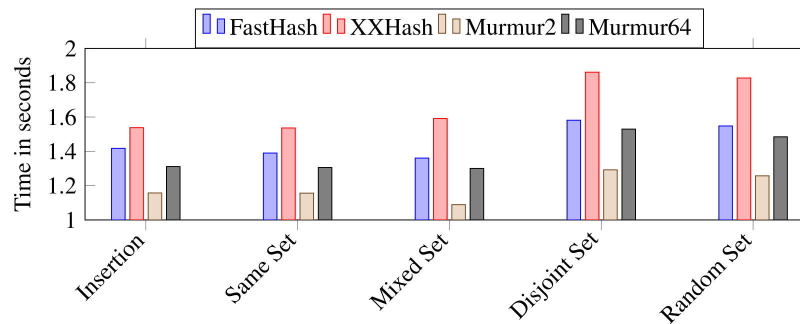


FIGURE 4.2 Time taken in diverse operations of 10 million in robustBF using various non-cryptographic string hash functions. Lower is better.

Table 4.2 and Fig. 4.2 demonstrate the performance of robustBF in diverse operations using four different hash functions. The experiment is carried out for 10 million data on robustBF. We deploy various hash functions, namely FastHash [2], XXHash [3], Murmur2 [4], and Murmur64 [4]. The query performance is measured using various query operations, namely, Same Set, Mixed Set, Disjoint Set, and Random Set. We also measure the insertion performance of 10 million data into robustBF. The input and query data are 10 million integers; however, the behavior of robustBF with the selected hash function will also be similar to the real dataset. The integer dataset is used to unearth the strength and weaknesses of a filter.

Fig. 4.2 shows the performance of robustBF with different hash functions. The Murmur2 hash function outperforms all other hash functions, but the performance of Murmur2 degrades due to the large-sized hash value. The XXHash hash function is the slowest in all operations (insertion and query). The performance of robustBF with the Murmur64 hash function is the second fastest because Murmur64 produces a 64-bit hash value. Thus, it is slower than the 32-bit version of Murmur2.

TABLE 4.3 False positive probability of robustBF with various hash functions.

Process	FastHash	XXHash	Murmur2	Murmur64
Same Set	0	0	0	0
Mixed Set	0.309517	0.309541	0	0.309453
Disjoint Set	0.619185	0.619329	4E-06	0.619123
Random Set	0.616292	0.616461	3E-06	0.615966

Table 4.3 and Fig. 4.3 demonstrate the false positive probability. The desired false positive probability is 0.001, and we set the total number of hash function calls to two (2). Therefore, robustBF with FastHash, XXHash and Murmur64 cannot achieve the desired false positive probability; however, robustBF with Murmur2 can achieve the desired false positive probability, and it is approximately zero. Therefore, it is important to select a good hash function for Bloom Filter. However, robustBF with Murmur2 hash function has no false positive probability in three or more

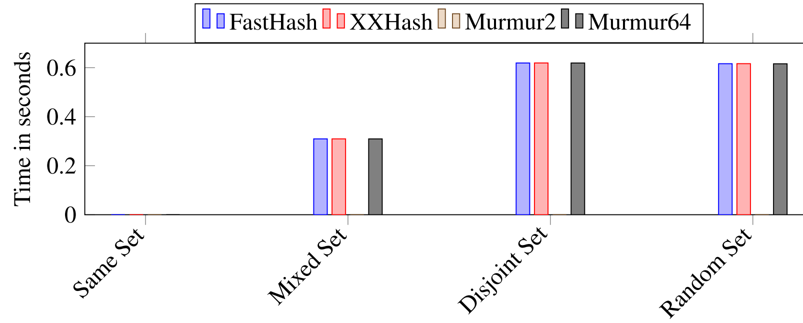


FIGURE 4.3 False positive probability of robustBF with various hash functions. Lower is better.

hash function calls. Alternatively, if we use more than three bits in placing a single item, then robustBF has no false positive probability without the sacrificing memory footprint.

There are two ways to reduce the false positive probability of robustBF with FastHash and XXHash, and these are: (a) increase the number of hash function calls and (b) increase the memory size. However, robustBF with Murmur2 has already exhibited almost zero false positive probability with the memory size of 1.62 MB for 10 million input items.

TABLE 4.4 Accuracy of robustBF with various hash functions.

Process	FastHash	XXHash	Murmur2	Murmur64
Same Set	100	100	100	100
Mixed Set	69.0483	69.0459	100	69.05467
Disjoint Set	38.08155	38.06709	99.99959	38.08771
Random Set	38.37078	38.35391	99.9997	38.40342

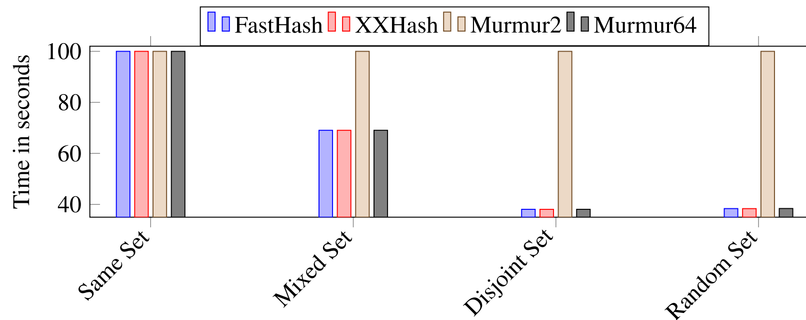


FIGURE 4.4 Accuracy of query process of 10 million items on robustBF with various hash functions. Higher is better.

Table 4.4 and Fig. 4.4 demonstrate the accuracy of robustBF with different hash functions. The accuracy is calculated as $accuracy = 1 - FP$ because there is no false negative in robustBF and in a conventional Bloom Filter. The accuracy is 100% in the Same Set query operation; however, the accuracy differs in other query operations. The minimum accuracy of robustBF with Murmur2 is 99.99959%, and the maximum is 100%.

4.6 Comparison of robustBF and libbloom

Bloom Filter converts input items into a few bits, namely k bits where k is the number of hash functions. For instance, an image size of 10 million is also converted into k bits. However, this does not imply bits per item. The bits per item are calculated as $bpi = \frac{m}{n}$ where m is the total number of bits, and n is the total number of input items. Table 4.5 demonstrates the memory requirement for robustBF and libbloom in MB. libbloom takes a lot of memory

for 10 million input items. robustBF takes 1.62 MB of memory, whereas libbloom takes 17.14 MB for 10 million input items. robustBF takes 10.59× less memory than libbloom, which is significant. Therefore, robustBF is able to save 90.55% of memory over libbloom. The bits per item of robustBF and libbloom are 1.36 and 14.38 bits, respectively. Therefore, robustBF uses significantly less memory. Notably, libbloom also uses the Murmur2 hash function.

TABLE 4.5 Memory footprint for 10 million input items for both robustBF and libbloom with a desired false positive probability of 0.001.

Name	Memory in MB
robustBF	1.618935
libbloom	17.139420

Table 4.6 demonstrates a comparison of time taken by robustBF and libbloom in diverse operations using 10 million data items, namely, for insertion and query. The query test cases are the Same Set, Mixed Set, Disjoint Set, and Random Set. libbloom and robustBF take 4.39 and 1.16 seconds to insert 10 million items, respectively. robustBF is 3.79× faster than libbloom in the insertion operation. Similarly, robustBF is faster than libbloom in every aspect. We can make libbloom faster if we reduce the number of hash function calls to two, but we cannot achieve the desired false positive probability with two hash functions. Therefore, libbloom takes ten hash function calls to achieve the desired false positive probability of 0.001 for 10 million items.

TABLE 4.6 Time-taken for 10 million input items by both robustBF and libbloom with a desired false positive probability of 0.001.

Name	Insertion	Same Set	Mixed Set	Disjoint Set	Random Set
libbloom	4.390539	3.670390	3.032267	2.690225	2.634851
robustBF	1.157248	1.156138	1.089147	1.292238	1.25729

Table 4.7 demonstrates the false positive probability of libbloom and robustBF for 10 million items. robustBF exhibits its false positive probability of 0.000004 and 0.000003 in the Disjoint Set and Random Set for 10 million items, respectively. robustBF exhibits zero false positive probability for the Same Set and Mixed Set. Similarly, libbloom achieves the desired false positive probability, but requires more memory than robustBF. It is slower than robustBF due to the use of more hash functions.

TABLE 4.7 Time-taken for 10 million input item by both robustBF and libbloom with a desired false positive probability of 0.001.

Name	Same Set	Mixed Set	Disjoint Set	Random Set
libbloom	0	0.001017	0.001011	0.000996
robustBF	0	0	4E-06	3E-06

4.7 Conclusion

This chapter aimed to discuss the role of the hash function in Bloom Filter. Initially, we have raised some questions, which we have answered experimentally in this chapter. Bloom Filter using non-cryptographic hash functions performs better than cryptographic hash functions because cryptographic hash functions require more time for the generation of hashed value and more memory to store it. The number of hash functions plays an important role in the performance of the Bloom Filter. A prime number of hash functions lead to a smaller false positive probability. Furthermore, the high performance of Bloom Filter requires the calculation of an optimal number of hash functions. The lowest number of hash functions has a high false positive probability; on the contrary, a higher number of hash functions also has a high false positive probability. This chapter verified these statements experimentally using various hash functions in standard Bloom Filter, libbloom, and robustBF.

References

[1] J.J. Virkki, libbloom, Retrieved in 2022 from https://github.com/jvirkki/libbloom.
[2] Eric, Fasthash, Retrieved on April 2020 from https://github.com/ztanml/fast-hash.
[3] Y. Collet, Xxhash, Retrieved on Aug 2019 from https://create.stephan-brumme.com/xxhash/, 2004.
[4] A. Appleby, MurmurHash, https://sites.google.com/site/murmurhash/. (Accessed July 2022).
[5] A. Horvath, Murmurhash3, an ultra fast hash algorithm for C# / .net, https://blog.teamleadnet.com/2012/08/murmurhash3-ultra-fast-hash-algorithm.html. (Accessed July 2022).
[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, MIT Press, 2022.
[7] A. Kirsch, M. Mitzenmacher, Less hashing, same performance: building a better Bloom filter, in: Algorithms – ESA 2006, Springer, Berlin, Germany, 2006, pp. 456–467.

CHAPTER

5

Analysis on Bloom Filter: performance, memory, and false positive probability

5.1 Introduction

Bloom Filter has a key issue of false positive probability. Many researchers are focusing on how to reduce the false positive probability of the Bloom Filter. However, there is an important trade-off between the reduction of the false positive probability and the memory footprint. The false positive probability can be reduced significantly if we increase the memory footprint, but it invalidates the property of Bloom Filter of having a tiny memory footprint. Otherwise, it becomes a conventional hash-table data structure.

Many researchers have already proposed a false-positive fee zone for Bloom Filter [1–3]; however, memory footprint and query performance may suffer. There is an important trade-off between the false positive probability and the number of hash functions, the false positive probability and the memory consumption, and the number of hash functions and the query performance. In this chapter, we discuss all the important parameters of the Bloom Filter theoretically and experimentally. Moreover, the hash function plays a vital role in Bloom Filter's performance. Also, it helps in reducing memory consumption and false positive probability. Therefore, we experimentally show the performance of different variants of hash functions.

5.2 False positive probability

False positive probability (FPP) is the crucial issue of the Bloom Filter. As we know, the Bloom Filter is an approximate data structure; therefore, it is important to understand the FPP of the Bloom Filter. Unlike deterministic data structures, it never guarantees whether an item is a member of the Bloom Filter or not. If Bloom Filter returns $True$, then it means that the item may be present without guarantee, i.e., the Bloom Filter may return a false positive. Therefore, it is crucial to understand the FPP probability of the Bloom Filter.

5.2.1 Approximate false positive analysis

Let m be the size of Bloom Filter, i.e., the bit array size is m bits. The probability of a certain bit is not set to 1 by a certain hash function is

$$1 - \frac{1}{m}. \tag{5.1}$$

Let k be the number of hash functions for insertion of an item to the Bloom Filter, then the probability of a certain bit not set to 1 by the k hash functions is

$$\left(1 - \frac{1}{m}\right)^k. \tag{5.2}$$

As we know,

$$\lim_{x \to \infty} \left(1 - \frac{1}{m}\right)^m = e^{-1}. \tag{5.3}$$

Bloom Filter https://doi.org/10.1016/B978-0-12-823520-1.00010-9
Copyright © 2023 Elsevier Inc. All rights reserved.

By rewriting Eq. (5.2) using Eq. (5.3), we get

$$\left(1-\frac{1}{m}\right)^{k}=\left(\left(1-\frac{1}{m}\right)^{m}\right)^{k/m}\approx e^{-k/m}. \tag{5.4}$$

Let n be the total number of items inserted into Bloom Filter, then the probability of a certain bit not set to 1 is

$$\left(1-\frac{1}{m}\right)^{kn}\approx e^{-kn/m}. \tag{5.5}$$

Now, the probability of the certain bit set to 1 is

$$1-\left(1-\frac{1}{m}\right)^{kn}\approx 1-e^{-kn/m}. \tag{5.6}$$

Therefore, the probability of all bits set to 1 by k hash function is

$$\varepsilon=\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k}\approx(1-e^{-kn/m})^{k}. \tag{5.7}$$

This ε is the false positive probability by inserting n items into m sized Bloom Filter by k hash functions. However, the calculation of ε is not correct due to the assumptions of the independence for the probabilities of each bit being set to 1 [4].

5.2.2 Exact false positive analysis

The exact false positive probability was first described by Bose et al. [5], and later by Christensen et al. [4] and F. Grandi [6]. Bose et al. [5] describes the exact false positive probability using balls-and-bins problems. F. Grandi [6] modifies the exact false positive probability using the γ-transformation [7]. Conditioning on that $X=x$ of bits are set to 1 [6], we get

$$Pr(FP \mid X=x)=\left(\frac{x}{m}\right)^{k}. \tag{5.8}$$

Then, the exact probability is calculated using the law of total probability as

$$\begin{aligned} FPP &= \sum_{x=0}^{m} Pr(FP \mid X=x)Pr(X=x) \\ &= \sum_{x=0}^{m} Pr(FP \mid X=x)f(x) \end{aligned} \tag{5.9}$$

where $f(x)$ is the probability mass function of X. Using γ-transformation, the probability mass function is calculated as

$$f(x)=\binom{m}{x}\sum_{j=0}^{x}(-1)^{j}\binom{x}{j}\left(\frac{x-j}{m}\right)^{kn}. \tag{5.10}$$

Rewriting Eq. (5.9) using Eq. (5.10), we get

$$FPP=\sum_{x=0}^{m}\left(\frac{x}{m}\right)^{k}\binom{m}{x}\sum_{j=0}^{x}(-1)^{j}\binom{x}{j}\left(\frac{x-j}{m}\right)^{kn}. \tag{5.11}$$

Eq. (5.11) gives the exact false positive probability of Bloom Filter; however, Eq. (5.7) gives a lower false positive probability than the actual false positive rate.

5.3 Memory

Bloom Filter uses a tiny amount of main memory, and therefore, it can be applied to diverse applications, for instance, IoT. To calculate the memory requirement, we need to approximate the number of hash functions k. The value of k should not be too large or too small. The correct approximation of the total number of hash functions for each item is $k = \frac{m}{n} \ln 2$. By replacing the value of k in Eq. (5.7), we get

$$\varepsilon = \left(1 - e^{-(\frac{m}{n}\ln 2)\, n/m}\right)^{(\frac{m}{n}\ln 2)} \tag{5.12}$$

By taking ln of both sides, we get

$$\ln \varepsilon = -\frac{m}{n}(\ln 2)^2. \tag{5.13}$$

Therefore, the total memory requirement of the Bloom Filter is

$$m = -\frac{n \ln \varepsilon}{(\ln 2)^2}. \tag{5.14}$$

The memory requirement may differ if the architecture of Bloom Filter differs. Thus, the optimal number of bits per item is

$$\frac{m}{n} = -\frac{\log_2 \varepsilon}{\ln 2} = -1.44 \log_2 \varepsilon. \tag{5.15}$$

From the above equations, we can calculate all required parameters for the conventional Bloom Filters.

5.4 Performance

This section shows the experimental evaluation of the conventional Bloom Filter, libbloom [8]. We conducted the experiment on a Desktop computer with Ubuntu 18.04.6 LTS. The processor is Intel Core i7-7700 CPU @ 3.60 GHz × 8, with the memory capacity of 8 GB. The source code is compiled with GCC version 7.5.0 compiler.

5.4.1 Dataset

We have conducted our experiment using a synthesis dataset, i.e., the dataset is generated by a computer. We insert 1 million (1M) integers ranging from 1 to 1000000. To verify the correctness of the Bloom Filter, we perform a query operation on the inserted Bloom Filter, and the query set ranges from 1000001 to 2000000. On the contrary, the real dataset contains repetitive items, and hence fewer memory requirements. This synthetic dataset is used to measure the strength of the Bloom Filter.

5.4.2 Bloom Filter settings

We have conducted our experiment on libbloom [8] which is an implementation of the standard Bloom Filter proposed by Kirsch and Mitzenmacher [9,10]. Bloom Filter parameters are set before running it, and therefore, we set the false positive probability to 0.001, and the input item size is 1M. Bloom Filter adjusts the memory size and the number of hash functions automatically; however, we have manually set the number of hash functions to unearth the trade-off between the number of hash functions and the false positive probability. The memory requirement for the Bloom Filter is 14377587 bits (1.71 MB).

5.4.3 Experimental results

Fig. 5.1 demonstrates the trade-off between false positive probability by fixing the input size and memory size in the log chart. The exact value of the chart is not reproducible, but the exact shape of the curve is reproducible. Initially, the false positive probability decreases when the number of hash functions increases. The false positive

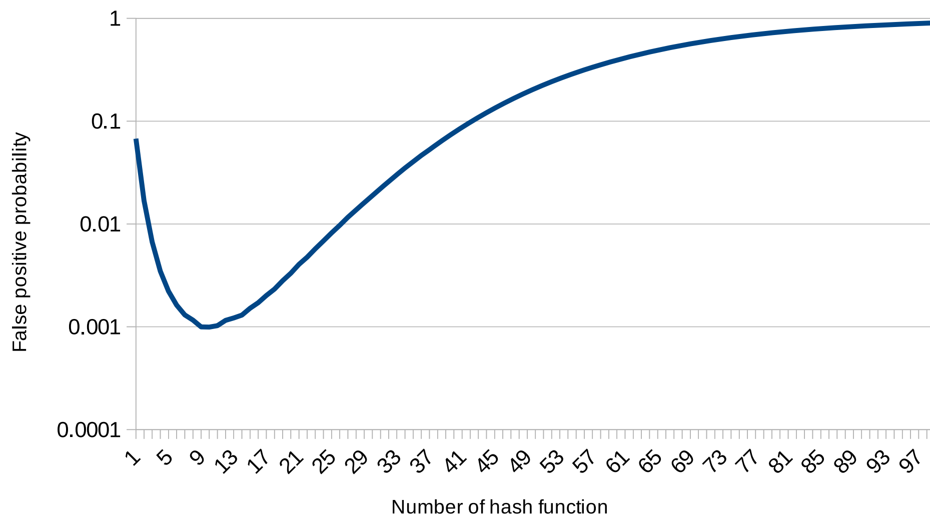


FIGURE 5.1 Trade-off between false positive probability and number of hash function in log chart.

probability decreases sharply for the number of hash functions from 1 to 10. Then, the false positive rises continuously. Alternatively, the false positive probability increases if the number of hash functions increases, but the number of hash functions cannot be very small. As shown in Fig. 5.1, the false positive probability becomes almost 1 from 80 numbers of hash functions. The reason behind this tick shape of the curve is that a large number of hash functions fill many 1s in the Bloom Filter array. Therefore, the false positive probability should decrease with the increase of the number of hash functions, but it increases. Moreover, the false positive probability with a small number of hash functions also raises the false positive probability. Therefore, the number of hash functions should be optimal, i.e., the number of hash functions should not be too large or too small. Hence, the optimal number of hash functions is 10 for 1M input items with the false positive probability of 0.001. It can be confusing that false positive probability is set to 0.001, and Fig. 5.1 shows variable false positive probability. The desired false positive probability is set to 0.001, and the experimentation examines when we can achieve this desired false positive probability. Thus, the desired false positive probability is achieved on 10 numbers of hash functions as shown in Fig. 5.1. On the contrary, the number of hash functions may differ in different Bloom Filter architectures. For instance, robustBF [11] achieves its desired false positive probability with fewer hash functions without sacrificing the memory footprint. Therefore, it features a fast and memory-efficient Bloom Filter.

Fig. 5.2 demonstrates the disadvantage of a large number of hash functions. A large number of hash functions directly affect the performance of the Bloom Filter. A large number of hash functions slow down the execution of the Bloom Filter. Therefore, the large number of hash functions is undesirable in enhancing the performance of the Bloom Filter. A developer must take care of the number of hash functions to efficiently deal with the false positive probability and execution time. The memory requirements of Bloom Filter can vary with different architecture. Therefore, the Bloom Filter designer must take care of the memory footprint, too, because most Bloom Filters run on an embedded system, and the embedded system has limited memory size. Therefore, Kirsch and Mitzenmacher [9,10] reduced the number of hash functions using double hashing concepts.

We evaluate the Bloom Filter using different hash functions for exposing the performance and false positive probability. The parameter settings are the same as earlier. Fig. 5.3 demonstrates the performance of the Bloom Filter. The performance is different for the different variant hash functions. Fig. 5.3 shows that the Murmur hash function outperforms the FastHash and xxHash functions in both insertion and query operations. However, the xxHash function is slower than the FastHash function.

Fig. 5.4 demonstrates the false positive probability of the Bloom Filter with different hash functions. The desired false positive probability is set to 0.001. The Bloom Filter with the Murmur hash function achieves the false positive probability of 0.000994, whereas other hash functions have a higher false positive probability than Murmur hash functions, as shown in Fig. 5.4. Therefore, the performance of the Bloom Filter depends not only on the number of hash functions but also on hash algorithms. Notably, Bloom Filter with CRC32, MD5, and SHA hash functions has a higher false positive probability than the non-cryptographic hash functions and higher time for query execution

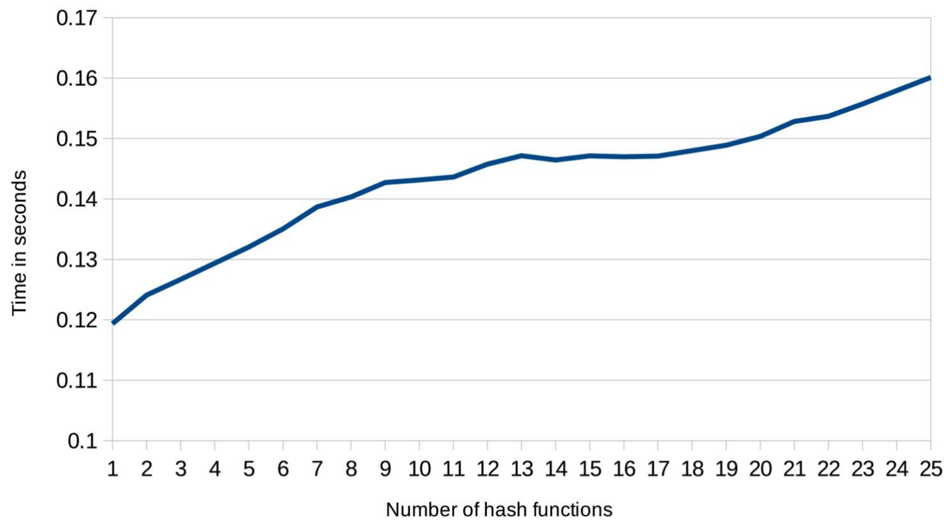


FIGURE 5.2 Trade-off between number of hash functions and execution time in seconds.

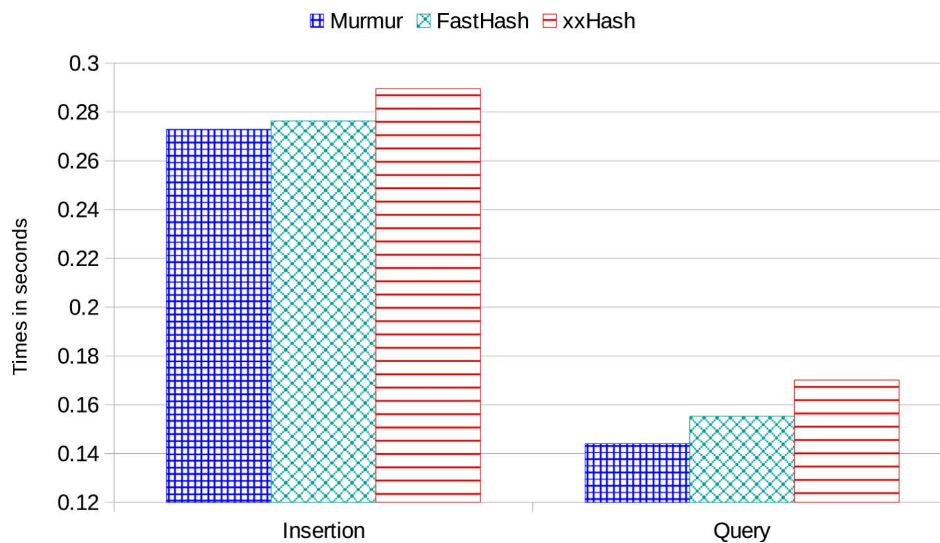


FIGURE 5.3 Performance of various hash functions on Bloom Filter.

as reported by Patgiri et al. [12]. A cryptographic hash function produces large hash values, and these hash values are transformed with the modulus operator to locate the desired bit position of the Bloom Filter array. Let us assume that H_v is the hash value produced by a cryptographic hash algorithm, and its size is H_s. Now, the Bloom Filter array size is m where $m < H_s$. To locate the desired position of Bloom Filter's bit array, we perform $H_v \% m$ where % is the modulus operator. Due to this operation, large hash values are truncated to find the desired bit position. Therefore, secure hash functions suffer in Bloom Filter even if they produce high-quality hash value.

To design a new variant of Bloom Filter, one requires a fast hash algorithm which reduces the false positive probability. As we know, the false positive probability is the key issue of the Bloom Filter; therefore, it should be reduced as much as possible. It is not possible to eradicate the false positive probability from Bloom Filter; however, it can be reduced. Many researchers are focusing on reducing the false positive probability by sacrificing the memory footprint. If we increase the memory footprint, then the false positive probability reduces drastically. On the contrary, the Bloom Filter is used for approximation of input items where it occupies negligible memory. A good hash function always reduces the false positive probability and memory consumption. Moreover, it can also enhance query performance.

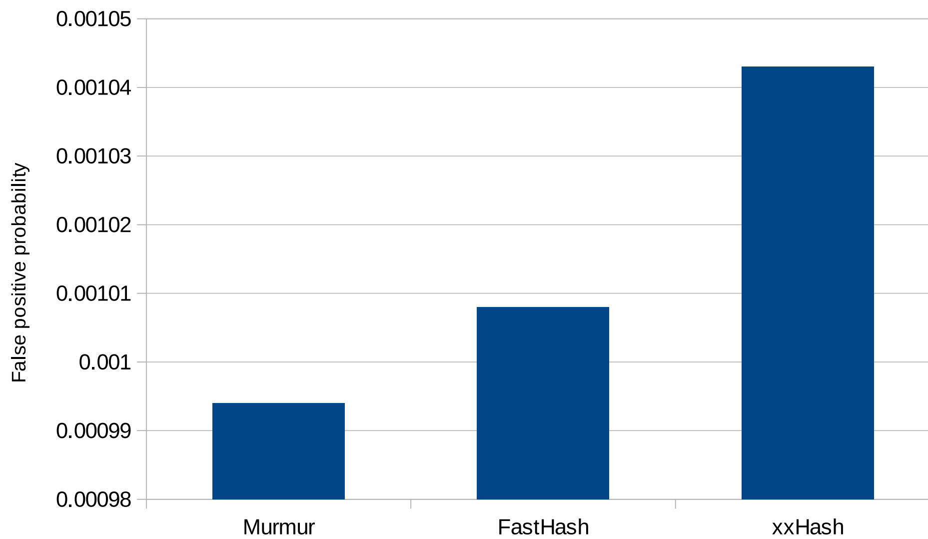


FIGURE 5.4 False positive probability of various hash functions on Bloom Filter.

5.5 Conclusion

This chapter demonstrated how to calculate the false positive probability for the conventional Bloom Filter. Moreover, we have calculated how to estimate the memory requirements for the Bloom Filter. In addition, we have demonstrated the important trade-off between the false positive probability and the number of hash functions. We conclude that a large number of hash functions does not help in reducing false positive probability; rather, it increases the false positive probability. Also, a small number of hash functions cannot reduce the false positive probability. Therefore, it requires an optimal number of hash functions which is demonstrated using the mathematical equations. Furthermore, the required optimal number of hash functions is demonstrated experimentally. In addition, the number of hash functions affects the performance of the Bloom Filter, which is demonstrated experimentally.

References

[1] J. Tapolcai, J. Bíró, P. Babarczi, A. Gulyás, Z. Heszberger, D. Trossen, Optimal false-positive-free Bloom filter design for scalable multicast forwarding, IEEE/ACM Trans. Netw. 23 (6) (2014) 1832–1845, https://doi.org/10.1109/TNET.2014.2342155.

[2] S.Z. Kiss, É. Hosszu, J. Tapolcai, L. Rónyai, O. Rottenstreich, Bloom filter with a false positive free zone, IEEE Trans. Netw. Serv. Manag. 18 (2) (2021) 2334–2349, https://doi.org/10.1109/TNSM.2021.3059075.

[3] O. Rottenstreich, P. Reviriego, E. Porat, S. Muthukrishnan, Constructions and applications for accurate counting of the Bloom filter false positive free zone, in: SOSR'20: Proceedings of the Symposium on SDN Research, Association for Computing Machinery, New York, NY, USA, 2020, pp. 135–145.

[4] K. Christensen, A. Roginsky, M. Jimeno, A new analysis of the false positive rate of a Bloom filter, Inf. Process. Lett. 110 (21) (2010) 944–949, https://doi.org/10.1016/j.ipl.2010.07.024.

[5] P. Bose, H. Guo, E. Kranakis, A. Maheshwari, P. Morin, J. Morrison, M. Smid, Y. Tang, On the false-positive rate of Bloom filters, Inf. Process. Lett. 108 (4) (2008) 210–213, https://doi.org/10.1016/j.ipl.2008.05.018.

[6] F. Grandi, On the analysis of Bloom filters, Inf. Process. Lett. 129 (2018) 35–39, https://doi.org/10.1016/j.ipl.2017.09.004.

[7] F. Grandi, The γ-transform approach: a new method for the study of a discrete and finite random variable, Int. J. Math. Models Methods Appl. Sci. 9 (2015) 624–635.

[8] J.J. Virkki, libbloom, Retrieved in 2022 from https://github.com/jvirkki/libbloom.

[9] A. Kirsch, M. Mitzenmacher, Less hashing, same performance: building a better Bloom filter, in: Algorithms – ESA 2006, Springer, Berlin, Germany, 2006, pp. 456–467.

[10] A. Kirsch, M. Mitzenmacher, Less hashing, same performance: building a better Bloom filter, Random Struct. Algorithms 33 (2) (2008) 187–218, https://doi.org/10.1002/rsa.20208.

[11] S. Nayak, R. Patgiri, RobustBF: a high accuracy and memory efficient 2D Bloom filter, arXiv:2106.04365, https://doi.org/10.48550/arXiv.2106.04365.

[12] R. Patgiri, S. Nayak, N.B. Muppalaneni, Is Bloom filter a bad choice for security and privacy?, in: 2021 International Conference on Information Networking (ICOIN), 2021, pp. 648–653.

CHAPTER

6

Does not Bloom Filter bloom in membership filtering?

6.1 Introduction

Bloom Filter is a probabilistic data structure that speeds up data filtering. It has many applications, but also many limitations. Bloom Filter cannot be used in all applications, in particular those requiring the exact response of a query. Yet, it has seen diverse applications; for instance, networking algorithms use Bloom Filter to filter packets. Network security algorithms deploy Bloom Filter to detect malicious URLs. DoS and DDoS algorithms implement Bloom Filter for detecting the same packets. Similarly, Bloom Filter has vast applications for filtering because it reduces the traffic of queries that need to be checked in the main system. Furthermore, the high-speed query operation of Bloom Filter increases the efficiency of the whole system. To this list of applications, we add another domain, that is, machine learning.

Machine learning algorithms have two parts, training and testing. Past known data are used by machine learning algorithms for training. After this stage, the machine learning algorithm generates inferences or models. This model is used to predict the new data. Some examples of machine learning algorithms are linear regression, naive Bayes, random forest, and deep learning. However, machine learning algorithms are compute- and data-intensive. A large dataset used for training increases the accuracy of prediction and also burdens the computation. Furthermore, query operation during testing is slow. Recently, Bloom Filter has been explored to improve the performance of machine learning algorithms. Bloom Filter is not a complete system. It does not store the original data, but rather maps the data into its data structure. As Bloom Filter has no false negatives, all negative response queries are correct; however, true response queries may be true positives or false positives. Hence, true response queries are forwarded to the application/system for verification. Although this may be true, Bloom Filter is deployed in various applications to improve performance. The simple data structure, small size, and low time complexity of operation help in enhancing the performance.

This chapter tries to elaborate on why, despite being not a complete system, Bloom Filter is gaining popularity, and new domains are exploring Bloom Filter to solve their problems. It also includes the limitation of the Bloom Filter for filtering applications and adaptability. For better understanding, the chapter reviewed two articles that implement Bloom Filter to enhance machine learning-based applications. Furthermore, the chapter proposes a Bloom Filter and deep learning-based malicious URL detection system. Finally, the chapter discusses the limitation of the Bloom Filter in filtering applications.

6.2 Bloom Filter is not a complete system!

One big issue with the filtering system is data storage. The data needs to be stored to perform query operations. The data volume is directly proportional to the complexity of searching algorithms. Many search algorithms are proposed; however, they have high time complexity due to data complexity. The complexity of searching algorithms is reduced by storing the data following an efficient data organization. While struggling to design an efficient and low-time complexity search algorithm, the researchers are now facing the big challenge of Big Data, which refers to data that has high volume, velocity, and variety [1]. Volume refers to the data size, velocity is the speed of data generation, and variety is the types of data. However, Big Data has more features than these three, which further increases the complexity of data processing. Patgiri and Ahmed [2] proposed that Big Data has ${V_3}^{11} + C$ features: the

 Copyright © 2023 Elsevier Inc. All rights reserved.

11 Vs, three features of volume, and complexity. The 11 Vs are volume, velocity, variety, veracity, validity, value, visualization, variability, vendee, vase, and virtual. The three features of volume are voluminous, vacuum, and vitality. These features need to be considered while designing a search algorithm for Big Data. Due to the huge volume, Big Data cannot be stored in a single system. One solution for this is the distributed file system. The data are stored in multiple systems where a central system stores the metadata and the location of the data. During the search operation, the central system searches for the requested metadata and returns the data location. Again the data is searched in the system containing the data. While the researchers are coping with Big Data, the difficulty of dealing with it is escalating further due to the exponential generation of data called Big Data 2.0 [3]. The data volume is huge, making the commodity hardware incapable of performing operations on the data. High-performance computers will be needed to complete the operation within an acceptable time duration. Data searching is complex, but it is going to be more complex in the future.

Bloom Filter can be a solution for this Big Data issue. Bloom Filter is not a complete system because it does not store the data. The data cannot be retrieved from the Bloom Filter. But Bloom Filter speeds up the search operation for the system. Bloom Filter maps the data into its data structure for quick query response. Searching is required for all operations, mainly insertion, deletion, and query. The system performs search operations during insertion to prevent data repetition and memory waste. In case of deletion operation, searching helps to determine the data location and query for data status, namely whether it is present or absent from the system. Data storage or removal, i.e., insertion and deletion operations, respectively, are costly. Thus, Bloom Filter helps in the removal of this search overhead. On the contrary, Bloom Filter is also overhead for the system because it adds another layer of query operation. However, in the case of true negative queries, it is beneficial. A small-sized Bloom Filter can store a huge volume of data. Moreover, the small size helps in storing the Bloom Filter in the main memory, which further increases the speed of Bloom Filter's operations. Another advantage of deploying Bloom Filter is reducing the data traffic for the system. The high-speed Bloom Filter performs the operation. The true negatives are responded to by the Bloom Filter, whereas positive response queries may be forwarded to the system for verification. Hence, Bloom Filter reduces the traffic when the query is forwarded to the system. Thus, Bloom Filter is not a complete system but the best solution for Big Data applications.

6.2.1 Limitations of Bloom Filter

Bloom Filter is applicable in various fields, namely, Bioinformatics, Big Data, IoT, security, etc., to enhance performance. It was believed that Bloom Filter is not applicable in the security domain [4], but recent development suggests [5–7] that Bloom Filter can also be used to enhance the performance in the security domain. However, it has limitations. For instance, it cannot be applied in exact query-response requirements and hard real-time systems. However, it depends on how Bloom Filter is adapted for a system. Notably, Bloom Filter is not suitable to be an independent system, but it is used as a subsystem to enhance it. For instance, the hash-table can stand by itself, but Bloom Filter requires additional structures to provide high accuracy of the queries. Bloom Filter has a false positive issue but is false negative free. Therefore, it is inapplicable to the exact answer requirements. Unlike hashing, it responds to a query with a probability that might be a true positive. Therefore, Bloom Filter can be used to enhance the performance of a system, but it cannot be a stand-alone solution in many domains. Let us take an example of a password database, which stores passwords. If a Bloom Filter returns false, it is guaranteed that a password is not in the database. Bloom Filter can answer whether a password may be present in the password database. Therefore, it requires additional verification of the lookup process. If a password is a member of Bloom Filter, then the password must be exactly matched with the password database. Otherwise, Bloom Filter may allow unintended users due to false positives. Moreover, Bloom Filter improves the performance by preventing many query operations with a password database, where the response speed of Bloom Filter is much higher compared to a password database.

6.2.2 Applicability of Bloom Filter

The key challenge is how to adapt a Bloom Filter in a system such that it enhances the entire system's performance. For instance, BigTable uses Bloom Filter to avoid unnecessary access to hard disk [8]. Thus, BigTable enhances the performance dramatically. Now, we need to understand the overhead of employing Bloom Filter in a system. We assume a database lookup as demonstrated in Fig. 6.1. Let $\eta = Q_n + Q_f + Q_p$ be the total query to the database where Q_n is the total number of true negative queries, Q_f is the total number of false positive queries, and Q_p is the total number of true positive queries. The false positive queries are an overhead to the database. It requires querying

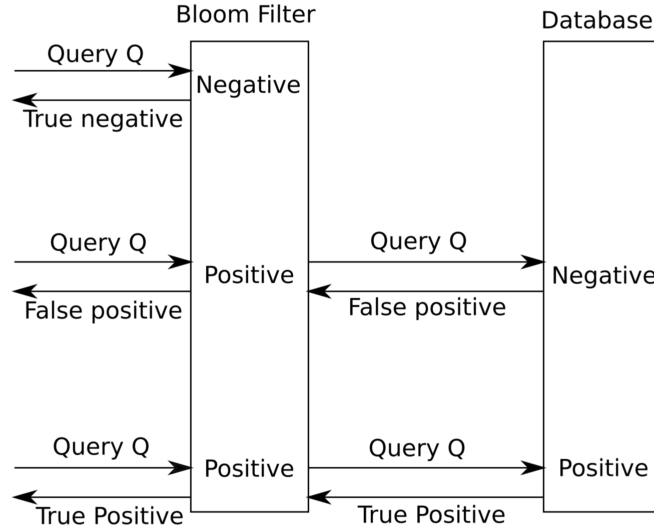


FIGURE 6.1 Query and their response in terms of database.

Bloom Filter as well as the database itself. Alternatively, the false positives are losses, and the true negatives are the profit. But the true positive is the state of both no loss and no profit. Now, if Bloom Filter returns a true positive, then it again needs to access the database for exact comparison because Bloom Filter cannot store the data. Therefore, the overhead is calculated as $overhead = Q_f$. The total benefit can be calculated as $benefit = Q_n$. Let us assume that all queries are true positives and no negative query arrives; then the Bloom Filter is an overhead. Therefore, the total overhead is $overhead = \eta$ and the total benefit $benefit = \eta - \eta = 0$. It indicates that Bloom Filter does not give any benefit to a system having no negative queries. Therefore, we calculate the benefit and overhead of Bloom Filter given in Eqs. (6.1) and (6.2), respectively:

$$Benefit = \begin{cases} 0 & \text{if } Q_n = 0, \\ Q_n - Q_f & \text{otherwise,} \end{cases} \tag{6.1}$$

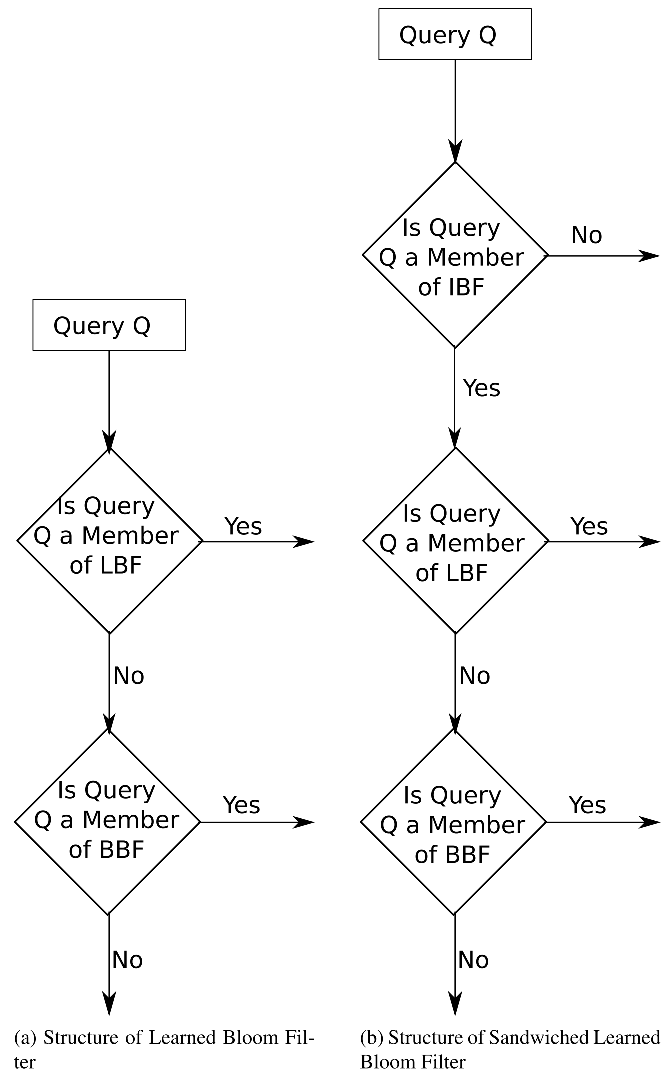
$$Overhead = \begin{cases} \eta & \text{if } Q_n = 0, \\ Q_f & \text{otherwise.} \end{cases} \tag{6.2}$$

From Eq. (6.1), we can easily estimate whether adapting a Bloom Filter is beneficial or not. Eq. (6.2) gives us the loss or overhead of using Bloom Filter in a system. However, the negative and positive queries are equally likely probable events in most systems. Therefore, using Bloom Filter is always beneficial in a system because of the negative queries. Also, it uses a tiny amount of memory. Moreover, the false positive probability can be tuned as desired by the user, which is a small number, for instance, 1%.

6.3 Learned Bloom Filter

We know that Bloom Filter cannot understand patterns, i.e., it cannot answer > or < queries. Therefore, it requires an additional learning model for the system to work. Alternatively, it requires an additional structure or model for > or < queries. Therefore, we explore how to adapt Bloom Filter in other functions for such kinds of queries.

Kraska et al. [9] introduced learned Bloom Filter (LBF) to increase the accuracy of the deep learning algorithm. Fig. 6.2a demonstrates the LBF proposed by Kraska et al. [9]. The LBF is trained using Neural Networks. The Neural Networks are used as a pre-filter for the LBF. The Neural Networks produce two keys, namely, positive and negative keys. The positive keys are correctly classified by the Neural Networks, and the negative keys are false negatives of the Neural Networks. The positive keys are inserted into LBF, and the negative keys are inserted into the backup Bloom Filter (BBF). For a query Q, it queries the LBF for membership. If LBF returns $True$, then it is assumed to be true positive. Otherwise, it queries the backup Bloom Filter. If the backup Bloom Filter returns $True$, then



(a) Structure of Learned Bloom Filter

(b) Structure of Sandwiched Learned Bloom Filter

FIGURE 6.2 Structures of learned Bloom Filters.

it is assumed to be true positive; otherwise, true negative. This kind of LBF increases the accuracy significantly. Moreover, Bloom Filter is a lightweight data structure that has a tiny memory footprint. However, LBF can return false positives, and therefore, the model can have lower accuracy than expected. Therefore, M. Mitzenmacher [10] proposed modified structures of LBF to address the false positive issue.

The LBF, in Fig. 6.2a, may return a false positive. Therefore, M. Mitzenmacher [10] added initial Bloom Filter (IBF), termed as sandwiched learned Bloom Filter, Sandwiched-LBF, for short. Fig. 6.2b demonstrates the Sandwiched-LBF by featuring three Bloom Filters. IBF contains all the keys, both positive and false negative, from the learned function (Neural Networks), LBF contains the positive keys, and BBF contains the false negative keys from the learned function (Neural Networks). We know that Bloom Filter guarantees no false negatives in a query process; therefore, if IBF returns $False$ for a query Q, then it is a true negative. It filters out most of the non-keys from the queries. Then, query Q is queried to LBF, and if LBF returns $True$, then it is assumed to be a true positive because the chance of the combined false positive in both IBF and LBF is low. If LBF returns $False$, then it might be in BBF. If BBF returns $True$, then it is assumed to be a true positive because the combined false positive probability of both IBF and BBF is low. If BBF returns $False$, then it is a true negative. When the IBF returns a false positive, the query Q reaches BBF, and BBF identifies it as a true negative. Thus, it significantly reduces the false positive probability of LBF. Also, it enhances the accuracy of Neural Networks dramatically. This kind of learned filter requires training on a huge dataset such that the results can be inserted into the Bloom Filters a priori.

6.4 Malicious URL filtering using Bloom Filter

Learned Bloom Filter contains the trained data from Neural Networks, which can be easily classified. On the contrary, Bloom Filter can also be used to enhance the performance of Deep Learning [5]. Fig. 6.3 demonstrates the Bloom Filter to improve Deep Learning. One application is filtering of malicious URLs. It uses two Bloom Filters, namely, malignant Bloom Filter $\mu\mathcal{BF}$ and benign Bloom Filter $b\mathcal{BF}$. The $\mu\mathcal{BF}$ contains malignant URLs from the learned model, whereas the $b\mathcal{BF}$ contains the benign URLs from the learned model. Deep Learning performs the training separately without integrating the Bloom Filter. After completion of training, the Bloom Filter is integrated with the Deep Learning model for future queries. The key idea is that if a query is already classified by the Deep Learning model, then it is unnecessary to classify the query again using the Deep Learning model. We know the Deep Learning algorithm is costly. Therefore, once a query is classified into malignant or benign, then the query can be inserted into a malignant or benign Bloom Filter. When the inserted query is queried again, then Bloom Filter can answer the query rather than query in the Deep Learning model. Therefore, Bloom Filter filters out all the seen or already classified queries. Thus, it prevents unnecessary testing of a query in a Deep Learning algorithm which is slower than Bloom Filter. A new query that is unseen or unclassified requires the Deep Learning model to classify it into malignant or benign.

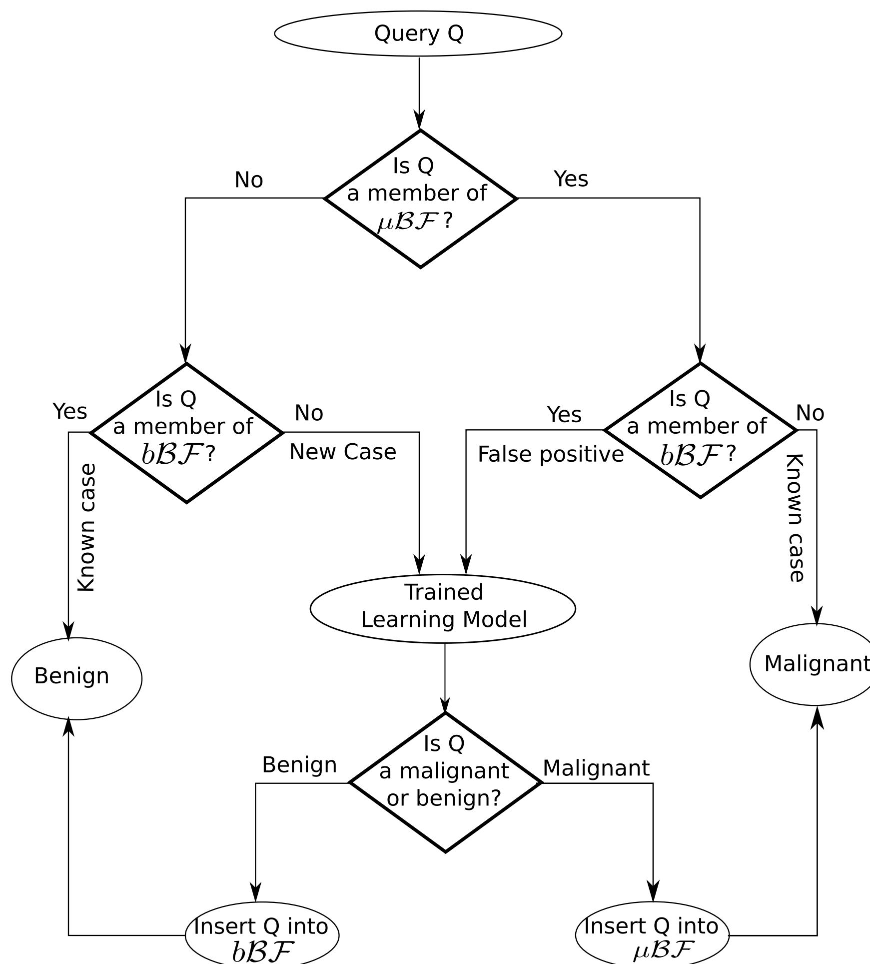


FIGURE 6.3 Adaptive Bloom Filter to speed up the execution process of Deep Learning.

Deep Learning can be trained using a malicious URL dataset. Deep Learning outputs malignant or benign sets. The malignant dataset is inserted into $\mu\mathcal{BF}$, and the benign dataset is inserted into $b\mathcal{BF}$. The trained Deep Learning model is integrated with Bloom Filter as demonstrated in Fig. 6.3. The trained Deep Learning model is used to classify input queries that are new.

Initially, a query Q, a URL, is queried to $\mu\mathcal{BF}$. If $\mu\mathcal{BF}$ returns $True$, then it might be a case of a false positive. Therefore, the URL Q is checked in $b\mathcal{BF}$. If $b\mathcal{BF}$ filter returns $False$, Q is a malignant URL. Otherwise, it is a false positive because either $\mu\mathcal{BF}$ or $b\mathcal{BF}$ is returning a false positive. Therefore, it should be classified in the Deep Learning model. Similarly, if a query Q is queried to $\mu\mathcal{BF}$ and returns $False$, then it is guaranteed that the item does not belong to $\mu\mathcal{BF}$. It does not mean that the item cannot be malignant. The query Q is queried to $b\mathcal{BF}$ for membership. If $\mu\mathcal{BF}$ returns $True$, then it is benign. Otherwise, it is a new URL where Bloom Filters cannot classify it as malignant or benign. Therefore, the query should be classified in the Deep Learning model.

The Deep Learning model can classify the new URL or false positive case as discussed above. If the Deep Learning model classifies the input query Q into malignant, the query Q is inserted into $\mu\mathcal{BF}$; otherwise, we insert Q into $b\mathcal{BF}$. This implies that the URL Q is already classified, and it will not be classified again by Deep Learning, where Bloom Filter can answer the query on the seen/classified items. Thus, this method avoids unnecessary checking of a query in Deep Learning, and hence, it enhances the overall performance.

6.5 Conclusion

Bloom Filter is attracting attention from many applications. Bloom Filter is not a complete system because it does not save the original data. However, the merits of Bloom Filter have are huge regarding this issue. Bloom Filter reduces the data traffic processed by the system. All negative response queries are handled by the Bloom Filter. In the case of true response queries, the queries are forwarded to the system to verify the response; however, it depends on the system. Bloom Filter has false positive issues, but less; hence, if the system can tolerate errors, then the true response queries are also handled by the Bloom Filter. On the contrary, Bloom Filter cannot be contrived in hard real-time systems or systems which require exact query-response. One important point to note is that Bloom Filter is an overhead to the system. So, the system has to determine whether the advantages of Bloom Filter exceed the overhead to the system.

References

[1] D. Laney, et al., 3D data management: controlling data volume, velocity and variety, META Group Res. Note 6 (70) (2001) 1.
[2] R. Patgiri, A. Ahmed, Big Data: the V's of the game changer paradigm, in: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), IEEE, 2016, pp. 17–24.
[3] S. Nayak, R. Patgiri, T.D. Singh, Big computing: where are we heading?, arXiv:2005.06964, 2020.
[4] A. Broder, M. Mitzenmacher, Network applications of Bloom filters: a survey, Internet Math. 1 (4) (2004) 485–509, https://doi.org/10.1080/15427951.2004.10129096.
[5] R. Patgiri, A. Biswas, S. Nayak, deepbf: malicious URL detection using learned Bloom filter and evolutionary deep learning, CoRR, arXiv:2103.12544 [abs], 2021, arXiv:2103.12544, https://arxiv.org/abs/2103.12544.
[6] Z. Dai, A. Shrivastava, Adaptive learned Bloom filter (Ada-BF): efficient utilization of the classifier with application to real-time information filtering on the web, in: H. Larochelle, M. Ranzato, R. Hadsell, M. Balcan, H. Lin (Eds.), Advances in Neural Information Processing Systems, vol. 33, Curran Associates, Inc., 2020, pp. 11700–11710, https://proceedings.neurips.cc/paper/2020/file/86b94dae7c6517ec1ac767fd2c136580-Paper.pdf.
[7] R. Patgiri, S. Nayak, S.K. Borgohain, PassDB: a password database with strict privacy protocol using 3D Bloom filter, Inf. Sci. 539 (2020) 157–176, https://doi.org/10.1016/j.ins.2020.05.135.
[8] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, ACM Trans. Comput. Syst. 26 (2) (jun 2008), https://doi.org/10.1145/1365815.1365816.
[9] T. Kraska, A. Beutel, E.H. Chi, J. Dean, N. Polyzotis, The case for learned index structures, in: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, Association for Computing Machinery, New York, NY, USA, 2018, pp. 489–504.
[10] M. Mitzenmacher, A model for learned Bloom filters and optimizing by sandwiching, Adv. Neural Inf. Process. Syst. 31 (2018), https://proceedings.neurips.cc/paper/2018/hash/0f49c89d1e7298bb9930789c8ed59d48-Abstract.html.

CHAPTER

7

Standard of Bloom Filter: a review

7.1 Introduction

The popularity of Bloom Filter is increasing day by day due to its capability of filtering extensive information. Bloom Filter has made a prominent place in the fields of Networking [1–3], Bioinformatics [4,5], Biometrics [6–8], IoT [9], and many other. With the start of the Big Data era, there is a requirement for a data structure to handle huge volumes of data. And Bloom Filter is fulfilling these requirements efficiently. Bloom Filter is highly adaptable. Therefore, it can be applied in many research areas. However, a conventional Bloom Filter has many issues, mainly those of false positive and false negative.

Consequently, Bloom Filter is affected more by false negatives than false positives. A false positive is defined when Bloom Filter responses $True$ in a query of an item when the item is not present in Bloom Filter. Similarly, a false positive is defined when Bloom Filter responses $False$ in a query of an item when the item is present in Bloom Filter. There is an overhead of searching for the data in the database in case of a false positive. However, a false positive creates disasters in deduplication, when a false positive shows that an item is a member of Bloom Filter while it is not present in the actual database. This plays a vital role in the deduplication process; however, an important item is filtered out due to false positives. On the contrary, a false negative is a result of the deletion operation. In data streaming, Bloom Filter is checked before sending a data packet. The data packet is transmitted repeatedly over the network due to false negatives, which creates more network traffic. In Bloom Filter, there is a searching overhead. In case of a false negative, the Bloom Filter returns a response that the data is absent. Then, a false negative response is returned to the client. The client is refused the requested data even if the data is present. Therefore, researchers have developed various variants of Bloom Filter. They are deployed in environments to gain in-built advantages.

This chapter is organized as follows: Sections 7.2 and 7.3 review the literature on various kinds of Bloom Filter. Section 7.4 concludes the chapter.

7.2 Literature review on standard Bloom Filter

This section provides a detailed literature review on Bloom Filter's variant. Moreover, we present the classification of these variants of Bloom Filter in Table 7.1.

Counting Bloom Filter (CBF) [10] is an extension of Bloom Filter, which solves the delete problem of the Bloom Filter. Conventional Bloom Filter does not support the deletion of an element due to the false negative issue. The deletion operation may delete the information mapped by the other elements and increase the false negative rate. Thus, counting Bloom Filter solves this problem by introducing a counter along with each slot. Each counter keeps track of the number of elements mapped to that slot. When an element is added to a slot for the first time, the slot is assigned the value 1, and the counter is incremented by 1. Again, if more elements are mapped to that particular slot, then only the counter is incremented. And, when an element is deleted, only the counter is decremented. And, when the counter becomes zero, the slot is assigned value 0. Intuitively, if a Bloom Filter supports the delete operation and it is not a counting Bloom Filter, then there is a probability of false negative.

Spectral Bloom Filter (SBF) [11] is an extension of the standard Bloom Filter used for multiset keys. In this Bloom Filter, each slot is a vector of counters C. Each counter represents the multiplicity of items. Initially, C is assigned to value 0. During insertion of an item x, $C_{h_1(x)}, C_{h_2(x)}, \ldots, C_{h_k(x)}$ are incremented by 1. Each slot stores the frequency of each item. During query operation, it returns the estimate of the frequency of the searched item. And, during deletion of an item x, $C_{h_1(x)}, C_{h_2(x)}, \ldots, C_{h_k(x)}$ are decremented by 1.

 https://doi.org/10.1016/B978-0-12-823520-1.00012-2 Copyright © 2023 Elsevier Inc. All rights reserved.

The Bloomier Filter [12] generalizes the Bloom Filter for high adaptability. The Bloomier Filter broadens the functions of the Bloom Filter, by using the concept of a cascade Bloom Filter. Also, the Bloomier Filter associates some values with a subset. Moreover, it introduces a meta-database with a set to keep track of the entries in the set.

Stable Bloom Filter (SBF) [13] is an extension of the Standard Bloom Filter that updates its Bloom Filter based on recent data. SBF is used for duplicate detection in streaming applications. In this Bloom Filter, each element is mapped to d slots. And each slot can have a minimum value 0 and maximum value Max where $Max = 2^d - 1$. The new element is hashed in the insertion operation, and the slots are checked for nonzero values. If the result of such a check is yes, then data is duplicated. If no, then P (a fixed number) slots are chosen randomly. Then, these selected slots are decremented by 1. And the hashed slots are assigned the value Max. Moreover, this Bloom Filter is called Stable because, after several iterations, the fraction of 0s in SBF becomes fixed irrespective of the parameters set at the beginning.

Weighted Bloom Filter (WBF) [14] uses the query frequency and the membership likelihood to increase the efficiency of Bloom Filter. In WBF, each element is hashed by a total of k hash functions, and k depends on the total number of queries and its likelihood of being a member. Moreover, the mean false positive probability is a weighted sum of the total number of queries for a particular element. In other words, an element is given more hash functions when its query frequency is high, and the likelihood of being a member is low.

Retouched Bloom Filter (RBF) [15] is an extension of Bloom Filter, which is able to remove some selected false positives. However, a tradeoff is made as it also generates some false negatives. In RBF, some slots which return false positives are selected randomly. Then, among those slots, RBF discovers the producer of a minimum number of false negatives and also maximizes the number of false positive removals; it assigns the value 0.

d-Left CBF (dlCBF) [16] is an improvement of the CBF. It uses the d-left hash table. This hash table consists of buckets, where each bucket has a fixed number of cells. Each cell is of fixed size to hold a fingerprint and a counter. This arrangement makes the hash table appear as a big array. Each element has a fingerprint. And each fingerprint has two parts. The first is a bucket index, which stores the element. The second is the remaining part of the fingerprint. The range of bucket index is $[B]$, and the remainder is $[R]$. So the hash function is $H: U \rightarrow [B] \times [R]$. During element insertion, the idea is to hash the element and store it in appropriate remainders in the cell of each bucket, and increment the counter. During deletion, the filter decrements the counter. dlCBF solves the problems that arise due to using a single hash function. The hashing operation has two phases. In the first, it applies a hash function, which gives the true fingerprint. And in the second phase, the filter finds the d locations of the element using additional (pseudo)-random permutation. One small disadvantage in the obtained d locations is that these are not independent and uniform, as the choice of the permutation determines it.

Scalable Bloom Filter (SBF) [17] is a Bloom filter having one or more Bloom Filters. In each Bloom filter, the array is partitioned into k slices. Each hash function produces one slice. During insertion operation, k hash functions produce an index in their respective slice for each element. So, each element is described using k bits. When one Bloom Filter is full, another Bloom Filter is added. During query operation, all filters are searched for the presence of that element. Each element's k bit description makes this filter more robust where no element is especially sensitive to false positives. In addition, this Bloom Filter has the advantage of scalability by adapting to set growth by adding a series of classic Bloom Filters and making the error probability tighter as per requirement.

Adaptive Bloom Filter (ABF) [18] is a Bloom Filter based on the Partial-Address Bloom Filter [19]. ABF is used in tracking the far-misses, which are those misses that hit if the core is permitted to use more cache. For each core set, a Bloom Filter array (BFA) with 2^k bits is added. When a tag is removed from the cache, the tag's k least significant bit is used to index a bit of the BFA, 1. The BFA is looked for the requested tag during cache miss, using the k least significant bit. A far-miss is detected when the array bit becomes 1.

Blocked Bloom Filter [20] is a cache-efficient Bloom Filter. It is implemented by fitting a b standard Bloom Filter sequence in each cache block/line. Usually, for better performance, the Bloom Filters are made cache-line-aligned. When an element is added, the first hash function determines the Bloom Filter block to use. The other hash functions are used for mapping of the element to k array slots, but within this block. Thus, this leads to only one cache miss. This is further improved by taking a single hash function instead of k hash functions. Hence, this single hash function determines the k slots. In addition, this hash operation is implemented using fewer SIMD instructions. The main disadvantage in using one hash function is that two elements are mapped to the same k slots, causing a collision. And this leads to an increased false positive rate (FPR).

Dynamic Bloom Filter (DBF) [21] is an extension of Bloom Filter, which changes dynamically with changing cardinality. A DBF consists of some CBF (Counting Bloom Filter), say s. Initially, s is 1, and the status of CBF is seen as active. A CBF is called active when a new element is inserted, or an element is deleted from it. During insertion

operation, DBF first checks whether the active CBF is full. If it is full, a new CBF is added, and its status is made active. If not, a new element is added to active CBF. During query operation, the response is given after searching all CBF. And during deletion operation, first, the CBF is found, which contains the element. If a single CBF contains that element, then it is deleted. However, if multiple CBFs are such, then the deletion operation is ignored, but deletion response (i.e., the operation is completed) is delivered. Furthermore, if the sum of two CBF capacities is less than a single CBF, then they are united. For that, the addition of counter vectors is used. The time complexity of insertion is the same, i.e., $O(1)$, whereas query and deletion operation is $O(k \times s)$ where k is the number of hash functions.

Deletable Bloom filter (DlBF) [22] is a Bloom Filter that enables false-negative-free deletions. In this Bloom Filter, the region of deletable bits is encoded compactly and saved in the filter memory. DlBF divides the Bloom Filter array into some regions. The regions are marked as deletable or non-deletable using a bitmap of size the same as the number of regions. During insertion operation, when an element maps to an existing element slot, i.e., collision, then the corresponding region is marked as non-deletable, i.e., the bitmap is assigned value 1. This information is used during deletion. The elements under the deletable region are only allowed to be deleted. Insertion and query operations on DBF are the same as the traditional Bloom Filter.

Index-split Bloom Filter (ISBF) [23] helps in reducing memory requirements and off-chip memory accesses. It consists of many groups of on-chip parallel CBFs and a list of off-chip items. When a set of items is stored, the index of each item is divided into some B groups. Each group contains b bits, where $B = \lceil \log_2 n/b \rceil$. So the items are split into 2^b subsets. A CBF represents each subset. Thus, a total of 2^b CBFs per group are constructed in on-chip memory. The response is given during query operation after matching the query element and the index of an item found by the B group of on-chip parallel CBFs. Also, for deletion operation, a lazy algorithm is followed. The deletion of an item requires the adjustment of indexes of other off-chip items and reconstruction of all on-chip CBFs. Moreover, the average time complexity for off-chip memory accesses for insertion, query, and deletion is $O(1)$.

Forest-structured Bloom Filter (FBF) [24] is designed to filter out duplicate data in large-scale data. FBF is a hierarchical Bloom Filter that combines the RAM and SSD (NAND Flash) for extreme scalability. Random read and random write operations in RAM are inexpensive. However, a random write operation of SSD is expensive since past data is erased before being written to the SSD. Thus, FBF adopts the lazy update policy upon insertion or deletion. However, read operation is allowed instantly to access information from SSD. Similarly, BloomFlash [25] and BloomStore [26] adapt the same principle.

Name Filter [27] is a two-tier filter that helps in looking up names in Named Data Networking. The first tier determines the length of the name prefix, and the second tier uses the prefix determined in the previous stage to search for a group of Bloom Filters. In the first stage, the name prefixes are mapped to Bloom Filter. After that, the process of building up a Counting Bloom Filter is taken up. This filter is built for the concerned prefix set, and then it is converted to take the form of a conventional Boolean Bloom Filter. As a final step, the second stage uses the merged Bloom Filter. In the first stage, the calibration of the name prefixes to the Bloom Filter is done on the basis of their lengths. It maps the k hash function into a single word. Hence, the Bloom Filter is called One Memory Access Bloom Filter as the query access time is $O(1)$ instead of $O(k)$. First, it acquires the hash output of the prefix using the DJB hash method. Then, the later hash value is calculated using the previous hash value. Thus, after $k-1$ loops, it obtains a single hash value and stores it in a word. This value is input for the calculation of the address in Bloom Filter, and the other bits are calculated from one AND operation. So, when $k-1$ bits are 1s, a graceful identification is declared. This stage aims to find the longest prefix. In the second stage, the prefixes are divided into groups based on their associated next-hop port(s). All groups are stored in the Bloom Filter. And the desired port is found in this stage. In MBF, each slot stores a bit string with machine word-aligned. The Nth bit stores the Nth Bloom Filter's hash value and the other bits are padded with 0s. To obtain the forwarding port number, AND operation is done on K-bit strings with respect to k hash functions. The location of 1 in the result gives the port number.

Rottenstreich et al. [28] devised a novel approach to reduce false positives in counting Bloom Filter, called VI-CBF. Conventional counting Bloom Filter increments the counter upon insertion of an item by 1 and decrements the counter upon deletion of an item by 1. On the contrary, VI-CBF uses the B_h sequence for incrementing or decrementing the counter value. VI-CBF increments the counter by B_h number in the insertion of an item. Similarly, VI-CBF decrement the counter by B_h number in the removal of an item. For query, the conventional counting Bloom Filter gives a response as positive if the counter is greater than zero. However, the VI-CBF response is positive if the counter value is greater than zero and B_h sequence is supports the conclusion. Thus, VI-CBF dramatically reduces the false positive rate.

Spatial Bloom filter (SpatialBF) [29] is a Bloom Filter variant for multiple sets. Let the number of input item sets be s. SpatialBF array consists of m slots, which store numbers up to s, i.e., the number of input item sets. The number of

TABLE 7.1 Comparison of various parameters of Bloom Filter variants. FP is false positive, FN is false negative.

Bloom Filter	Year	Architecture	Memory Allocation	Implementation	Platform	FP	FN	Scalability
Fan et al. [10]	2000	Counting	Static	Flat	RAM	✓	×	High
Spectral [11]	2003	Counting	Static	Flat	RAM	✓	×	High
Bloomier Filter [12]	2004	Non-counting	Static	Chain	RAM	✓	✓	Very high
Stable [13]	2006	Non-counting	Static	Flat	RAM	✓	✓	High
Weighted [14]	2006	Non-counting	Non-counting	Flat	RAM	✓	✓	Low
Retouch [15]	2006	Non-counting	Non-counting	Flat	RAM	✓	✓	Low
d-Left CBF [16]	2006	Counting	Static	Flat	RAM	✓	×	Medium
Scalable [17]	2007	Non-counting	Dynamic	Block	RAM	✓	✓	High
Adaptable [18]	2008	Non-counting	Static	Flat	Cache	✓	✓	Low
Blocked [20]	2009	Non-counting	Static	Block	Cache	✓	✓	Medium
Dynamic [21]	2010	Counting	Dynamic	Chain	RAM	✓	×	Very High
Deletable [22]	2010	Non-counting	Static	Block	RAM	✓	×	Medium
Index-split [23]	2011	Non-counting	Dynamic	Block	RAM, HDD	✓	✓	High
Forest-structured [24]	2011	Non-counting	Dynamic	Hierarchical	RAM, SSD	✓	✓	Very High
BloomFlash [24]	2011	Non-counting	Dynamic	Hierarchical	RAM, SSD	✓	✓	Very High
BloomStore [24]	2011	Non-counting	Dynamic	Hierarchical	RAM, SSD	✓	✓	Very High
Quotient Filter [30]	2012	Non-counting	Static	Block	RAM	✓	✓	Low
Name Filter [27]	2013	Both	Static	Hierarchical	RAM	✓	✓	High
Variable-increment [28]	2014	Counting	Static	Flat	RAM	✓	×	Medium
Bloofi [31]	2015	Non-counting	Dynamic	Hierarchical	RAM	✓	✓	High
Sliding [32]	2015	Non-counting	Static	Flat	RAM	✓	✓	Medium
BF Trie [33]	2016	Non-counting	Dynamic	Hierarchical	RAM	✓	✓	High
Autoscaling [34]	2017	Counting	Dynamic	Flat	RAM	✓	×	High
Ternary [35]	2017	Counting	Static	Flat	RAM	✓	×	Low
Difference [36]	2017	Non-counting	Dynamic	Chain	SRAM, DRAM	✓	✓	High
TinySet [37]	2017	Non-counting	Dynamic	Chain	RAM	✓	✓	High
BloomFlow [38]	2017	Non-counting	Dynamic	Hierarchical	RAM	✓	✓	High
Dynamic Reordering [39]	2017	Non-counting	Dynamic	Flat	RAM	✓	✓	Medium
ScaleBF [40]	2018	Non-counting	Static	3D	RAM	✓	×	Very high
rDBF [41]	2019	Non-counting	Static	Multi-dimensional	RAM	✓	×	high
Spatial [29]	2015	Non-Counting	Static	Flat	RAM	✓	✓	No

hash functions is k. Initially, the array is set to value 0. During insertion operation, first, the items of s_1 are inserted. Then k hash functions hash the item, and the hash value gives the array location. The k locations are set to 1. Then items of s_2 are inserted, where the k locations are set to 2. Similarly, for insertion of items of s_i, set the k locations to i where $i > 0$. In case of collision, the slot stores the maximum value. During query operation, the item of s_i is hashed to obtain k locations. The item is present if the value of the k slots is greater or equal to i and one of the slot values is i. The memory requirement of SpatialBF is $(\lfloor \log_2 s \rfloor)m$ bits.

Crainiceanu et al. proposed a Bloom Filter called Bloofi [31], which is a Bloom Filter index. It is implemented like a tree. The Bloom Filter tree construction is done as follows. The leaves are Bloom Filters. And the bitwise OR on the leaf Bloom Filters is done to obtain the parent nodes. This process continues till root is obtained. The element is checked at the root; if it does not match, Bloofi returns $False$, because if an element in a leaf does not match, then it will not match from the leaf to the root, whereas if the element matches, the query moves to root's children Bloom Filters until it reaches the leaf. During insertion of a new node, the search for the most similar node to the new node is done. Bloofi wants to keep similar nodes together. So, when found, this new node is inserted as its sibling. If an overflow occurs, the same procedure is followed as in a B^+ tree. During a deletion operation, the parent node deletes the pointer to the node. And when underflow occurs, the same procedure is followed as in B^+ tree.

Sliding Bloom Filter [32] is a Bloom Filter having a sliding window. It has parameters (n, m, ε). The sliding window remains over the last n elements, and the value of the slots is 1. In other words, the window only shows the elements that are present. The m elements that appear before the window elements do not have restrictions on the value. And ε is the most probable slot, and it is 1. This Bloom Filter is dictionary-based and uses the Backyard Cuckoo hashing [42]. To this hashing, a similar lazy deletion method is applied as used by Thorup [43]. A parameter c is used, which is the trade-off between the accuracy of the index stored and the number of elements stored in the dictionary. After

optimizing the parameter c, the sliding Bloom Filter shows good time and space complexity. The algorithm uses a hash function selected from the family of universal hash functions. For each element in the dictionary, D stores its hash value and location where it previously appeared. The data stream is divided into generations of size n/c each, where c is optimized later. Generation 1 is the first n/c elements, generation 2 is next n/c elements, and so on. Current window contains the last n elements and at most $c+1$ generations. Two counters are used, one for generation number (say, g) and the other for the current element in the generation (say, i). For every increment of i, g get incremented to mod $(c+1)$. For insertion, first, obtains the ith hash value and checks whether it is present in D; if it exists, the element's location is updated with the current generation. Otherwise, it stores the hash value and generation number. Finally, the algorithm updates the two counters. If g changes, it scans D and deletes all elements with associated data equal to the new value of g.

Bloom Filter Trie (BFT) [33] helps to store and compress a set of colored k-mers, and efficient traversal of the graph. It is an implementation of the colored de Bruijn graph (C-DBG). It is based on a burst tree, which stores k-mers along with the set of colors. Colors form a bit array initialized with 0. A slot assigns the value 1 if that index k-mers have that color. Later, this set of colors is compressed. BFT is defined as $t=(V_t, E_t)$ having the maximum height as k where the k-mers are split into k substrings. A BFT is a list of compressed containers. An uncompressed container of a vertex V is defined as $\langle s, color_{ps}\rangle$ where s is the suffix and p is the prefix that represents the path from the root to V. Tuples are ordered lexicographically based on their suffixes. BST supports operations for storing, traversing, and searching a pan-genome. And it also helps in extracting relevant information of the contained genomes and subsets. The time complexity for insertion of a k-mer is $O(d+2^{\lambda}+2q)$ where d is the worst lookup time, λ is the number of bits to represent the prefix, and q is the maximum number of children. And the time complexity of the lookup operation is $O(2^{\lambda}+q)$.

Autoscaling Bloom Filter [34] is a generalization of CBF, which allows adjustment of its capacity based on probabilistic bounds on false positives and true positives. It is constructed by binarization of the CBF. The construction of the Standard Bloom Filter is done by assigning all nonzero positions of the CBF as 1. And, given a CBF, the construction of ABF is done by assigning all the values which are less than or equal to the threshold value as 0.

Ternary Bloom Filter (TBF) [35] is a counting Bloom Filter to address the false positives and false negatives. TBF uses three values $\{0,\ 1,\ X\}$. The value 0 signifies the absence of a key, the value 1 signifies the presence of a key, and X signifies that the bit is shared. The shared bit cannot be deleted, and hence, TBF calls X as indeterminable and non-deletable; X cannot be used in querying an item, too. We claim that TBF requires at least two bits to represent the value field. Therefore, the bits are wasted to remove false positives and false negatives. In a very large amount of input, most of the bit fields of TBF are field by value X. Hence, practically TBF is not feasible. Let us assume that the values are represented by {00}, {01}, {10}, and {11}. The value {00} represents 0 in TBF, {01} represents 1 in TBF. Either {10} or {11} represents X. Clearly, a count is wasted in the binary representation. If most of the counter represents value X, then TBF is of no use. Moreover, an increment or decrement requires binary bit insertion in a practical implementation scenario. On the contrary, if TBF uses a character/integer array, then at least one byte is wasted to represent the three values. Thus, it consumes more space than expected.

Difference Bloom Filter (DBF) [36] is a probabilistic data structure based on Bloom Filter. It has a multi-set membership query that is more accurate and has a faster response speed. It is based on two main design principles. First, an item is reported positive in a Bloom Filter; then, no other Bloom Filter reports positive. Second is the use of DRAM memory to increase the accuracy of the filter. DBF consists of an SRAM and a DRAM chaining hash table. The SRAM filter is an array of m bits with k independent hash functions. During the insertion function, elements in the set i are mapped to the k bit of the filter. Arbitrary $k-i+1$ bits are set to value 1 and the other $i-1$ bits are set to value 0. This is called the $\langle i,\ k\rangle$ constraint. DBF uses a dual-flip strategy to make this bit shared if the new element gets conflicted with another element in the filter. Dual-flip is to change a series of mapping bits of the filters so that the filter satisfies the $\langle i,\ k\rangle$ constraint. During the lookup operation, if exactly $k-i+1$ bits are 1, it returns $True$. During a deletion operation, for each bit of the k bits of an element, DBF decides whether to reset it or not with the help of the DRAM table.

BloomFlow [38] is a multi-stage Bloom Filter that is used for multi-casting in software-defined networking (SDN). It helps to achieve reductions in forwarding state while avoiding false positive packet delivery. The BloomFlow extends the OpenFlow [44] forward action with a new virtual port called BLOOM_PORTS to implement Bloom Filter forwarding. An algorithm is implemented when a flow specifies an output action to BLOOM_PORTS Forwarding Element (FE). The algorithm first reads from the start of the IP option field the Elias gamma encoded filter length b and the number of hash functions of k fields. Then the algorithm treats the rest of the bits of the IP option field as a Bloom Filter. And this Bloom Filter is copied to a temporary cache for further processing. The remainder of

the IP option fields and the IP payload are left to remove the first stage filter from the packet header. Then the algorithm iterates through all interfaces and checks for membership test for each interface's Bloom identifier in the cached Bloom Filter. Bloom identifier is a unique, 16-bit integer identifier. The network controller assigns the Bloom identifier to every interface on the network that participates in multi-cast forwarding. The packet is forwarded from the matched interface if the membership test returns $True$.

Dynamic Reordering Bloom Filter [39] is another type of Bloom Filter that reduces the searching cost. It dynamically reorders the searching sequence of multiple Bloom Filters using One Memory Access Bloom Filter (OMABF), and the order of checking is saved in Query Index (QI). This approach considers two factors. First is the policy of changing the query priority of Bloom Filter. Second is the reduction of overhead due to a change in the order. This approach reduces the searching time of the query by sorting and saving the query data in Bloom Filter based on popularity. Sorting is done based on the query order, i.e., the popularity of data. So, when the request comes from that data, it quickly gives the response. And, when the popularity of a data becomes more, its query order is made a level higher in the Bloom Filter. However, this change of query order imposes overheads. To solve this, Query Index (QI) is used. QI saves the query priority of each block. Bloom Filter is checked according to the order saved in QI when membership is checked.

Patgiri et al. [41] presented a pure multidimensional Bloom Filter to reduce false positives. rDBF design philosophy is based on a multidimensional array to get rid of dependence on the number of hash functions. rDBF is independent of a number of hash functions. Also, rDBF does not depend on input size. An input item occupies at most one bit. Thus, memory consumption is very low. However, rDBF does not readjust the filter if the filter becomes full to exploit the advantages of static Bloom Filter. rDBF uses the Murmur hash function to improve the performance. The Murmur hash function hashes an input item. The returned value is placed in the filter using modulus and bitwise operations. Therefore, prime numbers are required to perform the modulus operation to distribute the input item evenly among all bits. Thus, the dimensions of the rDBF are always prime numbers.

ScaleBF [40] is highly-scalable Bloom Filter based on 3D Bloom Filter which is the extended implementation of rDBF [41]. scaleBF relies on a chain hash data structure to enhance the lookup and insertion performance. Even though scaleBF allocates memory dynamically, the memory allocation strategy is static because scaleBF does not allow altering once memory is allocated. However, scaleBF allows growing or shrinking the allocated memory size using chain hash data structures.

7.3 Other approximation filters

Cuckoo Bloom Filter [45] is based on Cuckoo hash table [46]. This Bloom Filter stores fingerprints instead of key–value pairs. At the same time, a fingerprint means the hash value of the element. For insertion, the index for two candidate buckets is calculated. One is the hash value of the element, and the other is the result of XOR operation between the hash value of the element and the hash value of the fingerprint of that element. This is called partial-key Cuckoo hashing. This method reduces a hash collision and improves table utilization. After finding the indexes, the element is stored in any free bucket. Otherwise, Cuckoo hash tables' kicking [46] of elements is done. For query operation, two candidate buckets are calculated as done in an insertion operation, then if the element is present in any one of them, $True$ is returned, otherwise one gets $False$. For a deletion operation, the same procedure as for lookup is followed, except that instead of returning $True$ or $False$, the element is deleted. The advantage of the basic algorithms (i.e., insertion, deletion, and lookup) is that they are independent of hash table configuration (e.g., number of entries in each bucket). However, the disadvantage of using partial-key Cuckoo hashing for storing fingerprints leads to a slow increase in fingerprint size to an increase in filter size. In addition, if the hash table is huge but stores short fingerprints, then the chance for hash collision increases. This leads to possible insertion failure and reduces table occupancy.

TinySet [37] is a Bloom Filter that has more space efficiency compared to a standard Bloom Filter. Its structure is similar to that of the blocked Bloom filter, but each block is a chain-based hash table [47]. It uses a single hash function, $H \rightarrow B \times L \times R$, where B is the block number, L is the index of the chain within that block, and R is the remainder (or fingerprint) that is stored in that block. All operations (insertion, deletion, and lookup) initially follow three common steps. First, the algorithm applies the hash function to the element and obtains the B, L, and R values. Second, it uses B to access the specific block. Third, it calculates the Logical Chain Offset (LCO) and Actual Chain Offset (ACO) values. During an insertion operation, the algorithm shifts all fingerprints from offset to the end of the block to the right. The first bits in a block contain a fixed size index I. Unset I means the chain is empty. If the I

bit is unset, it is made to set, and the new element is marked as the last of its chain. If I is unset during a deletion operation, the operation is terminated as it indicates the element is absent. Otherwise, the algorithm shifts all bits from the ACO value to the end of the block to the left by a single bit. If the deleted element is marked last, then the previous is made last or the algorithm marks the entire chain as empty. If I is unset, similarly as in the lookup operation, the operation is terminated. Otherwise, the algorithm searches the chain. TinySet is more flexible due to its ability to dynamically change its configuration as per the actual load. It accesses only a single memory word and partially supports the deletion of elements. However, the delete operation gradually degrades its space efficiency over time.

Quotient Filter (QF) [30] is a Bloom Filter where each element is represented by a multi-set F. The F is an open hash table with a total bucket of $m = 2^q$, called quotienting [48]. Besides, F stores a p-bit fingerprint for each element which is the hash value. In this technique, a fingerprint f is partitioned into r least significant bits, which store the remainder. The $q = p - r$ is the most significant bit which stores the quotient. Both quotient and remainder are used for the reconstruction of the full fingerprint. During insertion operation, F stores the hash value. During query operation, F is searched for the presence of the hash value of the element. And, during a deletion operation, the hash value of that element is removed from F. QF has the advantage of dynamical resizing, i.e., it expands and shrinks as elements are added or deleted. However, the QF insertion throughput deteriorates towards maximum occupancy.

7.4 Conclusion

In this chapter, we have discussed various kinds of Bloom Filter that evolve based on the applications' requirements. Also, various variants were proposed to improve the performance of the Bloom Filter. For instance, counting Bloom Filter was introduced to counter false negative issues. Moreover, the hierarchical Bloom Filter was introduced to filter the massive amount of data. Also, fingerprints were introduced to reduce the number of false positives. Therefore, various Bloom Filters were introduced to address the issues with the Bloom Filter. Multidimensional Bloom Filter was introduced to reduce the number of false positives. Also, this variant of Bloom Filter enhances the performance.

References

[1] S. Geravand, M. Ahmadi, Bloom filter applications in network security: a state-of-the-art survey, Comput. Netw. 57 (18) (2013) 4047–4064, https://doi.org/10.1016/j.comnet.2013.09.003.

[2] P. Xiao, Z. Li, H. Qi, W. Qu, H. Yu, An efficient DDoS detection with Bloom filter in SDN, in: 2016 IEEE Trustcom/BigDataSE/ISPA, 2016, pp. 1–6.

[3] K. Sasaki, A. Nakao, Packet cache network function for peer-to-peer traffic management with Bloom-filter-based flow classification, in: 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS), 2016, pp. 1–6.

[4] S.D. Jackman, B.P. Vandervalk, H. Mohamadi, J. Chu, S. Yeo, S.A. Hammond, G. Jahesh, H. Khan, L. Coombe, R.L. Warren, et al., ABySS 2.0: resource-efficient assembly of large genomes using a Bloom filter, Genome Res. 27 (5) (2017) 768–777.

[5] P. Melsted, J.K. Pritchard, Efficient counting of k-mers in DNA sequences using a Bloom filter, BMC Bioinform. 12 (1) (2011) 333, https://doi.org/10.1186/1471-2105-12-333.

[6] J. Bringer, C. Morel, C. Rathgeb, Security analysis and improvement of some biometric protected templates based on Bloom filters, Image Vis. Comput. 58 (Supplement C) (2017) 239–253, https://doi.org/10.1016/j.imavis.2016.08.002.

[7] C. Rathgeb, F. Breitinger, C. Busch, H. Baier, On application of Bloom filters to iris biometrics, IET Biometrics 3 (4) (2014) 207–218, https://doi.org/10.1049/iet-bmt.2013.0049.

[8] D. Sadhya, S.K. Singh, Providing robust security measures to Bloom filter based biometric template protection schemes, Comput. Secur. 67 (Supplement C) (2017) 59–72, https://doi.org/10.1016/j.cose.2017.02.013.

[9] A. Singh, S. Garg, S. Batra, N. Kumar, J.J. Rodrigues, Bloom filter based optimization scheme for massive data handling in IoT environment, Future Gener. Comput. Syst. 82 (2018) 440–449, https://doi.org/10.1016/j.future.2017.12.016.

[10] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area web cache sharing protocol, IEEE/ACM Trans. Netw. 8 (3) (2000) 281–293, https://doi.org/10.1109/90.851975.

[11] S. Cohen, Y. Matias, Spectral Bloom filters, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2003, pp. 241–252.

[12] B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, The Bloomier filter: an efficient data structure for static support lookup tables, in: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004, pp. 30–39.

[13] F. Deng, D. Rafiei, Approximately detecting duplicates for streaming data using stable Bloom filters, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2006, pp. 25–36.

[14] J. Bruck, J. Gao, A. Jiang, Weighted Bloom filter, in: 2006 IEEE International Symposium on Information Theory, 2006.
[15] B. Donnet, B. Baynat, T. Friedman, Retouched Bloom filters: allowing networked applications to trade off selected false positives against false negatives, in: Proceedings of the 2006 ACM CoNEXT Conference, CoNEXT '06, ACM, New York, NY, USA, 2006, pp. 13:1–13:12.
[16] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, G. Varghese, An Improved Construction for Counting Bloom Filters, Springer, Berlin, Heidelberg, 2006, pp. 684–695.
[17] P.S. Almeida, C. Baquero, N. Preguiça, D. Hutchison, Scalable Bloom filters, Inf. Process. Lett. 101 (6) (2007) 255–261.
[18] K. Nikas, M. Horsnell, J. Garside, An adaptive Bloom filter cache partitioning scheme for multicore architectures, in: 2008 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, 2008, pp. 25–32.
[19] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, K. Lai, Bloom filtering cache misses for accurate data speculation and prefetching, in: ACM International Conference on Supercomputing 25th Anniversary Volume, ACM, 2014, pp. 347–356.
[20] F. Putze, P. Sanders, J. Singler, Cache-, hash-, and space-efficient Bloom filters, J. Exp. Algorithmics 14 (2010) 4:4.4–4:4.18, https://doi.org/10.1145/1498698.1594230.
[21] D. Guo, J. Wu, H. Chen, Y. Yuan, X. Luo, The dynamic Bloom filters, IEEE Trans. Knowl. Data Eng. 22 (1) (2010) 120–133, https://doi.org/10.1109/TKDE.2009.57.
[22] C.E. Rothenberg, C.A.B. Macapuna, F.L. Verdi, M.F. Magalhaes, The deletable Bloom filter: a new member of the Bloom family, IEEE Commun. Lett. 14 (6) (2010) 557–559, https://doi.org/10.1109/LCOMM.2010.06.100344.
[23] K. Huang, D. Zhang, An index-split Bloom filter for deep packet inspection, Sci. China Inf. Sci. 54 (1) (2011) 23–37, https://doi.org/10.1007/s11432-010-4132-4.
[24] G. Lu, B. Debnath, D.H.C. Du, A forest-structured Bloom filter with flash memory, in: 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST), 2011, pp. 1–6.
[25] B. Debnath, S. Sengupta, J. Li, D.J. Lilja, D.H.C. Du, Bloomflash: Bloom filter on flash-based storage, in: 2011 31st International Conference on Distributed Computing Systems, 2011, pp. 635–644.
[26] G. Lu, Y.J. Nam, D.H.C. Du, Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash, in: 012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), 2012, pp. 1–11.
[27] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, Q. Dong, Namefilter: achieving fast name lookup with low memory cost via applying two-stage Bloom filters, in: 2013 Proceedings IEEE INFOCOM, 2013, pp. 95–99.
[28] O. Rottenstreich, Y. Kanizo, I. Keslassy, The variable-increment counting Bloom Filter, IEEE/ACM Trans. Netw. 22 (4) (2014) 1092–1105, https://doi.org/10.1109/TNET.2013.2272604.
[29] L. Calderoni, P. Palmieri, D. Maio, Location privacy without mutual trust: the spatial bloom filter, in: Security and Privacy in Unified Communications: Challenges and Solutions, Comput. Commun. 68 (2015) 4–16, https://doi.org/10.1016/j.comcom.2015.06.011.
[30] M.A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B.C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R.P. Spillane, E. Zadok, Don't thrash: how to cache your hash on flash, Proc. VLDB Endow. 5 (11) (2012) 1627–1637, https://doi.org/10.14778/2350229.2350275.
[31] A. Crainiceanu, D. Lemire, Bloofi: multidimensional Bloom filters, Inf. Syst. 54 (Supplement C) (2015) 311–324, https://doi.org/10.1016/j.is.2015.01.002.
[32] M. Naor, E. Yogev, Tight bounds for sliding Bloom filters, Algorithmica 73 (4) (2015) 652–672, https://doi.org/10.1007/s00453-015-0007-9.
[33] G. Holley, R. Wittler, J. Stoye, Bloom filter tree: an alignment-free and reference-free data structure for pan-genome storage, Algorithms Mol. Biol. 11 (2016), https://doi.org/10.1186/s13015-016-0066-8.
[34] D. Kleyko, A. Rahimi, E. Osipov, Autoscaling Bloom filter: controlling trade-off between true and false positives, CoRR, arXiv:1705.03934, arXiv:1705.03934, http://arxiv.org/abs/1705.03934.
[35] H. Lim, J. Lee, H. Byun, C. Yim, Ternary Bloom filter replacing counting Bloom filter, IEEE Commun. Lett. 21 (2) (2017) 278–281, https://doi.org/10.1109/LCOMM.2016.2624286.
[36] D. Yang, D. Tian, J. Gong, S. Gao, T. Yang, X. Li, Difference Bloom filter: a probabilistic structure for multi-set membership query, in: 2017 IEEE International Conference on Communications (ICC), 2017, pp. 1–6.
[37] G. Einziger, R. Friedman, Tinyset – an access-efficient self-adjusting Bloom filter construction, IEEE/ACM Trans. Netw. 25 (4) (2017) 2295–2307, https://doi.org/10.1109/TNET.2017.2685530.
[38] A. Craig, B. Nandy, I. Lambadaris, P. Koutsakis, Bloomflow: openflow extensions for memory efficient, scalable multicast with multi-stage Bloom filters, Comput. Commun. 110 (Supplement C) (2017) 83–102, https://doi.org/10.1016/j.comcom.2017.05.018.
[39] D.C. Chang, C. Chen, M. Thanavel, Dynamic reordering Bloom filter, in: 2017 19th Asia-Pacific Network Operations and Management Symposium (APNOMS), 2017, pp. 288–291.
[40] R. Patgiri, S. Nayak, S.K. Borgohain, scaleBF: a high scalable membership filter using 3D Bloom filter, Int. J. Adv. Comput. Sci. Appl. 9 (12) (2018), https://doi.org/10.14569/IJACSA.2018.091277.
[41] R. Patgiri, S. Nayak, S.K. Borgohain, rDBF: an r-dimensional Bloom filter for massive scale membership query, J. Netw. Comput. Appl. 136 (2019) 100–113, https://doi.org/10.1016/j.jnca.2019.03.004.
[42] Y. Arbitman, M. Naor, G. Segev, Backyard Cuckoo hashing: constant worst-case operations with a succinct representation, in: 2010 IEEE 51st Annual Symposium on Foundations of Computer Science, 2010, pp. 787–796.
[43] M. Thorup, Timeouts with time-reversed linear probing, in: Infocom, 2011 Proceedings Ieee, IEEE, 2011, pp. 166–170.
[44] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, Openflow: enabling innovation in campus networks, ACM SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74.
[45] B. Fan, D.G. Andersen, M. Kaminsky, M.D. Mitzenmacher, Cuckoo filter: practically better than Bloom, in: Proceedings of the 10th ACM Intl. Conf. on Emerging Networking Experiments and Technologies, CoNEXT '14, 2014, pp. 75–88.
[46] R. Pagh, F.F. Rodler, Cuckoo hashing, J. Algorithms 51 (2) (2004) 122–144.
[47] R.L. Rivest, C.E. Leiserson, Introduction to Algorithms, McGraw-Hill, Inc., 1990.
[48] D.E. Knuth, The Art of Computer Programming: Sorting and Searching, vol. 3, Pearson Education, 1998.

CHAPTER

8

Counting Bloom Filter: architecture and applications

8.1 Introduction

Standard Bloom Filter is a simple and very efficient data structure. It is widely used for deduplication and filtering, such as networking and data streaming. However, it does not permit a delete operation. Because the delete operation introduces false negatives into the standard Bloom Filter, it leads to an increase in FPP. As standard Bloom Filter does not permit the delete operation, eventually, it leads to saturation of the data structure. Then saturated standard Bloom Filter gives false positive responses. To solve this issue, counting Bloom Filter (CBF) is proposed.

CBF has a simple architecture similar to the standard Bloom Filter. A conventional CBF is a typical Bloom Filter, with each slot having an attached counter. The counter value is incremented upon insertion of a duplicate item. However, there is a probability that a counter is increasing its value for inserting a non-duplicate item. On the other hand, the counter value is decremented upon the deletion of an item from the CBF. CBF successfully implements a delete operation without introducing false negatives. But CBF consumes four times more memory compared to Bloom Filter. Hence, this Bloom Filter variant loses the space complexity advantage. Another major issue with CBF is the counter overflow. In conventional CBF, the counter is considered 4 bits long [1]. The length is considered long enough (mostly in networking applications). But in today's scenario, the flow of data has reached one million bits per second. So, the counter capacity of $2^4 - 1$, i.e., 15, is highly insufficient. Moreover, other issues with a CBF are due to false positives, false negatives, scalability, etc.

Many CBF variants are proposed to solve these issues [2]. The main focus of the CBF variant is to reduce memory consumption and FPP. Many variants also modified the counter implementation to give the frequency of the elements. CBF is implemented mostly in the field of networking. It is used in finding elephant flows [3], icebergs [4], DDoS attacks [5]. It is also widely used in hardware. It is used in the cache to reduce cache access. CBF is also used in many other fields. Parts II and III of this book explore the CBF applications in various domains in detail.

This chapter is organized as follows: Section 8.2 presents the detailed architecture of CBF. Section 8.3 surveys various variants of CBFs. Section 8.4 analyzes issues of CBF. Finally, this chapter is concluded in Section 8.5.

8.2 Counting Bloom Filter

8.2.1 Architecture

The architecture of CBF is very similar to that of the standard Bloom Filter. Fig. 8.1 represents the architecture of CBF, which is an array of size m, with each slot of size $c + 1$. Each slot is partitioned into a single bit (say, data bit) for data and a counter of length c. The data bit is assigned to either 0 or 1. The counter counts the number of elements hashed to that slot. The number of hash functions is K. Initially, the array is initialized to 0. Both data bit and counter are initialized to 0. The CBF is explained based on the CBF proposed by Lim Fan et al. [1]. It is discussed later in this section.

8.2.2 Operations of CBF

Unlike standard Bloom Filter, CBF supports the deletion operation. Moreover, it does not affect the false negative probability. Therefore, CBF supports three operations: insert, lookup, and delete.

Bloom Filter https://doi.org/10.1016/B978-0-12-823520-1.00013-4

Copyright © 2023 Elsevier Inc. All rights reserved.

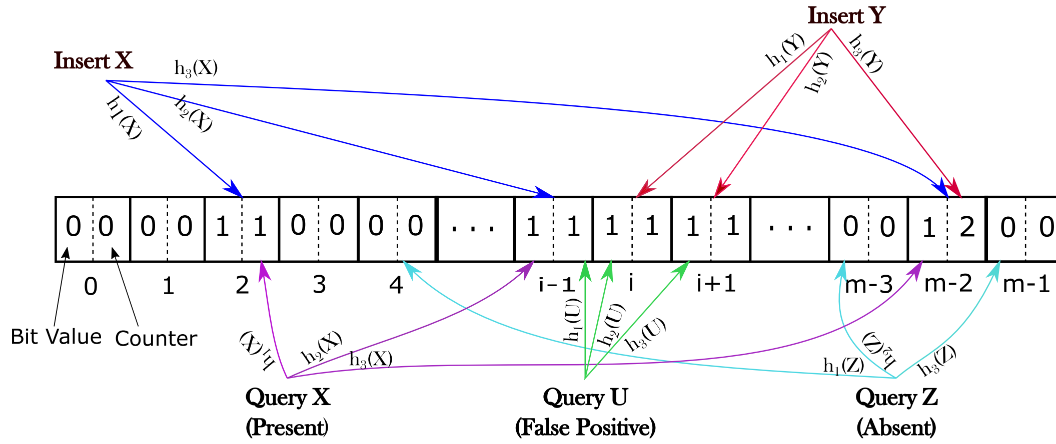


FIGURE 8.1 Architecture of Counting Bloom Filter with a 4-bit counter.

Algorithm 8.1 Insertion of an input item κ into Counting Bloom Filter $\mathbb{CBF}$ using three hash functions.

1: **procedure** INSERT($\mathbb{CBF}[]$, κ, S_1, S_2, S_3) ▷ Seeds are used to create different hash functions
2: $i_1 = \text{MURMUR}(\kappa, \, length, \, S_1) \;\%\; \mu$ ▷ μ is the size of $\mathbb{CBF}[]$ which is a prime number.
3: $i_2 = \text{MURMUR}(\kappa, \, length, \, S_2) \;\%\; \mu$ ▷ % is modulus operator
4: $i_3 = \text{MURMUR}(\kappa, \, length, \, S_3) \;\%\; \mu$ ▷ i is index of the bit array $\mathbb{CBF}[]$
5: **if** $\mathbb{CBF}[i_{1D}] == 1$ **then** ▷ i_{1D} is data bit of location i_1
6: $\mathbb{CBF}[i_{1C}] \leftarrow \mathbb{CBF}[i_{1C}] + 1$ ▷ i_{1C} is counter of location i_1, Counter value is incremented
7: **else** ▷ First time insertion to the slot
8: $\mathbb{CBF}[i_{1D}] \leftarrow 1$
9: $\mathbb{CBF}[i_{1C}] \leftarrow 1$
10: **end if**
11: **if** $\mathbb{CBF}[i_{2D}] == 1$ **then**
12: $\mathbb{CBF}[i_{2C}] \leftarrow \mathbb{CBF}[i_{2C}] + 1$
13: **else**
14: $\mathbb{CBF}[i_{2D}] \leftarrow 1$
15: $\mathbb{CBF}[i_{2C}] \leftarrow 1$
16: **end if**
17: **if** $\mathbb{CBF}[i_{3D}] == 1$ **then**
18: $\mathbb{CBF}[i_{3C}] \leftarrow \mathbb{CBF}[i_{3C}] + 1$
19: **else**
20: $\mathbb{CBF}[i_{3D}] \leftarrow 1$
21: $\mathbb{CBF}[i_{3C}] \leftarrow 1$
22: **end if**
23: **end procedure**

8.2.2.1 Insertion

When an input item is inserted into CBF, then the input item is hashed to obtain K locations. For the first time, when a slot is selected, both the data bit and counter are set to 1. Subsequently, only the counter value is incremented when input items are hashed to those slots. Algorithm 8.1 presents the algorithm for insertion operation of CBF. The CBF in Fig. 8.1 is initially empty. In this CBF, the number of hash functions K is 3. The hash functions are $\mathcal{H}_1()$, $\mathcal{H}_2()$, and $\mathcal{H}_3()$. Suppose input item X is given as input to the CBF. This X is hashed by $\mathcal{H}_1()$, $\mathcal{H}_2()$, and $\mathcal{H}_3()$, and three slot locations are obtained. As shown in Fig. 8.1, both data bit and counter are assigned value 1. Next, input item Y is inserted into CBF. This Y is hashed, and three locations are obtained. One out of the three locations obtained is common with those obtained in the case of X. The two uncommon slots are assigned value 1, i.e., both data bit and

counter are assigned value 1. In the common location, the data bit is already assigned to 1; hence, only the counter is incremented. Now, the counter value is 2.

8.2.2.2 Lookup

The lookup operation is the same as in the standard Bloom Filter. When an item is queried, the item is first hashed to obtain K slot locations. Then, those location's data bits are checked. If all the data bit values are 1, then the item is present. If at least one slot has a data bit value of 0, then the item is absent. Algorithm 8.2 presents the algorithm for the lookup operation of CBF. Consider the CBF in Fig. 8.1 with all the conditions assumed as in the insertion operation section. Moreover, suppose the CBF contains items X and Y, i.e., we consider that the insertion operations explained in the above section are performed. Suppose a lookup operation for X is requested. This X is hashed by $\mathcal{H}_1()$, $\mathcal{H}_2()$, and $\mathcal{H}_3()$ and three slot locations are obtained. The data bit of those locations is checked. All data bits are 1, hence element X is present. As assumed earlier, item X is inserted into the CBF. Therefore, the response given by CBF is a true positive. Again, consider another lookup operation for item Q. The three locations returned by $\mathcal{H}_1()$, $\mathcal{H}_2()$, and $\mathcal{H}_3()$ are checked. The location returned by $\mathcal{H}_2()$ is 1, whereas the other two location's data bits are 0. As at least one data bit is 0, item Z is absent. For this response, item Z is not inserted to CBF, and CBF also returned $False$. Therefore, the response is a true negative. Suppose another lookup request is performed for item Q. The data bit of three locations returned by $\mathcal{H}_1()$, $\mathcal{H}_2()$, and $\mathcal{H}_3()$ is 1. It means CBF returned $True$. However, item Q is not inserted into CBF. Therefore, this response is called a false positive.

Algorithm 8.2 Lookup an item κ in Counting Bloom Filter $\mathbb{CBF}$ using three hash functions.

1: **procedure** LOOKUP($\mathbb{CBF}[]$, κ, S_1, S_2, S_3)
2: $i_1 = \text{MURMUR}(\kappa,\ length,\ S_1)\ \%\ \mu$
3: $i_2 = \text{MURMUR}(\kappa,\ length,\ S_2)\ \%\ \mu$
4: $i_3 = \text{MURMUR}(\kappa,\ length,\ S_3)\ \%\ \mu$
5: **if** $\mathbb{CBF}[i_{1D}]$ AND $\mathbb{CBF}[i_{2D}]$ AND $\mathbb{CBF}[i_{3D}]$ **then** ▷ AND-ing operation is perform to check the data bit values of $\mathbb{CBF}$.
6: **return** $True$
7: **else**
8: **return** $False$
9: **end if**
10: **end procedure**

8.2.2.3 Deletion

CBF allows the delete operation without increasing the false negative probability. During a deletion operation, first the item is hashed by K hash functions to obtain K slot locations. The counter of the locations is decremented by 1. When the counter value becomes 0, the corresponding data bit is reset to 0. CBF does not provide the number of occurrences/counts of items. Algorithm 8.3 presents the algorithm for the deletion operation of CBF. Assume the CBF in Fig. 8.1 with all the conditions as mentioned in the insertion operation section. Moreover, suppose the CBF contains items X and Y, i.e., consider the insertion operations as explained in the insertion section are performed. Suppose a delete operation for item X is performed. This X is hashed by $\mathcal{H}_1()$, $\mathcal{H}_2()$, and $\mathcal{H}_3()$, and three slot locations are obtained. The counters of $\mathcal{H}_1()$ and $\mathcal{H}_2()$ locations are decremented. The counter becomes 0, hence, the data bit is reset to 0. In case of $\mathcal{H}_3()$ location, the counter is decremented. The counter value becomes 1, and, as the counter value is a non-zero value, the data bit is not modified.

Theorem 8.1. *Deletion operation does not influence the false negative probability.*

Proof. When an item is deleted from CBF, the data bit is not reset to 0. Rather, the counter is decremented. Decrementing the counter does not influence the data bit till the counter becomes 0. In case the counter becomes 0, the data bit is reset to 0. But, the counter becomes 0 only when the number of items hashed to that slot location is deleted. Therefore, CBF does not return false negatives. However, the false negative probability is not zero in CBF. Other factors of CBF influence the false negative probability, which is discussed in Section 8.4 of the chapter. Assume the CBF in Fig. 8.1 with all the conditions as mentioned in the insertion operation section. Moreover, suppose the CBF contains items X and Y. In addition, suppose a delete operation for item X is also performed, as explained earlier.

Algorithm 8.3 Deletion of an item κ from Counting Bloom Filter $\mathbb{CBF}$ using three hash functions.

```
1:  procedure DELETE(CBF[], κ, S1, S2, S3)
2:      i1 = MURMUR(κ, length, S1) % μ
3:      i2 = MURMUR(κ, length, S2) % μ
4:      i3 = MURMUR(κ, length, S3) % μ
5:      if CBF[i1D] AND CBF[i2D] AND CBF[i3D] then                          ▷ κ is present
6:          CBF[i1C] ← CBF[i1C] − 1                                          ▷ Decremented counter
7:          if CBF[i1C] == 0 then
8:              CBF[i1D] ← 0             ▷ In case counter value is 0, the reset data bit to 0
9:          end if
10:         CBF[i2C] ← CBF[i2C] − 1
11:         if CBF[i2C] == 0 then
12:             CBF[i2D] ← 0
13:         end if
14:         CBF[i3C] ← CBF[i3C] − 1
15:         if CBF[i3C] == 0 then
16:             CBF[i3D] ← 0
17:         end if
18:     else
19:         False                                                            ▷ κ is absent
20:     end if
21: end procedure
```

Suppose that a lookup operation for item Y is performed. This Y is hashed by $\mathcal{H}_1()$, $\mathcal{H}_2()$, and $\mathcal{H}_3()$, and three slot locations are obtained. Among items X and Y, the X's $\mathcal{H}_3()$ location and Y's $\mathcal{H}_1()$ location are the same. But the data bit value is still 1, because during deletion of X the counter of that location is decremented and the counter value is non-zero; hence, the data bit value is not changed. Thus, the data bit value of all three slot locations for item Y is 1. So, CBF returns $True$. □

8.2.3 First counting Bloom Filter

Fan et al. [1] proposed the first CBF in the year 2000. CBF maintains a summary of the cache directory of every participating proxy. When a proxy sends a query, it first checks with the summaries. The URL of the document is given as input to CBF. Each proxy has its own local CBF. The size of the counter is kept at 4 bits. It has both false positive and false negative issues. Moreover, when the counter overflows, the counter value is kept constant at 15. Performing many delete operations leads to an increase in the probability of false negatives.

8.2.4 Analysis

False positive is a type of classification of the wrong response given by the Bloom Filter. It means Bloom Filter responds $True$, but the element is not present in Bloom Filter.

The false positive probability of CBF is

$$FPP_{CBF} = \left(1 - \left(1 - \frac{1}{\mu}\right)^{Kl}\right)^K \left(1 - e^{-Kl/\mu}\right)^K, \tag{8.1}$$

where μ is CBF array length, K is the number of hash functions, and l is the number of slots.

Every data bit in CBF is attached to a counter, which may overflow in case the duplicate occurrence of an item exceeds the capacity of the counter. Suppose 4 bits are assigned to the counter, then the probability of overflow is [1]

$$1.37 \times \mu \times 10^{-15}. \tag{8.2}$$

The overflow probability of CBF is minuscule. However, a huge duplicate entry into CBF causes the counter overflow. It may lead to an increment in the number of false negatives. This issue is discussed in Section 8.4 of the chapter.

8.3 Variants

Many variants of CBF are proposed mainly with the following aims: (a) having a CBF appropriate for a particular application, (b) reducing FPP, and (c) reducing memory consumption. In this section, many variants of CBF are discussed in detail. Moreover, various parameters of these variants are tabulated in Table 8.1, and the various advantages and disadvantages present in them are mentioned in Table 8.2.

8.3.1 Space-code Bloom Filter (SCBF)

Abhishek Kumar et al. [6] proposed a CBF variant called Space-Code Bloom Filter (SCBF) to represent multiset data. The filter of SCBF is partitioned into groups of hash functions. And each group is a Bloom Filter. In an insertion operation, any group is randomly chosen, and the slots are set to 1. During a query operation, all groups check for the element; if any group slot is set to 1, then that element is present. The number of groups that matched that element is counted to find the frequency of its occurrence. SCBF is unable to count multiplicity more than $g \ln g$ where g is the number of groups. Another variant of SCBF, called multiresolution SCBF (MRSCBF), is proposed, which correctly counts the multiplicity of elements. MRSCBF consists of multiple SCBFs where each SCBF has a different resolution. Each SCBF has a different sampling probability, which is referred to as resolution. All the SCBFs are used to cover the whole range of multiplicities. Higher-resolution SCBF predicts the lower multiplicity elements, and vice versa. During an insertion operation, the element is inserted into every SCBF. During a query operation, the number of groups that matched an element are counted in all filters. The multiplicity of the element is estimated using two methods, namely, by mean value estimation and maximum-likelihood estimation. However, these methods are very complex in MRSCBF due to the usage of multiple SCBFs.

8.3.2 Linked counting Bloom Filter

Kui Wu et al. [4] proposed a linked CBF, which gives the count of an element but does not follow the same structure as CBF. Linked CBF attaches a linked list to each slot of the standard Bloom Filter to record the count. Initially, all slots of Bloom Filter are set to 0. When an element is inserted, it is hashed to K locations. If a collision occurs, that location creates a linked list and stores the count. If a new element is inserted and a collision occurs, then a new node is created in that slot's linked list to store the new element's count. In the case of the removal of an element, the frequency is decreased from the linked list. In the case of multiple linked list matches, any random link is deleted, but this increases FPP. Moreover, the linked CBF uses extra bits to store the link and count. The Bloom Filter is further improved by implementing it in two layers. This improved linked CBF helps in reducing bandwidth when it is implemented in distributed systems. The first layer has an array of size m. After hashing, the obtained slot is set to 1. And the second layer has the linked CBF and stores the count. The bits in the first layer with bit value 0 do not have a linked list in the second layer. This helps in reducing space complexity. After setting the bit to 1, the corresponding linked list is searched to store the count.

8.3.3 L-counting Bloom Filter (L-CBF)

Safi et al. [7] proposed a CBF called L-CBF that improves the CBF at the hardware level. L-CBF consists of an array of special registers and local zero detectors. The register is up/down linear feedback shift registers (LFSR). LFSR is the counter in L-CBF. LFSR produces less delay, consumes less power, and has lower complexity compared to other synchronous up/down counters. L-CBF has a hierarchical decoder and a hierarchical output multiplexer. It is partitioned into several parts. Each row of a part has an LFSR and a zero detector. L-CBF takes three inputs and gives a single-bit output called *is-zero*. But this sharing of the counter increases the delay. The hierarchical decoder is used for address decoding. It helps to reduce energy–delay product [8]. The decoder consists of a pre-decoding stage, a global decoder, and a set of local decoders. The global decoder chooses an appropriate part of the L-CBF. Each part has a single local output line which is shared. Hierarchical multiplexer assembles all local outputs and

gives a single-bit output. L-CBF also implements divided word line (DWL) to save power and increase speed [9]. The improvements in hardware level reduce the time complexity, but space complexity is equal to that of CBF. It removes read and writes operations over long bit lines. L-CBF has implemented aggressive optimization to reduce energy and delay in the decoder and sense-amplifiers. For example, it uses a divided word line (DWL) [9] to consolidate power and speed. When the LFSR width increases, the delay also increases. Hence, the counter of L-SBF cannot be of large size.

8.3.4 d-left counting Bloom Filter

Lin et al. [10] proposed a d-left Counting Bloom Filter for implementing in dynamic packet filtering. The d-left CBF occupies less memory compared to CBF and increases rule capacity. It uses d-left hashing [11] and $k = d$, i.e., the number of hashing operations performed on the Bloom Filter is d. In the d-left hashing technique, the whole table is partitioned vertically into adjacent sub-tables. Suppose the whole hash table has a total of a buckets, then each sub-table has a/d buckets and each bucket has p slots where p is the number of sub-tables. In d-left CBF, each slot stores the fingerprint of the input element, and the other bits are used for counting. Each sub-table uses different hash functions. During insertion, the element is hashed to obtain d locations. The fingerprint is obtained and added to the lightly loaded bucket. In case of multiple lightly loaded buckets, the element is added to the leftmost bucket. When a duplicate fingerprint is encountered, the counter of the cell is incremented. This technique ensures load balancing in the Bloom Filter. In a query operation, the fingerprint is searched in d locations. In a delete operation, a procedure similar to CBF is followed. However, in the case of two elements having the same fingerprint, the common delete operation is avoided. In such a scenario, two different buckets have the same fingerprint. Hence, to know which fingerprint to delete, the hashing procedure is divided into two steps. In the first, the true fingerprint of the element is obtained. In the second step, d locations are obtained using additional linear random permutations. The FPP is $mn2^{-r}$ where the p is the number of sub-tables, l is the load of each bucket, and r is the remainder of fingerprints. Moreover, FPP of d-left CBF depends on r. Initially, d-left CBF takes less memory but gives more false positive results,which reduce with an increase in the number of inputs.

8.3.5 Variable-length counting Bloom Filter (VLCBF)

Li et al. [12] proposed a variable-length counting Bloom Filter having a variable-length counter. The counter length depends on the probability of the counting value. When the probability of a count is more, the counter length is long, and when the probability is less, the counter length is short. Two approaches are proposed, one is the direct approach, and the other is an improved approach. The direct approach is a simple approach. In an insert operation, first, an element is hashed to obtain K locations, i.e., h_i where $0 \leq i \leq k$. For each h_i location, the algorithm finds the h_ith 0. The bit after the h_ith 0 is set to 1. All bits after that bit is shifted right by one bit. For a query operation, K locations are determined. The bit after the h_ith 0 is checked whether it is 1. If that bit is 1 then the algorithm returns $True$. If all K locations give $True$, then the element is present. In a delete operation, first, K locations are obtained. The bit after the h_ith $(0 \leq i \leq k)$ 0 is deleted and all bits after that bit are shifted left by one bit. In the direct approach, all three operations (insert, delete, and query) consume some time while searching for the h_ith 0. Another improved approach is proposed to remove this overhead that uses standard Bloom Filter with VLCBF. In an insert operation, the same procedure is followed as in the direct approach. The K locations of standard Bloom Filter are also set to 1. The query operation is done only on a standard Bloom Filter. In the delete operation, the same procedure is followed as in the direct approach. In addition, after deletion, the bit after the h_ith 0 is checked whether it is 0. If it is 0, then the h_ith bit of the standard Bloom Filter is set to 0. The improved approach just removes an issue of the direct approach. However, using only standard Bloom Filter for insertion and query operation is sufficient to give efficient results. But such usage of Bloom Filter increases FPP.

Xuan et al. [13] proposed an improvement on VLCBF using a buffer to reduce the number of shift operations. The buffer stores the element's hash values and frequency of occurrence, i.e., each buffer slot has two parts. One part stores the hash value, and the other part, called the buffer counter, stores its frequency. The buffer counter is allowed to store negative values. To achieve that, buffer counter adds a sign bit. It also uses an auxiliary Bloom Filter. Initially, the counter value is 0. During the insertion of an element, the buffer stores the hash values of the element and increments the counter value. When the buffer is full, a single bit shifting is performed. Two approaches are proposed to improve VLCBF, one based on the initialized buffer and another based on a buffer update operation. In the approach based on the initialized buffer, the bit array is initially constructed. First, hashing of the element is

performed to obtain K locations. In an auxiliary standard Bloom Filter, h_i value for index i is set to 1. The buffer is searched for h_i value. If found, the buffer counter is incremented; otherwise, h_i value is added to the buffer, and the buffer counter is set to 1. When the buffer is full or all elements are appended at the end, the buffer is sorted in decreasing order based on hash values. Using the buffer and its counter information, shift operations in the bit array are performed. After completing the shift operations, the buffer is flushed. During an insert operation, if the buffer counter value for the hash value of the element is positive, then the standard Bloom Filter is directly set to 1. But, in the case of a negative buffer counter value, the buffer cannot be used to judge. Then the auxiliary standard Bloom Filter has to be changed accordingly. Another approach is based on updating operations on the buffer. In an insert operation, first, K locations are obtained. The buffer is checked for the hash values. If present, the buffer counter value is checked. If the counter value is more than 0, then the algorithm increments the value. If the buffer counter value is 0 or negative, then the value is incremented, and the hash value is queried in the array. If the value is greater than 0, then the h_i location is set to 1 in the standard Bloom Filter; otherwise, the h_i location is set to 0 at the end state. Finally, the buffer is checked for the fullness. If full, then the steps followed in the above approach are taken. The query operation is similar to the above approach. During a delete operation, K locations are found, and the buffer is checked for the hash values. If a hash value is present, then the corresponding buffer counter is decremented. If the counter is less than 0, then the h_i value is searched in the standard Bloom filter. If the value is more than 0, then the algorithm goes to the last state. But if the buffer does not have the hash value, then the counter value is set to 1. Finally, the buffer is checked for the fullness. If full, then the same steps as followed in the above approach are taken.

8.3.6 Balanced counting Bloom Filters (BCBFs)

Zhang et al. [14] have proposed balanced Counting Bloom Filters (BCBFs) to have space-efficient storage and efficient query operation. They improved the Bloom Filter by using hash fingerprinting, dividing array slots into equal memory segments, and storing elements in lightly loaded buckets. BCBFs spill the array slots into K equal-sized logical segments. Each segment corresponds to a hash function. And each segment has an equal number of buckets, i.e., m/k, where m is the array size. Each hash function distributes the elements uniformly among the buckets of its corresponding segment. Moreover, each bucket contains h memory cells, where h is the depth of the basket. BCBF uses the d-left hashing technique [15]. In case of an insertion operation, the element is hashed to obtain K buckets. In each segment, the element is inserted into a lightly loaded bucket. If multiple buckets are eligible, then the leftmost bucket is chosen. And in the case of heavily loaded buckets, the counter overflows, and the element is discarded. In a query operation, the bucket locations are obtained and checked for the presence of the element. If found, then the algorithm returns $True$; otherwise it returns $False$. BCBF has not devised a method to delete an element. Increasing the bucket depth value or a number of hash functions increases the FPP.

8.3.7 Floating counter Bloom Filter (FCBF)

Wang and Shen [16] proposed Floating Counter Bloom Filter (FCBF) to store existential probabilities in Probabilistic Data Stream effectively. FCBF has floating counters instead of an integer. FCBF stores the probability in a counter, so the counter value will not exceed 1. Hence, counter overflow does not occur in FCBF. During insertion, the element is hashed to obtain K locations. If, among the locations, the minimum counter value is 0, then the element is absent. But if the minimum counter value is not 0, then the existential probability of the element is the minimum counter value. Then the element is inserted and the probability of the element is stored in the floating counters using the formula $array[k_i] = array[k_i] + (1 - array[k_i]) * p$ where $i = 0, \ldots, k$ and p is the existential probability of the element. If the index of the element is more than the sliding window size, then the counter values stored in the K locations are outdated and the counter values are updated using the formula $array[k_i] = 1 - (array[k_i] - 1)/(1 - p)$ where $i = 0, \ldots, k$.

8.3.8 Detached counting Bloom array (DCBA)

Detached Counting Bloom array (DCBA) [17] is capable of efficiently identifying duplicates over sliding windows in the data stream. DCBA is an array of detached CBFs (DCBF). In DCBF, the counter is like a timer that stores the element's time-stamp. DCBF works like a CBF. When an element is inserted, first, it is checked with all DCBFs. If $False$ is returned, then the element is hashed to obtain K locations. All obtained locations are set to 1, and the counter stores the time-stamp. All timers are groups called timer array (TA) to increase access efficiency. The timer

size is $\log_2 W/(d-1)$, where W is the window size and d is the number of DCBFs. All the elements in a sliding window are inserted into DCBA. The capacity of each DCBF is $W/(d-1)$. The elements are inserted sequentially in the ith DCBF where $i = 1, \ldots, d$. The first DCBF contains the oldest elements. DCBF becomes old when the time-stamp is less than the base time-stamp of the current window. TA consumes a large amount of memory. TA of the oldest DCBF may be copied to disk to save RAM space. When the oldest DCBF becomes empty, it is reused as a new DCBF. DCBA is deployed both in a single node and in a distributed system. In a distributed system, DCBA is partitioned and distributed among the nodes. All the DCBF work as a circular FIFO queue. All the DCBF are checked simultaneously during a query to reduce searching overhead. Hence, the query time complexity is $O(k)$. However, if the oldest DCBF returns $True$, then the time-stamp is checked. If the data is old, then $False$ is returned.

8.3.9 Cache-based counting Bloom Filter (C^2BF)

Zhou et al. [18] proposed a cache-based CBF (C^2BF) to speed up the pattern matching. C^2BF has high-speed hardware, off-chip memory, and a fast cache-replacement technique. First, the requested string is hashed and checked with a standard Bloom Filter. When the Bloom Filter returns $True$, the cache checks for the data. If the cache hit occurs, the data is returned; otherwise, it is checked with CBF. The CBF slot has a linked list that stores the matched pattern. After the linked list is created, the associated patterns are stored in a table called T_1. Another table T_2 is also created that stores tag and pattern offset. When a string is deleted, it is also removed from CBF. But the string information is kept in T_1 and T_2. Another table T_3 stores the invalid entry address of T_1 and pattern of T_2. The usage of the standard Bloom Filter leads to more false positive responses. It leads to more cache checking. C^2BF has a complex design and larger space complexity.

8.3.10 Compressed counting Bloom Filter (CCBF)

Jin et al. [19] proposed a global compressed counting Bloom Filter (CCBF) reduce the transmission delay. And it helps to give an early response in the presence of specific content among every peering surrogate before checking on the local cache. Initially, CBF with 4-bit counter is constructed. The transmission time is decreased by reducing the size of CBF. Multilayer compressed CBF [20] is used to encode CBF messages using run-length code. Transmission size is further compressed using the delta compression technique. As per this technique, only the changes made in CBF are sent. When the interval increases in both the synchronization mechanisms (i.e., event-driven and periodic synchronization), the false negative probability of CCBF increases.

8.3.11 Temporal counting Bloom Filter (TCBF)

Zhao and Wu [21] proposed Temporal Counting Bloom filter (TCBF) to decrease content routing overhead. TCBF helps in decreasing computational complexity and compressing user interest. TCBF consists of slots with the attached counter. Instead of incrementing the counter by 1, TCBF increment the counter by a fixed value called initial counter value (ICV). Initially, the counter value is 0. When an element is inserted, the counter is assigned with ICV. And, if another element gets hashed to the same slots, the counter value is not changed. The element that has not been merged before is inserted into TCBF. Multiple elements are inserted into an empty TCBF, and then all TCBFs are merged. A TCBF is merged using two techniques, namely, A-merge and M-merge. A-merge does the addition of counter values. M-merge takes the maximum of the counter values. Deletion in TCBF is performed based on the decay. TCBF continuously decreases the counter value by the decay factor. After completion of TCBF construction, the counter gives the frequency of the element. Multiple TCBFs are maintained, which increases the space complexity. Moreover, merging TCBFs is also time-consuming.

8.3.12 Double layer counting Bloom Filter (DLCBF)

Lai et al. [22] proposed Double Layer Counting Bloom Filter (DLCBF) that increases the query searching speed. DLCBF has an extra layer of hash function and counting included at each entry of the filter. DLCBF structure uses a two-tier Bloom Filter. The top layer has a consecutive memory. The bottom layer consists of CBF and banked Bloom Filter [23]. The bottom layer provides bandwidth with high assessment capability. In this layer, a memory bank is an array with a counter and a bit vector. Moreover, DLCBF only maintains three memory banks in this layer. DLCBF also introduces an extra hash function to produce different permutations for the hash functions of

the bottom layer. The hierarchical structure helps in increasing the query speed. DLCBF is inserted between every private cache and shared system bus. When data is read from RAM, a coherent bus sends the data and address to both cache and DLCBF. In DLCBF, the corresponding top layer filter slots corresponding to the data are set to 1. And the corresponding bottom-layer filter entries are incremented when a cache triggers a search request during a query operation; all caches, except that which initiates the request, check whether the data is present with them. Simultaneously, both cache and DLCBF search for the data. If DLCBF returns $True$ (i.e., the cache contains the data), then its cache continues searching. But if DLCBF returns $False$ (i.e., the cache does not contain the data), then its cache terminates the search operation. DLCBF also deletes the data to keep the same record as the cache when the latter removes data. DLCBF does deletion when the cache performs a write-back or invalidation operation. In such situations, the cache does a search operation. After a cache hit occurs, DLCBF deletes the data.

8.3.13 Fingerprint counting Bloom Filter (FP-CBF)

Pontarelli et al. [24] proposed a fingerprint-based CBF, called Fingerprint Counting Bloom Filter (FP-CBF), to reduce the FPP. In FP-CBF, each slot stores the fingerprint of the element and counter. Each slot size is $c + f$ where c is the number of counter bits and f is the number of bits for fingerprint. Hence, the size of FP-CBF is $(c + f) \times m$ bits. During insertion, the element is hashed to K locations. Then, another hash function is used to map the element to a fingerprint bit. A bitwise XOR operation is performed between the additional hash value and the existing value in the fingerprint bits. Then, the algorithm increments the counter. The addition of XOR in insertion operations increases the computation time. During a query operation, the counter of K locations is checked. If the counter value is 0, then the element is absent. If the counter value is 1, then the algorithm checks whether the hash value and the fingerprint field match. If they do not match, then the element is absent. Otherwise, the element is present. The dual checking for the presence of an element increases the query time complexity. During a deletion operation, the counters of K locations are decremented. A bitwise XOR operation is performed between the additional hash value and the existing value in fingerprint bits. This operation deletes the fingerprint. The use of an additional hash function increases the hashing overhead. The counter is not of variable length so it can overflow.

8.3.14 Wrap-around counting Bloom Filter (WCBF)

Saurabh and Sairam [5] have proposed a wraparound counting Bloom filter (WCBF) to reduce the number of packets required for IP traceback. In WCBF, each slot has a cyclic counter. The cyclic counter helps to choose which mark needs to be transmitted to the victim to perform IP traceback quickly. The counter does addition modulo M, where $M = 2^s$, s being the number of bits for each slot in WCBF. Modulo operation prevents counter overflow. During insertion, the element is hashed using a uniform and independent hash function. Hashing is based on the destination address of the packet. And the value of the slot is utilized to choose the mark, which is later inserted into the packet. Then, the slot counter is increased by 1 modulo M. In the case packet loss or number of collisions increases, the performance of WCBF decreases.

8.3.15 Counting quotient filter (CQF)

Pandey et al. [25] proposed a counting quotient filter that is in-memory, small-sized, and fast. It has a good locality of reference and allows deletion and counting even on a skewed dataset. Moreover, it allows merging, re-sizing, and a large number of concurrent accesses. In the case of saturated CQF, the metadata bits are restructured to give a faster query response. CQF is constructed by adding counters to the rank-and-select-based quotient filter (RSQF). RSQF is a simple Bloom Filter that stores 2 bits of metadata per slot and takes $O(n)$ time for insert and query operation. RSQF partitions the bits of the hash value obtained from the hash function into the quotient and remainder. The first q bits of the hash value are called the quotient (Q), and the remaining r bits are called the remainder (R). Hence, the RSQF array size is $2qr$ bits. During an insert operation, R is stored in $array[Q]$. If the slot is occupied, linear probing is done to find an unoccupied slot and store Q. RSQF uses two metadata bit arrays to store slot occupancy and home slot information. CQF, i.e., RSQF with a counter, tries to consume the same memory as RSQF. The counter has variable length, which helps to handle highly skewed distributed data. In CQF, some slots store counts. Hence, to determine the slot that stores the remainder, CQF stores the remainder in increasing value. When a value does not store this pattern, it is a count. However, if the count is also following the pattern, it will give false information.

TABLE 8.1 Comparison of various parameters of variants of CBF; K is the number of hash functions, m is the array length, and n is the total number of elements, FN stands for False Negative, CO means Counter Overflow, while "-" indicates an unknown value.

CBF	False Positive	FN	CO	Fingerprint	Space Complexity
CBF [1], 2000	Yes	Yes	Yes	No	4 times that of a standard Bloom Filter
Space-code [6], 2006	Yes	Yes	No	No	$g \cdot \mu$, where g is the number of groups
Linked [4], 2008	$(1 - e^{-K\eta/\mu})^K$	Yes	No	No	More compared to a standard Bloom Filter
L-CBF [7], 2008	Yes	Yes	Yes	No	Same as CBF
d-left [10], 2009	$p \cdot l \cdot 2^{-r}$ where p is the number of sub-tables, l is the load of each bucket, and r is the remainder of fingerprint	-	Yes	Yes	-
Variable length [12], 2010	Yes	Yes	Yes	No	2^μ
Floating [16], 2010	$(1/2)^{\ln 2 * N/W}$ where N is the floating counter number, W is the sliding window size	No	No	No	-
Detached Counting Bloom Array [17], 2011	Yes	Yes	No	No	$d \cdot (c+1) \cdot (\log_2 e \cdot k \cdot W/d - 1)$ where c is the counter size, W is the window size, d is the number of DCBFs
C^2BF [18], 2012	Yes	Yes	Yes	No	Bloom Filter + CBF + three arrays
Balanced [14], 2012	$K/2^l$ where l is length of the element identifier	Yes	Yes	Yes	$\mu hr/\eta$ where h is the bucket depth, r is the length of element identifier
Compressed [19], 2014	Yes	Yes	Yes	No	Less than CBF
Temporal [21], 2014	Yes	Yes	No	No	More than one TCBF is maintained
Double Layer [22], 2015	Yes	Very low	Yes	No	More compared to CBF
Fingerprint [24], 2015	Yes	Yes	Yes	Yes	$(c+f) \cdot m$ bits
Buffer-based VLCBF [13], 2015	Yes	Yes	No	No	More compared to VLCBF [12]
Wraparound [5], 2016	Yes	No	No	No	Same as CBF
Counting Quotient Filter [25], 2017	Yes	Yes	No	No	$O(x \log \frac{\eta\mu}{\delta x^2})$ where δ is FPR, and x is the number of distinct items
Dual [26]	Yes	Yes	Yes	No	$2(4 \cdot \mu + K \cdot l \cdot \ln\ \mu)$ where l is the packet size

It may be rare, but the possibility exists. Size of CQF is $O(x \log \frac{nm}{\delta x^2})$ where δ is FPR, and x is the number of distinct items.

8.3.16 Dual counting Bloom Filter (DCBF)

Dodig et al. [26] proposed Dual Counting Bloom Filter (DCBF) to reduce incorrect detection of matching packets in $SACK^2$ algorithm [27]. DCBF consist of two CBFs (say, CBF1 and $\overline{CBF1}$). CBF1 accepts acknowledgment (ACK) packets and recognizes the corresponding ACK for each SYN/ACK packet. $\overline{CBF1}$ analyzes the inverse packet. The inverse of the input packet is found by complementing every bit of the input packet. This additional CBF reduces the incorrect detection of matching packets but increases space complexity. CBF1 and $\overline{CBF1}$ use the same hash functions. The counters in both CBF1 and $\overline{CBF1}$ can overflow. The memory occupied by DCBF is $2(4 \cdot m + k \cdot l \ln m)$ where m is

TABLE 8.2 Listing of Advantages and Disadvantages of Variants of CBF.

CBF	Advantages	Disadvantages
CBF [1], 2000	• Deletion allowed	• Four times more space consumed compared to standard Bloom Filter • Counter overflow occurs
Space-code [6], 2006	• Efficiently stores millions of search keywords • MRSCBF is efficient for low multiplicity elements	• SCBF is unable to detect a higher multiplicity of an element • Estimation methods are complex in MRSCBF
Linked [4], 2008	• No counter overflow • Bandwidth usage is less in two-layered Linked CBF • Gives the frequency of the elements • Two-layered linked CBF has less space complexity compared to linked CBF	• Majority entries in the linked list is empty • Space complexity more compared to standard Bloom Filter • Extra bits used to store links and frequency • Bandwidth increases with increase in the number of hash functions in two-layered Linked CBF
L-CBF [7], 2008	• Removes read and write values over long bit-lines • Reduces energy consumption • Used DWL [9] to reduce power and increases speed • Implemented aggressive optimization to reduce energy and delay in decoder and sense-amplifiers	• Occupies larger area in the hardware circuit • Sharing of counter consumes more energy • Increase in linear feedback shift registers (LFSR) width leads to increase in delay
d-Left [10], 2006	• Loads on buckets are balanced • Compact information storage • FPP depends on the reminder of the fingerprint • Higher memory-utilization • Bigger rule capacity	• Initially, during Bloom Filter construction, the false positive probability is more compared to that of CBF
Variable length [12], 2010	• Space complexity is 2/3 times that of CBF	• Time complexity: Insertion = Deletion = same as CBF and $Query_{VLCBF} > Query_{CBF}$ • Including standard Bloom Filter in an improved approach increases space complexity • Shifting of bits increases time complexity
Floating [16], 2010	• Counter is used to store existential probability • Counter overflow does not occur as the probability is within [0, 1] • Applicable in probabilistic data streams • False negative error is absent	• Change in sliding window size leads to deletion and update of values in FCBF • False positives exist
Detached Counting Bloom Array [17], 2011	• Can be deployed in a single or distributed system • Timer array increases access efficiency • Searching in DCBFs is done in parallel	• Timer array consumes a large amount of memory • Timer array is written to disk to reduce RAM space but increases I/O operations • Multiple CBFs are used, which increases the memory space requirement • In case of a large window size, RAM cannot accommodate DCBA
C^2BF [18], 2012	• High speed hardware • Follows a fast cache-replacement technique	• Element is only removed from CBF not from other tables • Standard Bloom Filter has more false positives, hence more cache checking is performed • Complex design • Many tables used, which increases space complexity
Balanced [14], 2012	• Element is inserted into lightly loaded bucket • d-left hashing reduces FPP • Storing hash fingerprint decreases FPP • FPP is less compared to CBF	• In heavily loaded buckets, correlated element are discarded • Increasing h, the depth of each bucket increases FPP • Increasing K increases FPP • No delete method followed
Compressed [19], 2014	• Reduces response time • Bloom Filter size is less • Local cache checking overhead is reduced	• It is just a CBF of smaller size • No new features • False negative probability increases with an increase in interval

continued on next page

TABLE 8.2 (*continued*)

CBF	Advantages	Disadvantages
Temporal [21], 2014	• Decreases content routing overhead • Counter gives the frequency of the element • Reduces computational complexity • Helps in compressing user interest • Reduces memory and bandwidth consumption in B-SUB	• More than one TCBF is maintained, which increases the space complexity • Merging of Bloom Filters increases time complexity • False positive response leads to useless message forwarding
Double Layer [22], 2015	• Reduces query operation time • Reduces unnecessary cache operation • Multiple bank increases the access speed of bandwidth • Reduces false negative responses • Parallel access to multiple bank leads to one query operation for each banked memory • Permutation array lowers the data collision probability	• Top layer is not efficient • Memory area is more compared to CBF
Fingerprint [24], 2015	• Reduces the number of false positives • Simple design	• Uses $k+1$ hash functions, which increases the hash operation overhead • Involves query operation dual checking, which increases time complexity • FPP is reduced but still present • XOR operation during insertion
Buffer-based VLCBF [13], 2015	• Reduces the number of shift operations • Reduces the time complexity compared to VLCBF	• Space complexity increases as an additional buffer gets added to VLCBF • Increase in buffer size increases search time
Wrap-around [5], 2016	• Reduces the number of packets for IP traceback • Usage of a cyclic counter prevents counter overflow	• Increase in packet loss leads to increase in requirement for packets in IP traceback • Increase in collision leads to degradation in performance
Counting quotient filter [25], 2017	• Smaller and faster with skewed inputs compared to uniform random inputs • In SSD storage, insert and query operation takes $O(1)$ I/O • Has cache-efficient variable-sized counters • Does in-memory insertion and query operation • Has good locality of reference and supports deletion, resizing, merging, and high concurrency • No counter overflow as counter is of variable length	• Time complexity of insertion and query operations is $O(n)$
Dual [26], 2017	• Reduces incorrect detection of matching packets • Less memory used • Less processing required • Convenient for embedded systems	• Uses standard CBF • Counter overflow occurs • When the counter is half-full, DCBF resets all slots

number of slots in CBF, K is the number of hash functions, and l is the packet size. The factor of 2 is due to the use of two CBFs.

8.4 Issues

- **Counter overflow.** Usually, CBF counters are of fixed size. The corresponding counter value is incremented with each encounter of duplicate elements. However, the fixed-sized counter overflows in many applications when the number of duplicate elements exceeds the capacity of the counter. After overflowing the counter, it is unable to record the correct number of duplicate elements. Counter overflow increases the false negative probability. Suppose a CBF is defined as having a counter length of 4 bits. The counter capacity is $2^4 - 1 = 15$. For simplicity, the insertion of a single element is considered. Let x be an element, and suppose x is inserted into the CBF 100 times. Let c be one of the corresponding counters of x, and c is incremented. After reaching the value 15, the counter overflow occurs, and the value is not incremented furthermore. Now, suppose consecutive delete operations of x are performed. Then, the slot and the counter are set to 0 after 15 delete operations. After the delete operations,

for any query operation on element x, CBF returns a false negative response. As x is inserted 100 times, another $85 = 100 - 15$ instances of x are present in CBF, but the slot value is currently 0. Therefore, the counter of CBF needs to have variable-sized to increase the counter size with increasing duplicate data.

- **Space complexity.** Space complexity is another main issue in CBF. The conventional CBF has a size four times larger than that of the conventional Bloom Filter. CBF is used to store unique data. A huge dataset requires a big-sized CBF as a small CBF gives false positive responses upon the saturation. Usually, Bloom Filters are kept in RAM for faster access. But, when the size of CBF is huge, it is impossible to maintain a big-sized CBF in RAM. Therefore, an appropriate-sized CBF is required to achieve higher performance.
- **False Positives.** Similar to Bloom Filter, CBF also has issues with false positives. However, the false positive rate is higher than that of the conventional Bloom Filter. CBF is also used in network security techniques (e.g., DDoS attack [5]). A false positive response is an error that leads to unnecessary searching in an application. Moreover, many variants of Bloom Filter also propose an extra step of searching to reduce the FPP. For example, DLCBF [26] introduces an extra CBF to double-check the response given by the first CBF. But this increases the space complexity of the DLCBF. In addition, achieving zero false positive probability is impossible, but it should be decreased as much as possible.
- **Scalability.** Currently, Bloom Filter is preferred for Big Data applications. Bloom Filter drastically reduces the processing. However, when the data size increases, the CBF becomes saturated, and FPP increases. Hence, CBF needs to be highly scalable to handle huge amounts of data. Besides, scalability is also required in the counter. When the number of duplicate data increases, the counter should be able to scale in size to avoid counter overflow. Therefore, a scalable CBF is required to face today's Big Data era.
- **Hashing.** CBF performs hashing to obtain K locations before performing any operation. So, the time complexity of individual operations increases by using complex hashing techniques. Usually, the time complexity of the hashing technique is ignored. Consequently, the time complexity CBF decreases with each operation asymptotically. However, this contradicts a practical scenario. In addition, the hash function should provide uniform distribution. If the function is biased towards some slots and maps frequently to a particular slot, it leads to their early counter overflow. Therefore, an appropriate and less complex hash function is essential.
- **Number of hash functions.** The time complexity of insert, query, and delete operations is $O(k)$ where K is the number of hash functions. As K is constant, the time complexity becomes $O(1)$. However, these operations are dependent on a number of hash functions. So, it is necessary to have fewer hash functions. But less hashing in some CBFs increases FPP. However, K also cannot be too large. Otherwise, FPP increases. Therefore, an appropriate number of hash functions are required to achieve optimal time complexity.
- **Dependency on input element size.** CBF is dependent on the input element size in contrast to the conventional Bloom Filter. Many Bloom Filter variants use fingerprints to reduce memory consumption. However, as shown in Table 8.1, very few CBF variants have used fingerprints for their design. As discussed above, the memory consumed by a CBF is more dependent on the input element size that needs to be resolved to reduce the space complexity of a CBF.
- **Performance.** CBF increases performance by filtering duplicate data. However, many factors influence the performance of CBF. If the space complexity of CBF is more and the data size is huge, then the size of CBF will be huge. Then, it is not possible to store the CBF in RAM. But, the presence of CBF in RAM makes the processing faster. In some applications, the data of the CBF are written to disk to reduce RAM space consumption [16]. In such cases, the I/O operation reduces the performance drastically. Therefore, the factors that negatively affect the performance need to be tuned to increase the overall performance of CBF.

8.5 Conclusion

CBF was proposed in the year 2000 to enable Bloom Filters to delete data from their data structure. As a variant of Bloom Filter, it is also used for deduplication and filtering. The counter present in every slot helps to execute the delete operation. However, CBF has larger false positive and false negative probabilities. The lesser time and space complexity enables it to be used in applications dealing with huge volumes of data. Its application drastically reduces the data processing by the main technique that reduces the computation time and increases performance. CBF is widely used in hardware, networking, data streaming, and many other domains. However, an additional counter is also introduced to reduce the false positive probability with this variant of Bloom Filter. A fixed-sized counter leads to counter overflow and makes the data structure size four times more compared to a standard Bloom Filter. As CBF

is also a Bloom Filter, it has the same issues due to false positives, false negatives, scalability, etc. Therefore, we need a variant of CBF that has all these issues resolved. Another challenge is to reduce memory consumption by the CBF.

References

[1] Li Fan, Pei Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area web cache sharing protocol, IEEE/ACM Trans. Netw. 8 (3) (2000) 281–293, https://doi.org/10.1109/90.851975.
[2] S. Nayak, R. Patgiri, countBF: a general-purpose high accuracy and space efficient counting Bloom filter, in: 2021 17th International Conference on Network and Service Management (CNSM), 2021, pp. 355–359.
[3] L. Bai, D.M. Li, Using Sample and Multilayer Compressed Counting Bloom Filter Algorithm to Realize Elephant Flows Identification, Applied Mechanics and Materials, vol. 651, Trans Tech Publ, 2014, pp. 2228–2232.
[4] K. Wu, Y. Xiao, J. Li, B. Sun, A distributed algorithm for finding global icebergs with linked counting Bloom filters, in: 2008 IEEE International Conference on Communications, IEEE, 2008, pp. 2757–2761.
[5] S. Saurabh, A.S. Sairam, Increasing the effectiveness of packet marking schemes using wrap-around counting Bloom filter, Secur. Commun. Netw. 9 (16) (2016) 3467–3482.
[6] A. Kumar, J. Xu, J. Wang, Space-code Bloom filter for efficient per-flow traffic measurement, IEEE J. Sel. Areas Commun. 24 (12) (2006) 2327–2339.
[7] E. Safi, A. Moshovos, A. Veneris, L-CBF: a low-power, fast counting Bloom filter architecture, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 16 (6) (2008) 628–638.
[8] B.S. Amrutur, M.A. Horowitz, Fast low-power decoders for RAMs, IEEE J. Solid-State Circuits 36 (10) (2001) 1506–1515.
[9] B.S. Amrutur, Design and analysis of fast low power SRAMs, PhD thesis, Stanford University, 1999.
[10] P. Lin, F. Wang, W. Tan, H. Deng, Enhancing dynamic packet filtering technique with *d*-left counting Bloom filter algorithm, in: 2009 Second International Conference on Intelligent Networks and Intelligent Systems, 2009, pp. 530–533.
[11] B. Vöcking, How asymmetry helps load balancing, J. ACM 50 (4) (2003) 568–589, https://doi.org/10.1145/792538.792546.
[12] L. Li, B. Wang, J. Lan, A variable length counting Bloom filter, in: 2010 2nd International Conference on Computer Engineering and Technology, vol. 3, IEEE, 2010, V3-504.
[13] S. Xuan, D. Man, W. Wang, W. Yang, The improved variable length counting Bloom filter based on buffer, in: 2015 Eighth International Conference on Internet Computing for Science and Engineering (ICICSE), IEEE, 2015, pp. 74–78.
[14] Z. Zhang, B. Wang, J. Liu, Balanced counting Bloom filters: a space-efficient synoptic data structure for a high-performance network, IET Commun. 6 (15) (2012) 2259–2266.
[15] Y. Azar, A. Broder, A. Karlin, E. Upfal, Balanced allocations, SIAM J. Comput. 29 (1) (1999) 180–200, https://doi.org/10.1137/S0097539795288490.
[16] X. Wang, H. Shen, Approximately detecting duplicates for probabilistic data streams over sliding windows, in: 2010 3rd International Symposium on Parallel Architectures, Algorithms and Programming, IEEE, 2010, pp. 263–268.
[17] J. Wei, H. Jiang, K. Zhou, D. Feng, H. Wang, Detecting duplicates over sliding windows with ram-efficient detached counting Bloom filter arrays, in: 2011 IEEE Sixth International Conference on Networking, Architecture, and Storage, 2011, pp. 382–391.
[18] Y. Zhou, T. Song, W. Fu, X. Wang, C2BF: cache-based counting bloom filter for precise matching in network packet processing, Proc. Eng. 29 (2012) 3747–3754.
[19] Y. Jin, Y. Wen, W. Zhang, Content routing and lookup schemes using global Bloom filter for content-delivery-as-a-service, IEEE Syst. J. 8 (1) (2013) 268–278.
[20] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, Multilayer compressed counting Bloom filters, in: IEEE INFOCOM 2008 – The 27th Conference on Computer Communications, IEEE, 2008, pp. 311–315.
[21] Y. Zhao, J. Wu, The design and evaluation of an information sharing system for human networks, IEEE Trans. Parallel Distrib. Syst. 25 (3) (2013) 796–805.
[22] B.-C.C. Lai, K.-T. Chen, P.-R. Wu, A high-performance double-layer counting Bloom filter for multicore systems, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 23 (11) (2015) 2473–2486.
[23] M. Breternitz, G.H. Loh, B. Black, J. Rupley, P.G. Sassone, W. Attrot, Y. Wu, A segmented Bloom filter algorithm for efficient predictors, in: 2008 20th International Symposium on Computer Architecture and High Performance Computing, IEEE, 2008, pp. 123–130.
[24] S. Pontarelli, P. Reviriego, J.A. Maestro, Improving counting Bloom filter performance with fingerprints, Inf. Process. Lett. 116 (4) (2016) 304–309.
[25] P. Pandey, M.A. Bender, R. Johnson, R. Patro, A general-purpose counting filter: making every bit count, in: Proceedings of the 2017 ACM International Conference on Management of Data, ACM, 2017, pp. 775–787.
[26] I. Dodig, V. Sruk, D. Cafuta, Reducing false rate packet recognition using dual counting Bloom filter, Telecommun. Syst. 68 (1) (2018) 67–78.
[27] C. Sun, C. Hu, B. Liu, Sack2: effective SYN flood detection against skillful spoofs, IET Inf. Secur. 6 (3) (2012) 149–156.

CHAPTER

9

Hierarchical Bloom Filter

9.1 Introduction

Bloom Filter is a simple data structure which helps in deduplication and filtering in many applications. The most attractive feature of Bloom Filter is its application versatility. The implementation of Bloom Filter is not confined to a specific domain. It can be implemented in any application. It just depends on the developer's idea of where to use and what to give as input. However, one of the issues of Bloom Filter is false positives. An element could be absent in Bloom Filter, but the filter gives a true response. This leads to overhead. Indeed, when the Bloom Filter gives $False$ as a response on a query for an element, then no further processing is done by the technique or application as the data is not present. But in case of false positives, further processing is done assuming that data is present. During processing, it is found that the data is not present. Hence, a false positive leads to overhead, which decreases efficiency.

The main motive for developing variants of Bloom Filter is to reduce its issues. However, the majority have the main motive of reducing the false positive probability. Therefore, to reduce it, another type of variant is developed, called the hierarchical Bloom Filter. In this filter, different variants or the same type Bloom Filters are combined as a single component. But the structure is hierarchical. Hence providing an architecture is difficult (as discussed in previous chapters). Different hierarchical Bloom Filters follow different architecture. In a hierarchical Bloom Filter, memory consumption is more because multiple Bloom Filters are implemented. Many hierarchical Bloom Filters use a function called $popcount(B[i])$ where B stands for Bloom Filter [1–3]. It helps to find the location in the next layer corresponding to the current layer values. $popcount$ operation is performed by hardware and is also supported by multi-core architectures. The cost of $popcount()$ is very low, which encourages the use of hierarchical architecture to develop Bloom Filters. Counting Bloom Filter suffers from counter overflow. To increase the capacity of the counters, hierarchical architecture is considered [4,5].

In this chapter, various variants of the hierarchical Bloom Filter are discussed in detail. A table is also included based on various parameters and the advantages and disadvantages of the variants. Then, various issues present in the hierarchical Bloom Filter are listed and discussed. Finally, the technique implementing the variants of the hierarchical Bloom Filter (only those discussed in this chapter) is explained.

9.2 Variants

With the motive of reducing the issue with false positives, various variants are developed. In this section, various hierarchical Bloom Filters are discussed. The variants' architecture and various operations supported by them are explained in detail. Table 9.1 shows a comparison of various parameters and advantages and disadvantages of hierarchical Bloom Filter variants.

9.2.1 Hierarchical Bloom Filter array (HBA)

HBA [6,7] is a hierarchical Bloom Filter proposed to help in mapping file names to servers. HBA has two layers. The first has lower accuracy and stores all metadata. Low accuracy reduces memory overhead. The second layer has high accuracy and caches partial distribution information while using the temporal locality of file access patterns. The first layer Bloom Filter is called LRU BF. It stores all the files that are cached in the LRU of the owner metadata server. Entries to LRU are very small, so a large bit/file ratio is considered to reduce FPP. In the second layer, the number of files stored in servers is large, so a small bit/file ratio is considered to reduce memory consumption.

Bloom Filter https://doi.org/10.1016/B978-0-12-823520-1.00014-6

Copyright © 2023 Elsevier Inc. All rights reserved.

When a miss in the first layer occurs, the second layer is searched. If searching in the second layer also fails, then a broadcast is made to all other servers. During a query operation, the file name is converted to indexes, which are given as input to Bloom Filter. To convert a file name to indexes, MD5 signature of the file full path name is calculated. FPP is $sf(1-f/s)^{s-1}$ where s is the total number of metadata servers and f is FPP of a single Bloom Filter. HBA uses universal hash functions to reduce FPP. However, implementing a conventional Bloom Filter to design HBA gives high FPP. In addition, HBA consumes more memory.

9.2.2 Multi-granularities counting Bloom Filter (MGCBF)

MGCBF [5] is a hierarchical Bloom Filter that is capable of giving the frequency of an element. It has lesser space complexity but takes more processing time. MGCBF consist of multiple CBFs at different layers. It uses a set of different counter granularities for counting the frequency of an element. During an insert operation, the element is hashed using k hash functions. The first layer counters at k locations are set to 1. Then MGCBF checks the counter value. If the counter is $c_1 = co_1$, then counter values are decreased by co_1. And the counters at the second layer increment the counter value. Then, second layer counters checks whether $c_2 = co_2$. A similar procedure is followed till the last layer. During a query operation, the element is hashed to obtain k locations. The frequency is calculated by following a formula using the counter values. In the case of a heavy-tailed frequency distribution in the dataset, the memory space by each CBF in subsequent layers decreases following a power law because small-sized elements have a high frequency. The space complexity of MGCBF's CBF is less compared to conventional CBF. However, data structure maintenance takes more computational time. The false positive and false negative probabilities are larger in the higher layers. MGCBF implements a recurring minimum method [8] to reduce the error probability in higher layers. But this increases space complexity.

9.2.3 Cached counting Bloom Filter (CCBF)

CCBF [9,10] is a variant of Bloom Filter with a multi-level cache architecture. It has less memory access compared to a conventional Bloom Filter. CCBF consists of a Bloom Filter, and each counter points to a layer based on its counter value; the Lth layer contains L buckets. The same counter values are stored in counter-value-numbered layers (e.g., if a counter value is 3 then it is stored in the 3rd layer). During insertion of an element, the element is hashed by k hash functions to obtain k locations. The counters of k locations are checked to find the maximum value. The element is stored in the layer corresponding to the counter with the maximum value. For example, if location 4 has a maximum counter value, then the element is stored in layer 4. CCBF does two types of operation. The first includes programming and query operation. Programming is inserting elements to cache layers in CCBF. In a query operation, first, the algorithm checks the counter value and then loads the elements from the related cache layer. The second type of operation includes fetch and insert/delete operations. In these operations, the procedure is similar to operations performed in the traditional hash table.

9.2.4 Hierarchical counting Bloom Filters (HCBF)

Yuan et al. [4] proposed a Bloom Filter based on length-descending and counter-ascending multi-layer CBF for storage of data. HCBF consists of multiple CBFs. Each CBF is in a different layer. And each CBF has a different length for both bit array and counter. The array size decreases with increase in layer, i.e., $m_0 > m_1 > \cdots > m_l$ where m is the size of the CBF and l is the number of CBFs. A Level Counter (LC) is used to store the maximal layer of that element. The maximal counter of each CBF is 2^c where c is the counter length. However, the counter capacity is less than the maximal counter. When inserting an element, every layer is searched for the presence of the element. The respective counter is incremented if the counter does not exceed its capability. If the counter capability is exceeded, the counter is decremented, and the layer of the element is increased. In the case of a new element (first-time insertion), its information is added to LC. Removal of an element follows a procedure similar to that of the insertion operation. From the lowest layer, the element is searched. If the counter of the CBF is more than 0, then the algorithm decrements. Otherwise, it increments the counter and decreases the layer of the element. After decrementing, if the element is only present in the lowest CBF and the counter becomes zero, then its information is removed from the LC. HCBF provides both membership and multiset queries. The membership query gives whether the element is present. A multiset query gives the frequency of the element. For a membership query, LC is checked to see if the value is more than 0, if this happens, that element is present in HCBF. In a multiset query, the counter of the element in every layer

is retrieved. Each layer unit is multiplied, and then all products are added. This gives the frequency of the element. HDFS stores multiple copies of elements. Keeping multiple copies reduces space advantages.

9.2.5 Blooming tree

Blooming Tree [1] is a hierarchical data structure to reduce space complexity and FPP. In this Bloom Filter, the data structure is distributed among different memory layers. This makes it small and faster. Its design is similar to a binary tree, so it is called the blooming tree. In Blooming Trees, binary trees are constructed on a Bloom Filter. In this Bloom Filter, the array size is based on the input elements, i.e., $nk/\ln 2$, where n is the number of input elements and k is the number of hash functions. Suppose the number of layers is L. Then, the first layer (L_0) is a Bloom Filter. The last layer (L_L) consists of counters of size c. The other $L-2$ layers consist of blocks of 2^b bits in size. The hash function produces an output string of size $\log_2 m + L \times b$ bits. This output string is divided into addresses to refer to Bloom Filter and blocks. The first $\log_2 m$ bits address the Bloom Filter. The next b bits address L_1 block, and the next sequential b bits address the next layer, and so on. Blooming Tree uses a function called $popcount(B[i])$ where B stands for Bloom Filter. This function returns the locations of the next layer that need to be considered for the operation. During an insert operation, the element is hashed to obtain the output string. If the element is absent from the Bloom Filter at L_0, then the algorithm sets the k locations and obtains the popcount value (p_1) for the next layer. In L_1, at p_1 locations the bits are right-shifted by 2 bits and the next layer p_2 is calculated. If the element is present in L_0, then the algorithm moves to the next layer. In the next layer, it sets p_1 location. A similar procedure is followed in subsequent layers. The address of the counter in the last layer is obtained from the hash output string and incremented. The time complexity for an insert operation is $k \cdot (hash + L \cdot (popcount + shift + bitset)$. In a query operation, L_0 Bloom Filter is checked. If the element is absent, then the false response is returned. If the element is present, then the popcount value (p_1) is calculated, and these locations are checked. If the element is absent, then the false response is returned. Otherwise, the popcount value (p_2) is calculated, and the same procedure is followed till the last layer. Therefore, in query operation, all layers are checked if the element is present, and then the next layer is checked. And, if all layers contain the element, then the true response is returned. Otherwise, if the element is absent from all layers, then the false response is returned. The time complexity for a query operation is $k(hash + L(popcount + 2 \cdot bit_{check}))$. However, the query operation cost is very small because hash and popcount operations are performed by hardware and are also supported by multi-core architectures. During the delete operation, the counter in the last layer of the element is decremented. If the counter value is 0, then the locations of the previous layer L_{L-1} are checked. If the bits are set, then the bits are left-shifted by 2^b bits. Then a similar procedure is followed for other layers. If the counter value is greater than 0, then the delete operation is terminated. An increase in the number of layers increases the space complexity but decreases FPP.

9.2.6 Multilayer compressed counting Bloom Filter (ML-CCBF)

Multilayer compressed counting Bloom Filter (ML-CCBF) [3] uses Huffman codes to increase efficiency and decrease query time. Huffman counting Bloom Filter (HCBF) uses Huffman coding in spectral Bloom Filters (SBF) [8] to improve the performance. HCBF uses Huffman code in SBF to encode a number element (say, x) with x consecutive 1s and a trailing 0. Using Huffman code reduces the complexity of a query operation. SBF partitions the array to sub-array and uses a set of tables during the query. ML-CCBF is similar to the rotated version of HCBF. Huffman-coded counter value bits in HCBF are stored in ascending layers. ML-CCBF stores a bit per layer for each encoded symbol. The first layer is a conventional Bloom Filter. The subsequent layers are dynamic, i.e., these layers are constructed and modified as per the operations performed. The bits that are Huffman-coded counter values are placed in ascending layers. During a query, only the layer 0 Bloom Filter is checked. Hence, the query time complexity is $O(1)$. ML-CCBF partitions all layers into the same bit-sized block D. Each layer has a table. $popcount$ function is used to traverse to the next layer. First $\log_2(m_i/D)$ bits of value returned by $popcount$ at layer i are index to the table where m_i is the number of bits in the layer. ML-CCBF is appropriate for storing multiset data. However, it is inappropriate for high updating datasets. A counter overflow is prevented by adding layers, but this increases space complexity. When the whole ML-CCBF is kept in the same cache level, it reduces the overall performance.

9.2.7 Multilevel counting Bloom Filter (MLCBF)

MLCBF [11] is proposed to perform stateful replication with less resources. In MLCBF, the first layer is the largest and tries to store the majority of the elements. This helps to perform all operations in the first layer. MLCBF has k hash functions and k layers. The hash functions are uniform and independent. Each layer consists of a different number of bucket elements (BE). The number of BEs in each layer decreases linearly. A BE consist of T cells and a loaded bitmap (LB). The size of LB is T bits, and it stores elements and records in-use cells' information. Furthermore, each cell consists of a counter called cell counter (CC) with c bitsize and a fingerprint of f size. A different hash function is used to obtain the fingerprint. During an insert operation, the fingerprint of the element is obtained. The presence of a fingerprint is checked. If the latter is absent, the element is stored in the cell whose location is obtained by k hash functions. If absent, then the corresponding CC is incremented. When a layer is full, the elements are stored in the next higher layer. During a query operation, the fingerprint of the element is obtained. The presence of a fingerprint is checked sequentially in every layer, starting from the first layer. During a delete operation, the first query operation is performed. If the response is $True$, then CC is decremented. In case of $CC = 0$, LB is set to 0.

9.2.8 Reversible multilayer hashed counting Bloom Filter (RML-HCBF)

Reversible MultiLayer Hashed Counting Bloom Filter (RML-HCBF) [12] is a variant of ML-HCBF [3] designed to identify elephant flows. It has a fixed number of layers, i.e., 3. The first layer uses 8 hash functions. The first seven hash functions give some consecutive bits from original strings as output which is considered the flow ID. The last h_8 function takes flow ID as input and gives a value range. It helps to prevent incorrect flow reconstruction. The second layer uses a single hash function (h_9) which takes the location of the saturated counter of the first layer as input. Similarly, the third layer uses a single hash function (h_{10}), which takes the location of a saturated counter of the second layer as input. During insert operation, the string is hashed, and 8 locations are found. If the counter in the first layer is saturated, then h_9 is used to hash and find the counter located in the second layer. Similarly, if the second layer is saturated, then the third layer is referred. FPP is $\left(1-\left(1-\frac{1}{m}\right)^{50n}\right)^k$ where m is the array size, n is the number of flows, and k is the number of hash functions. RML-HCBF has no false negatives and lesser FPP. It has no counter overflow. It neither stores flow ID nor flow ID lookup, which increases performance. However, the use of 10 hash functions and false reconstruction of the string increases the time complexity.

9.2.9 Multiple-partitioned counting Bloom Filter (MPCBF)

MPCBF [2] is proposed to provide a solution for large-scale data processing issue. A hierarchical Bloom Filter reduces FPP and performs single memory access. MPCBF partitions the counter array into a word array which is further partitioned into Hierarchical Counting Bloom Filters (HCBF). This HCBF has a structure similar to HCBF [4] but follows different operational procedures. Each HCBF further subdivides its array segment into d sub-vectors where d is a number of layers. The counter may cover different layers. It uses $popcount(j)$ function that calculates the number of 1s before position j. HCBF follows a hierarchy update technique. In this technique, $popcount(j)$ value (say, p) is used as index for the next layer. Suppose a bit is set to 1, then in the next layer, all bits after p are right-shifted by 1 bit. In the insert operation of HCBF, the element is hashed to obtain k locations. At the first layer, the k locations are set to 1, then the algorithm performs hierarchy update. But if in the first layer the bit is 1, then every layer is checked sequentially at the pth bit with 0 value. If found, the hierarchy update technique is followed. The counter value at the first layer is assigned a value which is equal to the depth of the layer traversed. The delete operation procedure is similar to the insert operation. The first layer is checked for the presence of the element. If not found, the next layer is checked. The layers are traversed till the last bit with value 1 is encountered. That bit is set to value 0, and the algorithm left-shifts the bits after p. In MCBF-1 (1 refers to one memory access), a single hash function is used to obtain a word location. During the insert or delete of an element, first, it is hashed, and then the word follows HCBF insert or delete procedure, respectively. Hence, the insert or delete operation takes only one memory access. One can ignore popcount time complexity because it is hardware-supported. FPP decreases with an increase in HCBF size at each layer, but it increases space complexity and counter overflow. This Bloom Filter variant is further improved by increasing the number of hash functions, which increases the memory access to g. The improved MPCBF is MPCBF-g, where g is the number of memory accesses. MPCBF-g has lesser FPP compared to MCBF-1. However, the space complexity and processing time increase with more Bloom Filters used.

9.2.10 Bloofi

Bloofi [13] is a hierarchical Bloom Filter introduced to reduce searching time. Bloofi has a tree-like structure. The nodes are Bloom Filters. The operations in Bloofi are similar to a B tree. Parent nodes are obtained by applying bitwise or on children Bloom Filters. This process is repeated till the root is reached. Each non-leaf Bloom Filter is the union of sub-tree Bloom Filters rooted at that non-leaf Bloom Filter. So an element not present in leaves is not present in any non-leaf Bloom Filter. During a search in Bloofi, the root Bloom Filter is checked for the presence of the element. If the latter is absent, then $False$ is returned. If present, children of the root are checked for the presence of the element. A similar process is repeated along the matching path till the leaf is reached. Hence, the query operation takes $O(d \cdot \log_d N)$ where d is a lower bound for the number of child pointers or order of Bloofi and N is the number of Bloom Filter nodes. However, considering multiple paths increases time complexity. An update operation in Bloofi is the insertion of an element. In the update operation, the value in the Bloom Filter is changed from the root to the leaf. The time complexity is $O(\log_d N)$. Similar Bloom Filters are kept in the same layer to decrease search time complexity. During new Bloom Filter insertion in Bloofi, the most similar Bloom Filter node is searched. Hamming distance is used to check the similarity between nodes. From the root, the similarity checking is done till the leaves are reached. When the most similar Bloom Filter leaf node is found, the new node is inserted as its sibling. If the order of the parent leaf is more than $2d$, then splitting of nodes is done. This splitting follows a procedure similar to that of the B tree. Similarly, the deletion of a Bloom Filter node follows the deletion procedure similar to the B tree. Bloom Filter node insertion and deletion operation takes $O(d \cdot \log_d N)$ time. The space complexity of Bloofi is $O(N/d)$.

9.3 Issues

1. **Space complexity.** Space complexity is one of the main issues in hierarchical Bloom Filters due to their large architecture. This type of Bloom Filter consists of multiple Bloom Filters of the same [13,1] or different type [3]. With an increase in the number of Bloom Filters, the space complexity increases. However, achieving a hierarchical Bloom Filter with lesser space complexity compared to a single Bloom Filter is impossible.
2. **False Positive.** The main motive to develop a hierarchical Bloom Filter is to reduce the FPP. Many hierarchical Bloom Filters use a conventional Bloom Filter to have a less space complexity. But FPP of a conventional Bloom Filter is higher compared to other Bloom Filter variants. Therefore, other Bloom Filter variants should be considered that give lesser FPP with smaller space complexity.
3. **Scalability.** In some hierarchical Bloom Filters, the number of layers is increased to increase scalability [13,1]. However, while achieving that the space complexity is also increased further. And, when the memory consumption becomes high, the whole Bloom Filter cannot be kept in RAM.
4. **Time Complexity.** Many hierarchical Bloom Filters store their data in every layer. And every operation first searches for the requested element. But this searching takes more time because all layers need to be searched. Therefore, the time complexity of all operations increases.
5. **Hashing.** Hashing is performed in every operation of Bloom Filter. Usually, it is taken as a constant value and gets ignored. But it adds to the overall time complexity of the operations performed by the hierarchical Bloom Filter. In addition, some hierarchical Bloom Filters use more hash functions due to more layers.
6. **Number of hash functions.** The time complexity of insert, query, and delete operations is directly dependent on the number of hash functions. Moreover, in hierarchical Bloom Filters, more hash functions are used due to more layers. Therefore, an appropriate number of hash functions are required to keep the time complexity of all operations as low as possible.
7. **Performance.** Compared to other types of Bloom Filter, hierarchical Bloom Filter performance is less. In hierarchical Bloom Filters, the use of multiple Bloom Filters increases the issues many-fold. In addition, hierarchical Bloom Filters consume a lot of memory. The time complexity of all operations is also more because all the layers need to be searched.

TABLE 9.1 Comparison of various parameters of variants of hierarchical Bloom Filter; k is the number of hash functions, m is the array length, and n is the total number of elements; "-" indicates an unknown value.

Hierarchical Bloom Filter	False Positive	Advantage	Disadvantage
Hierarchical Bloom Filter Arrays [6,7], 2004	$sf(1-f/s)^{s-1}$ where s is the total number of metadata servers, f is the FPP of a single Bloom Filter	• High bit/file ratio in the first layer reduces FPP • False negatives are possible • Universal hash functions are used	• Large memory consumed • Conventional Bloom Filter is used, which has high FPP
MultiGranularities Counting Bloom Filter [5], 2006	Yes	• For a heavy-tailed frequency distribution in the dataset, the memory space by each CBF in subsequent layers decreases following a power law • Space complexity of MGCBF's CBF is less compared to conventional CBF • False negatives are possible • Recurring minimum method reduces the error probability in higher layers	• Data structure maintenance takes more computational time • False positive and false negative probability is more in higher layers • Recurring minimum method increases space complexity
Cached counting Bloom Filter [9,10], 2007	Yes	• Less memory access compared to conventional Bloom Filter	• FPP is high as conventional Bloom Filter is used • Efficiency of CCBF decreases with increase in the number of layers • Counter overflow occurs • False negatives are possible
Hierarchical [4], 2008	Yes	• Time complexity is same compared to CBF • Gives the frequency of the element	• Stores multiple copies of elements • False negatives are possible
Blooming Tree [1], 2008	$2^{-(k+Lb)}$ where L is the number of layers, b is the number of bits per block	• Data structure is distributed among different memory layers which makes its smaller and faster • Query operation cost is tiny as hash and popcount operations are performed by the hardware and also supported by multi-core architectures • Increase in the number of layers decreases FPP • 2.88 times smaller than conventional CBF	• Lookup operation is more costly • Checking the presence of an element in every layer increases the cost of the operation • False negatives are possible • Increase in the number of layers increases the space complexity • First layer Bloom Filter is dependent on the number of input elements, $nk/\ln 2$
Multilayer compressed counting Bloom Filter [3], 2010	Yes	• Applicable to store multiset • Counter overflow does not occur • Faster query operation • Time complexity of query is $O(1)$ and that of insert/ delete is $O(1)$	• More space complexity • FPP is more • False negatives are possible • Counter overflow is eliminated by adding layers, which increases space complexity • Keeping the whole data structure in the same cache layer reduces overall performance • Not appropriate for high updating dataset
BLOOM-FLASH [14], 2011	Yes	• One flash read required for each query operation • Does single I/O per operation, independent of the number of hash functions	• Delete operation not allowed as conventional Bloom Filter is implemented • False negatives are possible
Multilevel Counting Bloom Filter [11], 2011	Yes	• Memory consumption is less compared to conventional CBF • Network cost is less • Fingerprint is used to reduce space complexity • The issue with false negatives is less	• Bucket overflow occurs • Incorrect state error occurs

continued on next page

TABLE 9.1 (*continued*)

Hierarchical Bloom Filter	False Positive	Advantage	Disadvantage
Reversible Multi-layer Hashed Counting Bloom Filter [12], 2012	$\left(1-\left(1-\frac{1}{m}\right)^{50n}\right)^k$	• No false negatives • Very low FPP • Fixed number of layers, i.e., 3 • No counter overflow • Does not store flow ID or needs flow ID lookup, which increases performance	• Usage of 10 hash functions increases the time complexity • False reconstruction leads to increase in time complexity
Multiple-Partitioned Counting Bloom Filter [2], 2013	Yes	• Does single memory access • In HDFS, counters are of variable size • FPP decreases with increase in HCBF size at each layer • MCBF-g has less FPP • Insert/delete operation requires memory access equal to the total number of layers	• In HDFS, counter overflow occurs • Increase in HCBF size at each layer increases space complexity and counter overflow • MCBF-g has larger space and time complexity • False negatives are less of an issue
Bloofi [13], 2013	Yes	• Similar Bloom Filters are kept in same layer to increase search performance • Higher order consumes less memory • Increase in order decreases update operation time complexity	• Space complexity is larger • Considering multiple path during query operation increases time complexity • Insertion of Bloom Filter node does similarity checking which increases time complexity compared to a delete operation • Higher order leads larger searching time complexity • False negatives are possible

9.4 Applications

Hierarchical Bloom Filters are used in many domains. They help to increase the scalability of the Bloom Filter. They also help to prevent counter overflow by partitioning the counter into different layers. In this section, the hierarchical Bloom Filter variants' (only those of this chapter) implementation in various techniques/methods is discussed.

9.4.1 MapReduce

Multiple-Partitioned Counting Bloom Filter (MPCBF) [2] is a hierarchical Bloom Filter that reduces FPP and does single memory access. MPCBF is implemented in MapReduce [15]. It helps to reduce I/O cost in a join operation in Reducer. In that operation, CBF is used to remove unwanted output records that are read after shuffling. MPCBF is an alternative to CBF. It helps in reducing Map output, which reduces the number of join operations. Hence, it improves the Reducer time complexity.

9.4.2 Distributed system

Feng et al. [11] proposed stateful replication using MLCBF in high availability clusters. It drastically reduces memory and network cost for replication. Moreover, the replication is performed with small and constant latency. MLCBF helps to include the skewness effect and distributes the elements among the layers. It helps to achieve stateful replication for a large number of active flows. In key-and-state access, MLCBF stores both the key and access values in the data structure. It has mainly three errors, namely, those due to false positives, false negatives, and inaccurate states. An inaccurate state refers to an incorrect state response returned by MLCBF. These errors mainly occur due to hashing, fingerprint collision in case of dynamic operations, bucket/counter overflow, and early recycling of active flows, which depends on memory management.

Zhu et al. [6,7] proposed Hierarchical Bloom Filter Array (HBA) to uniformly distribute metadata management tasks among metadata servers. HBA is present in every metadata server. Every metadata server constructs its own HBA locally and later replicates to all other metadata servers. All filters are stored in an array. When a client makes a request for a file randomly, a metadata server is chosen to check this file in its HBA. If metadata is present, then HBA

returns *True* as the response. Whenever the LRU list overflows as per LRU replacement policy, insert and delete operations are performed in its own HBA. When the changes made exceed a threshold, the metadata server sends its HBA to all other metadata servers. HBA balances metadata workload, with no migration overhead present, and provides a flexible metadata placement method.

9.5 Conclusion

With the aim of reducing the false positive probability and increasing scalability, another type of Bloom Filter is introduced, called hierarchical Bloom Filter. This type of Bloom Filter has a tree/forest structure. Its layer structure helps to increase scalability. Also, many counting Bloom Filters use hierarchical architecture to increase the scalability of their counter. However, the time complexity of all basic operations of a Bloom Filter increases due to its architecture, as all layers need to be searched before performing the operation. Moreover, they consume a lot of memory as they consist of multiple Bloom Filters compared to a conventional Bloom Filter. In a Bloom Filter, having false negatives is another main issue, but the majority of hierarchical Bloom Filters have ignored it. Also, no hierarchical Bloom Filters have considered using fingerprints even when their data structure occupies huge memory. However, they are used in many applications such as distributed systems and networking. In distributed systems, client requests come in millions per second to find metadata for the requested data. And all nodes have to search for the metadata in their respective node. But using a hierarchical Bloom Filter reduces that overhead by giving the membership response within a constant time complexity. Based on the response, the nodes take action. There are many issues that the hierarchical Bloom Filter needs to solve. Moreover, many options are also available to increase efficiency, which they can consider.

References

[1] D. Ficara, S. Giordano, G. Procissi, F. Vitucci, Blooming trees: space-efficient structures for data representation, in: 2008 IEEE International Conference on Communications, 2008, pp. 5828–5832.

[2] K. Huang, J. Zhang, D. Zhang, G. Xie, K. Salamatian, A.X. Liu, W. Li, A multi-partitioning approach to building fast and accurate counting Bloom filters, in: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 1159–1170.

[3] D. Ficara, A. Di Pietro, S. Giordano, G. Procissi, F. Vitucci, Enhancing counting Bloom filters through Huffman-coded multilayer structures, IEEE/ACM Trans. Netw. 18 (6) (2010) 1977–1987, https://doi.org/10.1109/TNET.2010.2055243.

[4] Z. Yuan, Y. Chen, Y. Jia, S. Yang, Counting evolving data stream based on hierarchical counting Bloom filter, in: 2008 International Conference on Computational Intelligence and Security, vol. 1, IEEE, 2008, pp. 290–294.

[5] Z. Mingzhong, G. Jian, D. Wei, C. Guang, Multi-granularities counting Bloom filter, in: International Conference on High Performance Computing and Communications, Springer, 2006, pp. 542–551.

[6] Yifeng Zhu, Hong Jiang, J. Wang, Hierarchical Bloom filter arrays (HBA): a novel, scalable metadata management system for large cluster-based storage, in: 2004 IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935), 2004, pp. 165–174.

[7] Y. Zhu, H. Jiang, J. Wang, F. Xian, Hba: distributed metadata management for large cluster-based storage systems, IEEE Trans. Parallel Distrib. Syst. 19 (6) (2008) 750–763.

[8] S. Cohen, Y. Matias, Spectral Bloom filters, in: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, ACM, 2003, pp. 241–252.

[9] M. Ahmadi, S. Wong, A cache architecture for counting Bloom filters, in: 2007 15th IEEE International Conference on Networks, IEEE, 2007, pp. 218–223.

[10] M. Ahmadi, S. Wong, A cache architecture for counting Bloom filters: theory and application, J. Electr. Comput. Eng. 2011 (2011) 12.

[11] Y.-H. Feng, N.-F. Huang, Y.-M. Wu, Efficient and adaptive stateful replication for stream processing engines in high-availability cluster, IEEE Trans. Parallel Distrib. Syst. 22 (11) (2011) 1788–1796.

[12] W. Liu, W. Qu, Z. Liu, K. Li, J. Gong, Identifying elephant flows using a reversible multilayer hashed counting Bloom filter, in: 2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems, 2012, pp. 246–253.

[13] A. Crainiceanu, Bloofi: a hierarchical Bloom filter index with applications to distributed data provenance, in: Proceedings of the 2Nd International Workshop on Cloud Intelligence, Cloud-I'13, ACM, New York, NY, USA, 2013, pp. 4:1–4:8.

[14] B. Debnath, S. Sengupta, J. Li, D.J. Lilja, D.H.C. Du, Bloomflash: Bloom filter on flash-based storage, in: 2011 31st International Conference on Distributed Computing Systems, 2011, pp. 635–644.

[15] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.

PART II

Applications of Bloom Filter in networking

CHAPTER

10

Applications of Bloom Filter in networking and communication

10.1 Introduction

A network consists of nodes such as servers, switches, and routers. These nodes communicate with each other following some communication protocols over digital interconnections, which are optical, wired, or wireless radio-frequency media. These interconnections may be arranged in various network topologies. The nodes are recognized in the network by their addresses, which are identifiers assigned to locate the node uniquely. However, these addresses may not be unique in local or private addresses. Some addresses are special and are used for broadcast or multicast. These addresses are also not unique. Some addresses are flat, i.e., do not contain any information about the node's location, for example, MAC address. Also, some addresses contain hierarchical information that helps in routing, such as Internet Protocol (IP) addresses. Currently, networks are capable of transmitting text, audio, and video data of high quality.

Two main issues in network communication are the huge traffic and the high quality of services. Internet users are increasing exponentially: each user is connected to the Internet through multiple devices. With the increase of the users, the network traffic is also growing exponentially. A packet needs to cross many devices before it reaches its destination. At the same time, each device performs some processing on the packet to determine the next path or check for security purposes. For different purposes, a device follows a different protocol. It makes the whole networking process very complex and lengthy. Thus, a network requires a data structure that makes these processes simple. In case of high quality of services, the packet needs to be delivered quickly without corruption because retransmission of a packet affects the quality of the services. Thus, a network requires a data structure that takes less time for the execution of its operations.

In this chapter, the role of Bloom Filter in the network and communication are discussed. This chapter only focuses on wired communication. Section 10.2 reviewed some Bloom Filter-based techniques on traffic management. Section 10.3 mentions some reviews on Bloom Filter-based techniques on packet management. Section 10.5 includes some reviews on Bloom Filter-based techniques for searching and prefix matching. Section 10.4 reviews some Bloom Filter-based techniques on routing. Section 10.6 discusses Bloom Filter from the context of network and communication. Finally, Section 10.7 concludes the article.

10.2 Traffic management

The development of information technology brought many changes in the network architecture. The number of users has increased, and the network services are more diversified. However, more packets need to be transmitted to achieve more services or high quality. Along with this, more illegitimate information has also flooded the network. Thus, measuring and monitoring network traffic has become essential. It is required for many network applications such as quality of service management, trending network applications, network optimization, and anomaly behavior detection. It also helps in optimizing the resources required for communication. However, it has many challenges, such as it is a data-intensive problem, the volume of the packet is in millions, high-speed communication, concurrent flows, etc. Bloom Filter is a simple data structure, and high-speed operation execution can handle such high speed and high volume packets. In this section, some methods are reviewed based on Bloom Filter for network traffic measurement, monitoring, and management.

 Copyright © 2023 Elsevier Inc. All rights reserved.

Zhou et al. [1] proposed a solution for the threshold-based widespread event detection problem. In this problem, the same event occurs in many nodes, for instance, the request for the trailer video of an expected super-hit movie. The threshold filter solution integrates Bloom Filters, counting Bloom Filter, and threshold filter. All nodes inform their events to the coordinator node. Instead of raw data, each node sends a Bloom Filter. The Bloom Filter presents the event set. Transmitting Bloom Filter reduces communication overhead. After receiving the Bloom Filters, the coordinator node updates a counter array. The counter array is an array of counters. The coordinator node calculates the frequency of an event and saves the frequency in the counter array. Then the counter array is converted to a threshold filter. The events having 0 or 1 occurrences are assigned to 0 in the threshold filter, whereas if the occurrence is more than 1, the corresponding event slot is set to 1.

Bloom Filters and Least Recent Used (BF-LRU) [2] combines both Bloom Filter and Least Recently Used (LRU) replacement policy. Bloom Filter determines elephant flow, and LRU policy helps to determine mice flow. When a packet arrives, it is first checked in Bloom Filter to determine whether the packet belongs to a previously identified elephant flow. If not, then it is inserted into the top of the LRU list. If the cache is full, the least used entry is removed, and the new entry is added at the top of the list. LRU is improved by using a bidirectional hash list with perfect hash functions to prevent collisions. The collisions generated by LRU increase the number of operations. Otherwise, each packet requires one lookup operation to obtain the locality of the flow. Bloom Filter and the bidirectional hash list are stored in the SRAM. To avoid the saturation of data structures, both Bloom Filter and bidirectional hash lists store the information for a fixed time period (say, bins). After completing the time period, all flow records and Bloom Filter are exported. Then both data structures are reset to the empty state. The exported bins are later used for traffic analysis.

10.2.1 Traffic statistics

In this section, we have reviewed some Bloom Filter techniques that help collect traffic statistics, and also compared them in Table 10.1.

FlexSketchMon [3] is a generic architecture for flexible sketch-based network traffic monitoring. It uses Bloom Filter for recording traffic flow statistics. The flow-level data is collected by a hardware data plane. It consists of two tables, flow counter and flow key table. The flow counter maintains the count of the packets and bytes. The flow key table stores the flow ID/key. Five values are extracted from the packet header when a packet is received, namely, source IP, source port, destination IP, destination port, and protocol. The hash function module hashes these five tuples. The hashing provides the flow ID. The flow ID is queried to a Bloom Filter to determine whether the packet is new or not. If it is new, then it is added to the flow key table. This procedure is followed for a fixed time interval. When this time duration is over, these tables are forwarded to the CPU, which are used as per the requirements of the monitoring applications.

Counting Bloom Filter Sketch (CBFSketch) [4] combines Count-Min Sketch and counting Bloom Filter (CBF) for efficient transmission of summaries of data streams. The sketch is a compact data structure that stores summaries of data streams such as a number of active flows and node connectivity. CBFSketch has layers of sketches. When one sketch overflows, then a new sketch is constructed. It helps to construct sketches based on the traffic dynamically. The size of lower layer sketches is more compared to higher layer sketches. CBFSketch has two parts, sketch and CBF. CBF records the depth of the sketches where the item may be found. An item has two parameters, key and value. The key value is different in different situations; for example, for data packet size, the key value is the length of the item, and for frequency of the item, the key value is 1. Initially, CBFSketch had a single sketch and a CBF set to 0. During an insertion operation, the item is hashed by multiple hash functions. The hashed value gives the location of the counters in the sketch. All these counters are checked to find the counter having the smallest value. Then the algorithm adds the key-value to this counter. In case many counters have the same smallest value, the key value is added to all the counters. No operation is performed in CBF. A new sketch is constructed when any counter overflows, and the next layer sketch does not exist. After the construction, the counter value of the old sketch is set to $c_i + v - 2^{k_i} - 1$ where c_i is previous value of the counter, v is key-value, and k_1 is the size of the sketch. The corresponding counter in the new sketch is set to 1. Then the item is again hashed using the hash functions of the CBF. Similarly, the hash value gives the location of the CBF counters. The counter is incremented, which has the least value among these counters. The value of CBF (say, d) indicates the item is present from layer 1 to layer d sketches. Items are always inserted into the first sketch. When overflow occurs in the first layer, insertion is performed in the second layer. If overflow occurs in the second layer, a third layer sketch is constructed. Insertion operation of CBF is performed accordingly. This procedure is followed for the construction of the layers. In a query operation, the item is

TABLE 10.1 Features and Limitations of Traffic Management and Statistics.

Technique	Features	Limitations
Zhou et al. [1]	• Nodes send Bloom Filter to reduce communication overhead • Threshold filter reduces communication overhead • Threshold filter occupies less memory	• Maintains many data structures • Overhead is reduced by multiple translation of the filter to another filter
BF-LRU [2]	• Bloom Filter determines elephant flow • LRU policy determines mice flow • Identifying the elephant flow using Bloom Filter reduces false negative probability • Bloom Filter reduces computation in LRU	• LRU list generates collision • LRU collision increases the number of operations • Smaller bin length leads to frequent flushing of Bloom Filter • Larger bin length leads to saturation of Bloom Filter • Accuracy of LRU depend on its size
CBFSketch [4]	• Sketches are generated dynamically • Occupies less memory • Size of higher sketch is less compared to lower sketch	• Overhead is more for elephant flow • Insertion and query operation are performed on multiple sketches • An item present in last layer takes the highest processing time • Time duration of the operation is depended on the number of layers the operation need to be performed
FlexSketchMon [3]	• Flow key table is updated when a new packet arrives whereas flow counter table is accessed for every packet	• Number of flow ID stored in the table depend on the Bloom Filter size • False positives of Bloom Filter result in inequality between number of entries of flow counter and flow key table • Implemented standard Bloom Filter
Bloomtime [5]	• Provides flow statistics • Calculates the mean and variance of the packet inter-arrival • Scalability is independent of number of flows	• False positive probability is based on the increase in the number of flows • Maintains multiple Bloom Filters • Updating of Bloom Filter is performed when Bloom Filter controller is ideal
PBF [6]	• During insertion, the location cells are not read to prevent reading overhead • Algorithm reaches Nash equilibrium • PBF calculates the frequency of a flow • PBF does not set k cells to 1 • Avoids reading of cells to reduce reading overhead	• Maintains two Bloom Filters, standard and PBF • Frequency calculation requires a statistics inference method

hashed by CBF hash functions. The hashed value is the location of the counters. These counters are searched for the least value. The least value gives the d value. Then the item is hashed by the hash function of the sketch. The hashed value gives the counter locations within the sketch layer from 1 to d. The least value among the counters from each layer is selected, then the final value estimate is calculated based on a formula.

Bloomtime [5] monitors the network, which is implemented in hardware. It measures the time difference between packets. The time is divided into t windows of duration w milliseconds. One Bloom Filter is maintained for each window. First, the window is stored in the first Bloom Filter. The first Bloom Filter content is stored in the second Bloom Filter when the time duration is over. Thus, t Bloom Filters are maintained for t windows. Bloomtime has five hardware modules: time counter, header parser, hash functions, bloom filter controller, and SRAM backend. The time counter module reminds the Bloom Filter to update the Bloom Filters. When a packet is received, the header parser module checks whether the packet is a TCP flow. If not, then it is discarded. Otherwise, the addresses and ports of source and destination are extracted from the header. Then the flow identifier is forwarded to the hash module. Hash module hash the flow identifier by two hash functions. The two hash values are forwarded to the Bloom Filter controller module. The Bloom Filter controller loads from Static Random Access Memory (SRAM). The hash value provides the cell location in the first Bloom Filter. These locations are set to 1. Then these are written back to SRAM. After receiving the signal for updating of Bloom Filter, the update is performed when the Bloom Filter controller is ideal. Measurement software is used for computing the mean and variance of packet inter-arrival times for every flow. Measurement software has three functions: monitoring the hardware registers, reading Bloom Filters marks, and initializing the SRAM value after a reading process. There is a monitoring time product of time window size and

number of Bloom Filters. Hardware is activated after these monitoring periods. The measurement software retrieves flow arrival time and Bloom Filter marks using the registers interface. The collected data are stored in a local file for future processing. These data are used to calculate the mean and variance of packet inter-arrival.

Probabilistic Bloom Filter (PBF) [6] is a variant of standard Bloom Filter proposed for efficient determination of flow frequencies. The flow ID is hashed by k hash functions in insertion operation. The k hashed value provides the locations in PBF. It uses a parameter to select which locations are set to 1. Hence, PBF does not set k cells to 1. The frequency query operation provides the frequency of the flow. The flow ID is hashed by k hash functions in this operation. The number of 1s in k locations provided by the k hashed values are counted. The flow frequency is calculated by using the count in a statistical inference method. After receiving a packet, the flow ID is checked in a standard Bloom Filter (say, Bloom Filter). If the flow ID is present in Bloom Filter, then that packet is from a frequent flow. If the flow ID is absent, it is checked in PBF and calculates its frequency. If the frequency is more than a threshold value, the flow ID is inserted into Bloom Filter.

10.3 Packet management

A packet is a formatted unit of data that contains the message and control information helping in transmission. The message is the data the sender wants to deliver to the destination: text, audio, or video. The message is called payload. The control data helps in transmission, security, error correction, etc. The main components of the packet are addresses, error detection and correction, hop limit, protocol identifier, priority, and payload. The addresses are the source and destination addresses. The packet contains parity bits, checksum, and cyclic redundancy for error detection and correction. The destination uses these to determine errors in the packet. If an error is present, then it is corrected or discarded. Hop limit gives the number of nodes the packet can traverse to reach the destination. This helps in eliminating infinite looping of packets. Length gives the size of the packet. Protocol identifier refers to the communication protocol the packet is following for transmission. Priority determines the importance of the packet, i.e., these are transmitted first to maintain the quality of service of the application the packet belongs to. Various information of the packet helps in solving various problems in the network. Bloom Filter is also an excellent solution to solve many issues related to packets. One primary use is Bloom Filter helps in faster filtering of packets. It eliminated malicious packets, erroneous packets, etc. In this section, some techniques based on Bloom Filter are discussed, which helps solve problems related to packets management and also, shown in Table 10.2.

Terzenidis et al. [7] implemented the Bloom Filter for packet forwarding to an optically-enabled node. The nodes have a programmable Si-pho switching matrix, and the packets are transmitted at 10 Gb/s. The OR operation is performed between the IDs of the resources represented by a switch port. The result gives the address which is assigned to the switch port. Each switch port has a Bloom Filter, and the address is inserted into the Bloom Filter. When a packet is received, AND operation is performed between the destination address present in the packet header and the Bloom Filters of all switch ports. If the result is the same as the destination address, the packet is forwarded to that port. Change of resources in any switch port requires reconstruction of the Bloom Filter.

HyperSight [8] is a system to monitor changes in packet behavior. It contributes a declarative query language called Packet Behavior Query Language (PBQL) to define the network monitoring intents. Hypersight uses an algorithm called Bloom Filter Queue (BFQ) for saving the packet behaviors on data planes. In addition, Hypersight enables BFQ for dynamic reconfiguration by using virtual BFQ (vBFQ). It helps in the dynamic compilation of network monitoring tasks without interrupting on-service switches. The architecture of HyperSight has four layers. The first layer receives queries from all applications where the queries are defined using PBQL. A centralized processor represents the second layer which receives data from all switches and provides data to queries. The third layer is CPUs of all switches that cleans the raw data from programmable ASIC and forwards the data to the centralized processor. The final layer has Application-specific integrated circuits (ASIC) that receive packet behavior change at line rate. HyperSight is capable of executing multiple queries simultaneously. The centralized processor forwards the queries collected in the first layer to switches. Then switches compile the queries to table entry, and the entries are installed dynamically into ASIC. Next, HyperSight instantiates vBFQ for each query. The BFQ consists of multiple Bloom Filters (say, t) of the same size. BFQ partitions the packets into blocks where all blocks have the same number of packets when packets are received. One Bloom Filter stores the behavior of a packet block. The packet behavior is checked with all the Bloom Filters. The packet behavior is hashed by t hash functions. Each hash value refers to a cell in a Bloom Filter. If all Bloom Filter has 1 in the cell, the packet is duplicate otherwise received for the first time.

TABLE 10.2 Features and Limitations of Packet Management Techniques.

Technique	Features	Limitations
Terzenidis et al. [7]	• Bloom Filter helps is determining the switch port quickly • Adaptable to the change in network topology	• Change of resources in any switch port requires reconstruction of the Bloom Filter • Implements standard Bloom Filter with high FPP
HyperSight [8]	• Decreases delay for every flow and reports all discretized delay changes • Reports changes that happen to the port-level throughput of all flows • Monitors packet retransmission • Provides security against DoS/DDoS attack by checking duplicate packets • Reduces monitoring overheads • High scalability • High coverage • Layered architecture helps in reducing the data volume • Increasing the window size reduces the overhead • BFQ exhibits good performance in resource-constrained switches	• Design of vBFQ cannot support dynamic reconfiguration of all elements • vBFQ cannot add new packet behavior fields dynamically • vBFQ cannot dynamically allocate or deallocate memory of physical arrays • Increase in window size of the packet blocks increases FPP of Bloom Filter • Increase in number of Bloom Filters decreases false negative probability but increases memory usage • vBFQ included additional processing logic affecting the performance
Time-out Bloom Filter (TBF) [9]	• Sampling bias towards long flows • Effective selection of packets for sampling	• All packets of a flow are lost if a false positive response is returned for all packets • False positive response leads to non-sampling of a packet • Inappropriate selection of hash function results in increase of correlation
Pache [10]	• Tree reduces the bucket searching time complexity • False positive table reduces the FPP by half • False positive table reduces communication cost • Large Bloom Filter size improves cache hit • Large Bloom Filter size reduces traffic	• Maintains many data structures • False positive issue of Bloom Filter is handled by using an additional data structure. • Large Bloom Filter size increases communication cost • Small broadcast duration increases traffic • Large broadcast duration decreases cache hit rate
COVE [12]	• Piggybacking of Bloom Filter reduces overhead • Synchronization is achieved by piggybacking the Bloom Filter rather than broadcasting	• Standard Bloom Filter is used • Substrate node having large radius does not get selected

Time-out Bloom Filter (TBF) [9] is a Bloom Filter variant used for packet sampling. The sampling method is selecting some packets from a flow and using these packets to determine the characteristics of the flow. This helps in the detection of attacks and determines the type of traffic, network requirements, optimal usage of resources, etc. However, in large flows, some small flows are missed, and that information is lost. This leads to a high loss overall. TBF is an array of m buckets which stores time-stamps. TBF considers a fixed value called bucket time-out value. This algorithm requires the packet key and time-stamp. When a packet is received, its key is hashed by k hash functions. These hash values provide the location of the buckets in TBF. If in k buckets the difference between item time-stamp and the bucket time-stamp is greater than the bucket time-out value, then that packet is selected; otherwise discarded. Then all k buckets are updated to the packet time-stamp. The buckets are updated even when the packet is discarded.

Pache [10] is a packet management scheme for enhancing network performance. It consists of a protocol, a cache management mechanism, a cache sharing mechanism, and a cache node placement mechanism. The Bloom Filter is implemented in the cache sharing mechanism. Every caching router broadcasts Bloom Filter to all nodes to share the cache information. Pache follows the cache management mechanism inspired by Bloom Filter-Aided haSh Table (BFAST) [11]. It uses CBF and a hash table for cache management. Each node maintains a CBF and hash table. Each CBF counter corresponds to a bucket in the hash table. During insertion, the item is hashed by multiple hash functions. The hash values provide the location of the CBF counters, which are incremented by 1. The counter which has the smallest value is selected among these counters. The item is stored into the hash bucket corresponding to the selected counter. During query operation, the item is hashed by multiple hash functions. The counters in the obtained locations are checked. If all counters have a non-zero value, then the item is present within the hash buckets corresponding to these counters. However, the hash bucket corresponding to the counter having the least

value is maybe incorrect. Hence, a tree is used to store the information about which hash bucket has stored the item. During delete operation, after obtaining the location of the counters. All counters are checked; if all have non-zero values, then the value is decremented by 1. The tree is queried to determine the location of the item in the hash table. Then that item is deleted from the bucket and the tree. The CBF is converted into a standard Bloom Filter during broadcasting of the cache information. If the CBF counter is non-zero, then the corresponding standard Bloom Filter cell is set to 1; otherwise, to 0. When a false positive situation occurs in the caching router, it sends a packet to the sender to report the false positive situation. Hence, the sender node maintains two tables for each caching router, cache and false positive. The cache table stores the Bloom Filter with its IP address. The false positive table stores the packets not present in the cache with its IP address.

Co-operative Virtual Network Embedding (COVE) [12] is a virtual network embedding algorithm that uses Bloom Filter, learning technology, and topology decomposition. The network has a central node, and other nodes are substrate nodes. The central node monitors and controls the network. The substrate nodes are autonomous and have embedding solutions. It synchronizes with other substrate nodes. COVE implements learning technology in substrate nodes to make them capable of performing mapping processes. The mapping processes are mapping of virtual nodes to physical nodes and mapping of virtual links to physical links. The central node receives and dispatches Virtual Network (VN) requests, mapping solutions, policies, and protocols. Substrate nodes determine resources available with other substrate nodes using the information provided by the central node. Using this knowledge, the substrate node makes a decision to complete the mapping process. Bloom Filter stores the set of mapped nodes and piggybacked messages to other nodes for synchronization. When the central node receives a VN request, the InP decides whether to accept or to wait. InP is responsible for deploying, managing, and maintaining the resources. InP accepts the VN request if the substrate nodes have the resources. If the resource is insufficient, InP waits for the next mapping period for some fixed time period. In the mapping process, a mapping message is transmitted in the network. The hop field in the mapping message is initialized to the number of nodes in the network. When a substrate node receives the mapping message, it accepts a virtual node. The substrate node adds its IP address into Bloom Filter. The IP address of the substrate node is hashed by k hash functions. These k locations are set to 1. The hop field is decremented by 1, and the count field is incremented by 1. In case the count field becomes a threshold value before the hop field is 0, then the mapping message is sent to the central node. It means the embedding is terminated. The central node informs the selected substrate nodes to release their resources. After completing the mapping of all virtual nodes, the central node transmits a new mapping message and piggybacking the Bloom filter to all substrate nodes.

10.4 Routing

Routing is the process of determining the path in the network. This is an essential task for traffic management. The packet needs to pass nodes called routers that direct the packets towards the direction of the destination. The efficiency of routing determines the performance of the network. Hence, it is essential for the router to process the packet quickly. Broadly, the packet transmission can be classified based by the number of senders and destination as unicast, broadcast, multicast, and anycast. The unicast delivers the packet to a single destination node. The broadcast delivers the packet to all nodes except the sender in the network. The multicast delivers the packet to many nodes, but not all, in the network. The anycast is delivering the packet to any one node in the network. The broadcast and multicast require more resources and computation.

Multicast is the process of transmitting data from a single source to many destinations. Hence, lots of resources are consumed, and many network nodes are loaded with processing. Multicast multiplexes a shared multicast tree to complete the delivery of packets. One solution is embedding the whole multicast tree in the packet header, which the routers can use to determine the next node. However, one big issue is the compression of the whole multicast tree into a small header. In such cases, Bloom Filter is a good solution. The information of the multicast tree is saved in the Bloom Filter, and the small Bloom Filter is embedded into the packet header. This solution is used in many applications of networking such as multisite Virtual private network (VPN) [13] and datacenters [14]. This section presents reviews on some techniques based on Bloom Filter for routing. It summarizes the features provided by the state-of-the-art routing algorithms and their limitations in Table 10.3.

Cheng et al. [15] proposed utilization of multiple Bloom Filters for multicast source routing. It proposed two schemes, Homo-MSRM and Hete-MSRM. MSRM is multicast source routing with multiple Bloom Filters. The whole multicast tree is partitioned into subtrees, and a Bloom Filter is assigned to each subtree. In Homo-MSRM, every

TABLE 10.3 Features and Limitations of Routing Techniques.

Technique	Features	Limitations
Cheng et al. [15]	• Assigning a subtree to a Bloom Filter reduces the space occupancy of Bloom Filter • Hete-MSRM requires less memory compared to Homo-MSRM • Hete-MSRM has less collisions compared to Homo-MSRM	• False Positives lead to misforwarding • In case of small subtree, fixed-length Bloom Filters in Homo-MSRM result in inefficient space usage • In case of large subtree, fixed-length Bloom Filters in Homo-MSRM result in high FPP • Misforwarding depends on FPp of Bloom Filter and network topology • Bloom Filter size in Homo-MSRM depends on the number of nodes in the largest subtree • Implements multiple Bloom Filters
Zhang et al. [16]	• Deterministic annealing method increases performance • Clustering helps in quick service recovery • Bloom Filter eliminates searching in some children nodes within a cluster • Searching in tree-like structure takes logarithmic time	• Maintains multiple Bloom Filters • Service search involves query operation to multiple Bloom Filters • Increase in services affects the search performance • Crossing a threshold value for node connectivity results in network rearrangement • Initial setup of the network takes time due to clustering
XBF [17]	• Scheme supports intra-domain multicast • Compression of zBF is easy because it is sparse • Determining the next node is faster due to Bloom Filter lookup operation	• Bloom Filter header overhead depends on the size of multicast tree • zBF is a sparse matrix • Bloom Filter is not secure • Number of lookup operations in Bloom Filter by popper switch depends on the number of incident sub-networks

Bloom Filter is homogeneous in terms of size and other parameters such as a number of hash functions. The Bloom Filter size depends on the number of nodes in the largest subtree. It leads to memory wastage in the case of small subtrees. In Hete-MSRM, every Bloom Filter is heterogeneous for efficient memory usage. The size of the Bloom Filter is different, and it is determined using the subtree size. When a packet is transmitted from any node within a subtree, the corresponding Bloom Filter is loaded into the packet.

Zhang et al. [16] proposed a routing mechanism for service discovery with enhanced scalability and performance. The service network has grouped the services into clusters where the root node of each cluster is analogous to the routers. Then deterministic annealing technique is applied to the clusters to search for a stable state. The services are arranged as a tree-like structure. Each node in the tree maintains a Bloom Filter, and the service name is inserted into the Bloom Filter. The Bloom Filter size varies based on the layer of the subtree. If the cluster has many leaf nodes, semantic match-making methods are implemented instead of Bloom Filter. During a service request, the Bloom Filter of the root node is searched. If found, then the Bloom Filter of the children nodes is searched. When the leaf nodes cross a threshold value, the whole cluster is partitioned into multiple clusters with fewer leaves. Clustering is reducing the service discovery time, however, repetitively, some clusters will be large, and again, the partitioning of the clustering algorithm needs to be executed.

Extensible Bloom Filter (XBF) [17] is a framing and forwarding technique. The network is divided into sub-networks where each sub-network contains an equal number of links, i.e., 256. The Bloom Filter stores the sub-network link. The Bloom Filter is of fixed length, i.e., 256 bits. Each bit of Bloom Filter corresponds to a link. This removes false positive issues. The packet header consists of two parts, iBF and zBF. The iBF has the Bloom Filter that is forwarded to other sub-network. The zBF has all the Bloom Filters of the sub-network which the packet has traveled. The zBF remains unchanged, whereas iBF content changes dynamically. This network has three main components: topology manager, forwarder switches, and popper switches. The topology manager has the following responsibilities: (a) division of a network such that each sub-network has at most 256 directed links, (b) creation of Bloom Filter, and (c) assignment of each link to a bit of Bloom Filter. A switch is called a popper switch if it connects links between two sub-networks. The forward switch is the switch where all links are within the sub-network. It performs basic operations for Bloom Filter forwarding within the sub-network. At the same time, the popper switch performs Bloom Filter forwarding operation between sub-networks. Each node has a Bloom Filter where its bits are set to 1 if a path exists, which the respective bit location represents. The topology manager creates an in-packet Bloom Filter where its contents are obtained by OR-ing the link identifiers of all the packet receivers, i.e., the destination node and the intermediate nodes. Referring to the bit values of the Bloom Filter, the packet traverses the network to

reach the destination. When a node receives the packet, it extracts the Bloom Filter and performs a lookup operation to determine the link the packet needs to take next. When a packet's destination is a node of another sub-network, then the popper switch copies Bloom Filter from zBF to iBF. The popper switch performs a lookup operation of Bloom Filter to determine the destination sub-network. Hence, if a popper switch is located in more than two sub-networks, then the number of lookup operations performed depends on the number of sub-networks.

10.5 Searching

The router stores information in the routing table to determine the next node the router needs to forward the packet. The routing table contains the network ID, subnet mask, next-hop, outgoing interface, and metric. The routing table is stored in the RAM of the device. The packet's destination address needs to be checked with the entries of the routing table. This matching is called the longest prefix matching. It is called so because the longest number of bits of the address matches the entries in the routing table. Bloom Filter takes constant time for a lookup operation; hence, Bloom Filter is also a good solution for longest prefix matching. In this section, some techniques based on Bloom Filter for packet searching and prefix matching are reviewed.

Lim and Byun [18] proposed an algorithm for enhancing the performance of leaf-pushing area-based quad-tree (AQT). Packets are classified based on some set of rules. There are many such rules and they have big content; hence, they cannot be stored in on-chip memory. However, storing in off-chip memory increases the time complexity for accessing the rules. The algorithm uses Bloom Filter and a hash table for retrieving required rules for each received packet. AQT is a tree where some nodes store the rules. The tree is searched for selecting the appropriate rule. AQT is constructed by using a code word where this word is obtained by combining the source and destination prefix fields. In leaf-pushing AQT, the internal rule nodes are pushed to the leaf. This improves the searching time. In this algorithm, each node is a Bloom Filter. All rule nodes and internal nodes are stored in a hash table. The rules are stored in the rule database using a linked list. The linked list is connected following the order of decreasing priority of rules. During a query, the Bloom Filter at the root is queried. If the address is absent, then the rule is absent. Otherwise, its children nodes (or Bloom Filters) are checked. When a Bloom Filter returns $False$, then its sub-tree is not checked further. When a false positive occurs, the Bloom Filter tree is backtracked, and that level's rules are searched in the hash table. The backtracking continues till the rule is not found.

10.5.1 Prefix matching

Bloom Filter plays vital role in longest prefix matching and Table 10.4 summarizes the features and their limitations.

Byun et al. [19] proposed implementation of vectored-Bloom Filter (VBF) in field programmable gate array (FPGA) for IP address lookup. The technique performs the longest prefix matching for IP address lookup. In FPGA, VBF is deployed in on-chip, whereas hash table is in off-chip memory. The VBF helps to reduce the frequency of accessing the hash table. Each cell of VBF has p bits, where each bit corresponds to an output port. A unique term is defined, which refers to a set of possible prefix lengths in a routing table. When an IP address is received, the longest prefix length among the unique lengths is queried to VBF. The v vectors provide v indexes in VBF. The AND operation is performed among the vectors to obtain a resultant vector. If only one bit is set to 1 in the resultant vector, then VDF returns $True$. If all bits in the resultant vector are zero, then VDF is queried again with a reduced prefix length, i.e., the next longest prefix length in unique length. In case all multiple bits in the resultant vector are 1, then the hash table in off-chip memory is looked at to determine the port number. If the hash table returns the output port, then the procedure is terminated. Otherwise, the VBF is queried with the next longest prefix length in a unique length. If the prefix length is the shortest, the technique returns the currently remembered best matching port. VBF is faster as it is located in the on-chip memory; however, repetitively querying VBF is reducing the advantages gained due to VBF.

GRv6 [20] is a GPU-accelerated software router for IP lookup in IPV6. It performs the longest prefix matching on IPv6 addresses. GRv6 maintains multiple (say, b) standard Bloom Filters on-chip and a CBF in off-chip memory. During the insertion operation of an IP address, the prefix is inserted into the CBF. The prefix is hashed by k hash functions, and hashed values provide the location of cells in CBF. If the data of the cell is 0, then it is set to 1. Otherwise, if the data of the cell is 1, then the counter of the cell is incremented. The length of CBF and Bloom Filters

TABLE 10.4 Features and Limitations of Prefix Matching Techniques.

Technique	Features	Limitations
Lim and Byun [18]	• Rule searching is faster • Bloom Filter tree decreases the time complexity of the rule searching	• False positive response leads to tree backtracking • False positive response increases the number of hash table searching instances • Bloom Filter tree maintains many Bloom Filters • Number of rule comparisons depends on tree characteristics and set types
Byun et al. [19]	• Each cell has p output ports • VBF is implemented in on-chip memory • Hash table is removed from basic accelerators due to rare usage	• When VBF returns indeterminable result, hash table is searched • VDF is accessed multiple times when the IP address is not the longest prefix length
GRv6 [20]	• Supports dynamic prefix inserting • H3 hash family is easy for hardware implementation • Using CBF helps in deletion in standard Bloom Filters • Technique more appropriate for more true negative cases	• True result of Bloom Filter is again checked in off-chip memory • Gives bad throughput with a large number of packets • Scheme requires lots of data structures • True positive cases are checked with a hash table

is the same. When the data of a cell is set to 1, then its corresponding cell in Bloom Filters in on-chip memory is also set to 1. A similar procedure is followed for the delete operation. If a data cell in CBF is reset to 0, its corresponding cell in Bloom Filters in on-chip memory is reset to 0. The hash function in Bloom Filter is the H3 family [21]. Let i be an IP address, and suppose H is a boolean matrix. The i is hashed as $i(1)\&h(1) \oplus i(2)\&h(2) \oplus \cdots \oplus i(l)\&h(l)$ where $i(j)$ is the jth bit of i, $h(j)$ is jth row of H, & is AND operation, and $\oplus$ is XOR operation. This hash value gives the location of a cell in the Bloom Filter, which is set to 1. When an IP address ip is received, that address is checked in Bloom Filter. The ip string is divided into b variable-length sub-strings. One sub-string is checked in one Bloom Filter in parallel. The first Bloom Filter checks the first bit of ip. The second Bloom Filter checks the first two-bit sub-string of ip. Similarly, the jth Bloom Filter checks the j-bit sub-string of ip. If all Bloom Filters return $True$, then the IP address is present; otherwise, it is absent. In case Bloom Filters conclude that the IP address is present, it is again checked in off-chip memory due to the possibility of a false positive response. Hence, this technique is more appropriate for a larger number of true negative cases.

10.6 Discussion

In network communication, there are many components where Bloom Filter can be used to reduce the computation and cost overhead. Traffic management is monitoring and controlling the network traffic. However, network traffic is a difficult task to manage due to its volume, which is increasing exponentially day by day. One of the most important advantages of Bloom Filter is its faster filtering capacity. With its simple data structure, Bloom Filter helps in the filtering of the traffic quickly. Filtering reduces the number of packets, hence computation by various nodes. Thus, Bloom Filter reduces the computational cost. The membership checking feature of Bloom Filter helps in determining traffic statistics, for example, the type of network flow. Determining the type of flow helps the devices in making some preparation for proper handling of the huge incoming traffic. Thus, Bloom Filter helps in handling the network.

Many techniques also prefer embedding Bloom Filter in the packet, for example, Extensible Bloom Filter (XBF) [17]. This helps in the faster transmission of the packet. Furthermore, a small-sized Bloom Filter stores a lot of information. Hence, Bloom Filter can be embedded in the packet without making the packet bulky. Routing requires checking the routing table to determine the next node of the packet. However, searching the routing table is also an overhead for packet transmission. Bloom Filter can be of great help in broadcast and multicast applications. In the case of multicast, many nodes are not eligible for receiving some packet. In such cases, deploying Bloom Filter in the nodes helps determine the node's eligibility quickly. It helps in the faster handling of the packets.

Table 10.5 lists a comparison of the techniques reviewed in the article based on various parameters. From the table, we can notice that most of the techniques use standard Bloom Filter. Standard Bloom Filter is faster; however, it has the most severe false positive and false negative issues. There are many variants that are faster and contain fewer issues compared to the standard Bloom Filter. These techniques should deploy these variants because the traffic

TABLE 10.5 Additional details regarding the network technique: Bloom Filter, name of Bloom Filter variant used in the technique; Reduce FP, technique used to reduce false positives; Remark, purpose of the technique.

Technique	Network Type	Bloom Filter	Reduce FP	Remark
Zhou et al. [1]	Centralized network	Counting	No	Determines Event frequency
BF-LRU [2]	IP	Standard	No	Determines elephant flow
FlexSketch-Mon [3]	IP	Standard	No	Traffic Monitoring
CBFSketch [4]	Data Stream	Counting	No	Traffic statistics of data stream
Bloomtime [5]	IP	Standard	No	Calculates mean and variance of packet inter-arrival
PBF [6]	IP	Standard, Probability Bloom Filter	No	Calculates the frequency of the traffic flow
Terzenidis et al. [7]	Optically-enabled node	Standard	No	Packet forwarding
HyperSight [8]	IP	Standard	No	Determines packet behavior
Pache [10]	IP	Counting	False positive table	Cache management
TBF [9]	IP	Time-out	No	Packet sampling
COVE [12]	Centralized	Standard	No	Broadcasting the mapping nodes
Lim and Byun [18]	IP	Standard	No	Packet classification based on rules
Byun et al. [19]	IP	Vectored-Bloom filter	No	Field programmable gate array (FPGA)
GRv6 [20]	IP	Standard, Counting	Verifies with the hash table	Prefix checking
Cheng et al. [15]	IP	Standard	Multiple Bloom Filters	Multicast source routing
Zhang et al. [16]	Services nodes classified into clusters	Standard	No	Service discovery
XBF [17]	IP (Network classified into sub-networks)	Standard	Each bit represent a link	Routing

is huge in practical scenarios, and standard Bloom Filter is incapable of handling it. Moreover, these techniques avoid delete operations to eliminate the false negative issue. The standard Bloom Filter has low FPP; however, huge traffic saturates the Bloom Filter, which increases FPP. Also, the technique avoids the deletion operation; hence, the saturation of the Bloom Filter cannot be reduced by deletion. Thus, the reasons for using standard Bloom Filter, i.e., low FPP and no delete operations, become a big issue due to the huge network traffic.

10.7 Conclusion

The simple architecture of Bloom Filter and constant time complexity operations help in faster processing of data. Hence, Bloom Filter efficiently filters the network traffic. Moreover, this simple data structure helps in determining the type of network flow. This is of great help to prepare for the possible huge traffic. The packet is the carrier of the message which traverses through the network. Bloom Filter helps in packet management, which greatly reduces packet computation. A small-sized Bloom Filter stores lots of information. This helps in embedding the Bloom Filter in the packet and improves the routing. Furthermore, prefix matching algorithms have high time complexity. However, using Bloom Filter reduces the time complexity to a large extent. In this chapter, we have illustrated how Bloom Filter is improving the efficiency and performance of network techniques and algorithms.

References

[1] Y. Zhou, Y. Zhou, S. Chen, Threshold-based widespread event detection, in: 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), 2019, pp. 399–408.

[2] Z. Zhang, B. Wang, J. Lan, Identifying elephant flows in Internet backbone traffic with Bloom filters and LRU, Comput. Commun. 61 (2015) 70–78, https://doi.org/10.1016/j.comcom.2014.12.003.

[3] T. Wellem, Y.-K. Lai, C.-Y. Huang, W.-Y. Chung, A flexible sketch-based network traffic monitoring infrastructure, IEEE Access 7 (2019) 92476–92498, https://doi.org/10.1109/ACCESS.2019.2927863.

[4] H. Zhu, Y. Zhang, L. Zhang, G. He, L. Liu, Cbfsketch: a scalable sketch framework for high speed network, in: 2019 Seventh International Conference on Advanced Cloud and Big Data (CBD), 2019, pp. 357–362.

[5] R.D. Pacífico, L.B. Silva, G.R. Coelho, P.G. Silva, A.B. Vieira, M.A. Vieira, Í.F. Cunha, L.F. Vieira, J.A. Nacif, Bloomtime: space-efficient stateful tracking of time-dependent network performance metrics, Telecommun. Syst. (2020) 1–23.

[6] S. Xiong, Y. Yao, M. Berry, H. Qi, Q. Cao, Frequent traffic flow identification through probabilistic bloom filter and its GPU-based acceleration, J. Netw. Comput. Appl. 87 (2017) 60–72, https://doi.org/10.1016/j.jnca.2017.03.006.

[7] N. Terzenidis, M. Moralis-Pegios, C. Vagionas, S. Pitris, E. Chatzianagnostou, P. Maniotis, D. Syrivelis, L. Tassiulas, A. Miliou, N. Pleros, K. Vyrsokinos, Optically-enabled Bloom filter label forwarding using a silicon photonic switching matrix, J. Lightwave Technol. 35 (21) (2017) 4758–4765, https://doi.org/10.1109/JLT.2017.2760013.

[8] Y. Zhou, J. Bi, T. Yang, K. Gao, J. Cao, D. Zhang, Y. Wang, C. Zhang, Hypersight: towards scalable, high-coverage, and dynamic network monitoring queries, IEEE J. Sel. Areas Commun. 38 (6) (2020) 1147–1160, https://doi.org/10.1109/JSAC.2020.2986690.

[9] S. Kong, T. He, X. Shao, C. An, X. Li, Time-out Bloom filter: a new sampling method for recording more flows, in: International Conference on Information Networking, Springer, 2006, pp. 590–599.

[10] T. Chen, X. Gao, T. Liao, G. Chen, Pache: a packet management scheme of cache in data center networks, IEEE Trans. Parallel Distrib. Syst. 31 (2) (2020) 253–265, https://doi.org/10.1109/TPDS.2019.2931905.

[11] H. Dai, J. Lu, Y. Wang, T. Pan, B. Liu, Bfast: high-speed and memory-efficient approach for NDN forwarding engine, IEEE/ACM Trans. Netw. 25 (2) (2017) 1235–1248, https://doi.org/10.1109/TNET.2016.2623379.

[12] M. Feng, J. Liao, S. Qing, T. Li, J. Wang, Cove: co-operative virtual network embedding for network virtualization, J. Netw. Syst. Manag. 26 (1) (2018) 79–107.

[13] R.B. Martínez-Aguilar, G.M. Fernández, Implementation of stateless routing mechanisms for multicast traffic on NetFPGA card, in: IEEE Colombian Conference on Communication and Computing (IEEE COLCOM 2015), IEEE, 2015, pp. 1–5.

[14] C.A. Macapuna, C.E. Rothenberg, M.F. Maurício, In-packet Bloom filter based data center networking with distributed openflow controllers, in: 2010 IEEE Globecom Workshops, IEEE, 2010, pp. 584–588.

[15] G. Cheng, D. Guo, L. Luo, Y. Qin, Optimization of multicast source-routing based on Bloom filter, IEEE Commun. Lett. 22 (4) (2018) 700–703, https://doi.org/10.1109/LCOMM.2018.2798668.

[16] J. Zhang, R. Shi, W. Wang, S. Lu, Y. Bai, Q. Bao, T.J. Lee, K. Nagaraja, N. Radia, A Bloom filter-powered technique supporting scalable semantic service discovery in service networks, in: 2016 IEEE International Conference on Web Services (ICWS), 2016, pp. 81–90.

[17] M. Antikainen, L. Wang, D. Trossen, A. Sathiaseelan, XBF: scaling up Bloom-filter-based source routing, arXiv:1602.05853.

[18] H. Lim, H.Y. Byun, Packet classification using a Bloom filter in a leaf-pushing area-based quad-tree, in: 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2015, pp. 183–184.

[19] H. Byun, Q. Li, H. Lim, Vectored-Bloom filter implemented on FPGA for IP address lookup, in: 2019 International Conference on Electronics, Information, and Communication (ICEIC), 2019, pp. 1–4.

[20] F. Lin, G. Wang, J. Zhou, S. Zhang, X. Yao, High-performance IPv6 address lookup in GPU-accelerated software routers, J. Netw. Comput. Appl. 74 (2016) 1–10, https://doi.org/10.1016/j.jnca.2016.08.004.

[21] M. Ramakrishna, E. Fu, E. Bahcekapili, Efficient hardware hashing functions for high performance computers, IEEE Trans. Comput. 46 (12) (1997) 1378–1381.

This page intentionally left blank

C H A P T E R

11

Bloom Filter for named-data networking

11.1 Introduction

Nowadays, along with text and audio, the videos are transmitted using the Internet. It is achieved by the advancement of packet switching. With the advancement of networking technology, the nature of applications is changing, and the number of users are increasing, along with the demand of the users. However, IP addresses are unable to satisfy the user requirements. IPv4 has serious scalability issues. Only IP is present in the network layer of the OSI communication model. This does not permit any modification of existing functionality nor appends any new functionality [1]. Moreover, IP follows a spanning tree for the forwarding of packets. All these reasons have made IP unsuitable for today's network [1]. The current user requirement is "what data" without the knowledge of "where data". Hence, the current Internet has ingressed into a new era of networking called Content-Centric communication [2].

Content-Centric communication has ubiquitous inter-connectivity and a wide range of services [3]. It supports information-intensive businesses (e.g., financial, banking, and traveling services) to expand their business using the Internet. Many projects are proposed to design the content-based future Internet paradigms [4–6]. One of the projects is Named Data Networking (NDN) [7–9]. NDN was proposed to achieve Information-Centric Networking (ICN) [10,2]. NDN is defined as an instance of the more general network research direction of ICN [7]. NDN has an efficient design and simple communication model. NDN can smoothly handle variable length and location-independent names, content searching, and content retrieval. Table 11.1 highlights the comparison between the NDN and IP.

This chapter explores the role of the Bloom Filter in NDN. Section 11.2 provides a brief introduction on its architecture. Section 11.3 presents the implementation of Bloom Filter in NDN packets for smooth forwarding in the network. NDN maintains many data structures for fast processing of the content. The data structures are content store, pending interest table, and forwarding base. The content store is the buffer of the router. Section 11.4 highlights the role of Bloom Filter in enhancing the performance of CS. Pending interest table stores the pending interest packet. Section 11.5 illustrates the role of Bloom Filter in improving the storage and searching in PIT. Forwarding base stores the reachable node information. Section 11.6 presents the contribution of Bloom Filter in improving the performance of FIB. Bloom Filter also helps in improving the security of NDN, which is discussed in Section 11.7. Section 11.8 provides the discussion on the role, issues and limitations of Bloom Filter in NDN and its data structures. All sections include the reviews of Bloom Filter-based techniques. Furthermore, every section has a table to highlight the features and limitations of the reviewed techniques. Finally, Section 11.9 concludes the chapter.

11.2 Named data networking

NDN is conceptualized from the Information-Centric Networks (ICN) [11,10]. The sender digitally signs every named content, which helps in storage in cache for future. It saves bandwidth and efficiently utilizes multiple network interfaces [12]. Moreover, it is resilient [13,14] while maintaining good packet delivery performance. It reduces congestion by balancing the packet flow in every hop, which prevents the execution of additional congestion control algorithms between the network path [1]. It provides high-level security such as origin authentication, integrity and evaluation of the importance of routing information.

In the NDN network, a consumer is a node which is requesting data, and the producer is a node that generates the requested data. An interest packet is a packet that has the details of the requesting data, and a data packet is a packet generated by the producer that contains the requested data. An interest packet passes through three data structures: Content Store (CS), Pending Interest Table (PIT), and Forwarding Information Base (FIB). CS is the

 Copyright © 2023 Elsevier Inc. All rights reserved.

router's buffer memory that caches data. An interest packet first passes through CS. Matching data is searched in CS. If found, the data packet is forwarded to the consumer. Otherwise, the interest packet is forwarded to PIT, which saves the pending interests. PIT saves the interest name and the interface which forwarded the interest packet. Then the interest packet is forwarded to the next-hop till it reaches the producer. After receiving the interest packet, the producer generates the data packet. The data packet follows the path in reverse order to reach the consumer. PIT provides the interface information. If the interface information is found in PIT, then it is deleted from PIT, and the data packet is forwarded to the consumer. FIB is responsible for storing the next-hop and other related information of every reachable node. After passing PIT, the interest packet is forwarded to FIB, which directs the interest packet toward the next hop. Table 11.1 presents comparison of various features among CS, PIT, and FIB.

TABLE 11.1 Comparison of various features among CS, PIT and FIB. In data structure feature, widely used technique is mentioned. N/A: not applicable.

Feature	CS	PIT	FIB
Frequency of read operation	High	High	High
Frequency of write operation	High	High	Low
Size of data structure	Different and based on edge and routers	Different and based on edge and routers	Same
Algorithm	Cache Replacement policy	Timeout operation	Forwarding Strategy
Matching Algorithm (Interest packet)	All Sub-Name Matching	Exact Name Matching	Longest Name Prefix Matching
Matching Algorithm (Data packet)	Exact Name Matching	All Name Prefix Matching	N/A
Data structure	Skip List	Bloom Filter	Tree
Issue	Fast name lookup	High memory access frequency	Memory consumption

11.3 Named data networking packet

NDN communication initiates when consumers request specific data by generating an interest packet. The interest packet provides the content name to recognize the data. Fig. 11.1 (left) indicates the segments of an interest packet. Fig. 11.2 illustrates the traversal of interest packet through NDN data structures where the box indicates the router. Content name is network independent, which helps to retrieve content. When the interest packet enters the router, first, the content name is searched in CS. If the content name is found, then CS forwards the data packet to the consumer. Otherwise, the interest packet is forwarded to PIT. PIT stores the required information regarding the content and the consumer. Then it is forwarded to FIB, which provides the next-hop information for traveling in the network till it reaches the producer. Table 11.2 illustrates the features and limitations of the Bloom Filter-based techniques for efficient packet delivery in NDN.

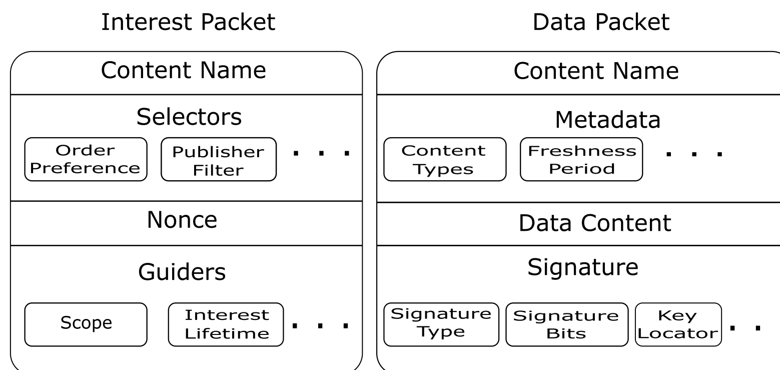


FIGURE 11.1 Packet Architecture of Named Data Networking.

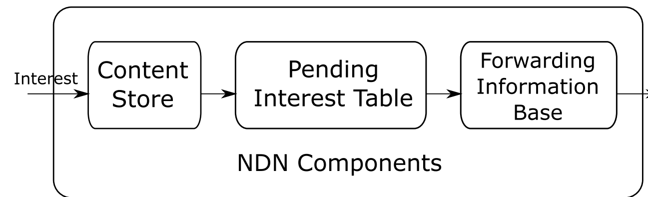


FIGURE 11.2 Components of Named Data networking.

11.3.1 Bloom Filter in the packet

Summary packet [15,16] is a packet that contains a Bloom Filter that stores a cache summary of a node. It enhances cache efficiency in routers by helping in taking cache decisions. It implements stateful Bloom Filter [17] which supports association queries that recognize the set to which an item belongs. The nodes store the summary packets in a data structure called the summary store. Bloom Filters are exchanged with neighboring nodes by the summary packet. After receiving a data packet, it is checked in the cache summaries of neighbor nodes. In case the data packet is present with a neighbor node, the packet is not cached. If absent, content is cached in CS and updates its own summary. After receiving an Interest packet, the summary is searched with Bloom Filter. If present in Bloom Filter, then the algorithm searches in its cache. Again, if it is present, then the requested data packet is forwarded to the consumer. In the scenario where the face of the matching summary and incoming interest packet is different, the interest packet is passed to the face of a matching summary. The scenario where both are matching indicates a case of false positives. In case of false positive or no matching summary, after referring to FIB, the interest packet is forwarded to the next node. One Bloom Filter is maintained for each face. In total, a summary store maintains the number of faces plus one Bloom Filter. Another Bloom Filter is for its cache.

Network Coding (NC) [18] is a technique that embeds Bloom Filter in interest and data packets. Bloom Filter stores the ID of the interest packet. Bloom Filter helps in taking various decisions after receiving the Interest packet: aggregation of the interest packet with other Interests stored in PIT, utilization of data packets stored in CS, and insertion of new PIT entry and forward to the neighbor node. Bloom Filter is modified based on some rules which are applied as the Bloom Filter moves toward the producer of the requested data. Interests are merged when the intersection between received interest's Bloom Filter and existing same content name interest is empty. If the intersection is not empty, then both interests are received by the producer. Bloom Filter of the interest packet is stored in the data packet and forwarded to the producer. After receiving a data packet, the node consumes the interest whose Bloom Filter stores a subset of the consumer IDs stored in the data packet's Bloom Filter. A node removes its name from the Bloom Filter to prevent re-transmission to the same consumer.

Watano et al. [19] proposed a reroute algorithm to forward interest packets. The algorithm is based on the degree of similarity of cached contents among each neighboring content router (CR). The similarity is checked by comparing the similarity of the Bloom Filters among chosen candidate CRs. In case the original route is lost, Bloom Filter helps to select an adequate alternate route with similar contents. CR consists of a CS, PIT, FIB, and a Bloom Filter corresponding to the faces connected to the CR. CRs exchange their Bloom Filter value using interest/data packet communications. The calculation of similarity is between the initial next CR hop according to its FIB entry and the candidate CR. The algorithm considers the $EXOR$ of the Bloom Filters. Similarity rate is the sum of bits calculated as 1. More number of 1s represents lesser similarity. The algorithm does not perform well when the link downtime is less. The reason is appropriate alternate next CR hop cannot be determined because Bloom Filters does not reflect the initial similarity differences. In case of link disconnection or inactive state of the next CR hop, the CR cannot reach the next hop and cannot forward an Interest. This results in degradation of performance and increases network traffic.

11.3.2 Content discovery

Bloom Filter-based Routing (BFR) [20,21] is a fully distributed content discovery technique. The origin server uses Bloom Filter to advertise its content objects. BFR follows three phases: representation, advertisement, FIB population and content retrieval. In the representation phase, the origin servers construct a Bloom Filter and insert the content objects. The content object name and name prefixes are inserted into the Bloom Filter. The advertisement phase forwards the Bloom Filter to advertise the content objects. Content Advertisement Interest (CAI) packet consists of

TABLE 11.2 Features and Limitations of Bloom Filter-based techniques of Named Data Networking Packet.

Technique	Features	Limitations
Summary packet [15,16]	• A packet that contains a Bloom Filter stores a cache summary of a node • Implements stateful Bloom Filter [17] • Nodes store the summary packets in summary store • If a data packet is present with a neighbor node then the packet is not cached • Copes with link failure, dynamic network topology, and congestion in some links • Changes in the request pattern does not affect content diversity	• Sequential query operations on Bloom Filters increases time complexity • Change in the request pattern does not affect content diversity • Sequential query operations on Bloom Filters increase time complexity • Maintains multiple Bloom Filters • Maintains one Bloom Filter for each face, hence many faces increase the memory requirements for the Bloom Filters
BFR [20,21]	• Fully distributed • Content oriented • Robust to topology change • Does not store topology information • Resilient to link failure • Delay is less • Takes less congested route	• An interest reaches a wrong origin server due to the false positive response • Transmits CAI through multiple paths to reach the origin server, increases traffic congestion • Takes longer route • Less efficient compared to pull-based BFR • Total communication overhead depends on the content universe size • Increasing content advertisement refresh rate increases communication overhead • Memory space increases with the increase in content universe
Watano et al. [19]	• Uses a reroute algorithm to forward interest packets • Is based on the degree of similarity of cached contents among each neighboring content routers • In case the original route is lost, Bloom Filter helps to select an alternate adequate route with similar contents • Cache hit ratio is higher than for the conventional method • Essential when there is an unwanted loss of connection • Space efficient	• Cache hit ratio is unsatisfactory for early link down-times • Degrades of performance and increases network traffic when CR cannot reach the next hop due to link disconnection or inoperative state of the next CR hop • Does not perform well when the link down-time is less • Implements standard Bloom Filter
Network Coding (NC) [18]	• Delivers scalable video • Packet delivery rate is based on the popularity of the content • Maximizes average video quality • Bloom Filter supports deploying of network coding in NDN • Decision of Bloom Filter helps in deciding the packet movement	• Choosing the network coding coefficients randomly leads to a decrease in video quality • Conventional Bloom Filter gives high false positive probability
Pull-based BFR [22]	• Advertises only on-demand content objects • Bandwidth is saved • Other network nodes require less memory to save content advertisement information • Scalable • Memory consumption is less compared to push-based BFR	• Server is unable to know the demanded content objects a priori • Union of maximum capacity Bloom Filters gives very high false positive probability • Due to false positive response, a server advertises the content which is not in demand • Increase in forwarding rate of routing messages increases content advertisement overhead • Increasing content advertisement refresh rate increases communication overhead
Marandi et al. [23]	• A feedback-based cooperative protocol for content discovery • Implements Interest Bloom Filter • Requires less bandwidth resources • Does not perform content advertisement • Bloom Filters are regularly updated for faster decoding of variables	• Delete operation is performed on the Bloom Filter • Presence of false negatives • Implements standard Bloom Filter

content advertisement Bloom Filter. CAI packets are sent to all neighbors. It is broadcast till all nodes receive the packet.

Pull-based BFR [22] algorithm advertises only on-demand content objects. This algorithm is an improvement of BFR [20]. It is scalable, saves bandwidth, and has fewer memory requirements compared to BFR. It has phases, obtaining demanded content objects and advertisement. In case there is no FIB entry for an interest against its content name and no Bloom Filter has content advertisement, the interest is not sent and marked as pending. Consumer constructs a Bloom Filter and inserts the content file name and name prefixes to request the server to pull the content advertisement. Then the consumer constructs a Content Advertisement Request (CAR) message and embeds the Bloom Filter. Router merges the same CAR received from different consumers and forwards in a different face. All nodes construct the same size Bloom Filter and execute the same hash functions for the efficient union of Bloom Filters. Saturated Bloom Filters are not united because it sets all slots to 1.

Marandi et al. [23] presents a feedback-based cooperative protocol for content discovery. It implements Interest Bloom Filter (IBF). A client constructs an IBF and inserts the name of the data message. IBF is embedded into an Aggregated Interest Message (AIM) and forwarded to a router. The router waits for a short time interval to receive other AIMs. The received AIMs are merged into a single IBF which is forwarded to the next router. The nodes also use Bloom Filter to send feedback information to its neighboring nodes. The information consists of a set of decoding variables and a set of decoded variables. Two Bloom Filters are constructed to insert each set. After receiving the information from other nodes, the node updates the state information of the respective nodes. The node constructs Feedback Interest Message (FIM), which is an interest message. FIM consists of the content and the two Bloom Filters.

11.4 Content store

The NDN router first forwards the interest packet to CS to search for matching data [24]. CS is the router's buffer memory to cache contents for future requests. The data structure of CS consists of an index and packet buffer. Index saves the content name, and the packet buffer is the cache. CS size is kept small, but its size depends on the router and the number of network edges. CS implements the All Sub-Name Matching (ASNM) name matching algorithm for content searching. ASNM searches for the content that has the initial name, same as the requested interest. After PIT decides the interface to forward the interest packet, CS determines whether to cache the content or not. CS executes the Exact Name Matching (ENM) algorithm for the data packet. ENM searches for a CS entry exactly the same as the data name. CS caches the data when there is no matching CS entry. CS has a high-frequency read and write operations. The absence of CS does not hinder the routing operations of the router. However, CS helps in storing popular contents. For example, CS is a great help in live streaming [25]. The main requirement of CS is an efficient cache replacement policy. To fulfill this purpose, CS implements a skip list. The linked list in the skip list maintains the order of data storage. Thus, a skip list helps in the replacement policy. One major limitation of skip lists is high query time complexity. In this regard, Bloom Filter is a better alternative for lower query time. The rest of the section presents the review of various Bloom Filter techniques to improve CS. Table 11.3 lists the features and limitations of these techniques.

Dai et al. [26] proposed a Bloom Filter-based technique to save popular content. The technique is proposed for CS to decrease the upstream traffic and enhance the cache hit ratio. It maintains many Bloom Filters, and each Bloom Filter saves content having a particular range of popularity. The content is separated into different ranges of popularity, and Bloom Filters are assigned to stored content of a particular range. Suppose, the content popularity ranges from 1 to s, and p is a content popularity such that $1 < p < s$. The first Bloom Filter stores content popularity ranging from 1 to p, the second Bloom Filter stores content having popularity within the range $p + 1$ to $2p$, and so on. The popularity of the first occurrence content is 1. With every occurrence, the popularity is incremented by 1. During insertion of new content, first, it is searched in all Bloom Filters. If found, then the popularity is incremented by 1. Otherwise, the content is inserted into the first Bloom Filter. When the popularity of content crosses a particular range, it is inserted into the next Bloom Filter. During query operation, the content is also searched in all Bloom Filters. If any Bloom Filter returns $True$, then the content is present. Otherwise, the content is absent.

Controller-Based Caching and Forwarding Scheme (CCFS) [27] is a technique to improve the forwarding decision by employing cooperative caching. In the technique, the network is partitioned into various domains, and each domain is controlled by a node (say, controller). Other nodes in the domain are regular routers. The controller takes the forwarding decisions based on content popularity. CCFS cache the content based on the content's popularity. It has a data structure to store its and neighbor's content summary, which is called Cache Information Base (CIB). Each

TABLE 11.3 Features and Limitations of Bloom Filter based techniques for Content Store.

Technique	Features	Limitations
Dai et al. [26]	• A Bloom Filter-based technique to save popular content • Proposed for CS and improves cache hit ratio • Reduces upstream traffic	• Maintains multiple Bloom Filters • Multiple presence of content in different Bloom Filter wastes memory • Exact content popularity cannot be determined from Bloom Filter • Bloom Filters storing lowest range popularity frequently becomes saturated • All Bloom Filters are searched in insertion and query operation
CCFS [27]	• A technique to improve the forwarding decision by employing cooperative caching • Improves in-network caching • Controller takes the forwarding decisions based on content popularity • Stable Bloom Filter reduces the FPP • Reduces communication load • Reduces bandwidth usage	• High server workload • Number of stable Bloom Filters depend on the number of interfaces • Monitoring using the controller may lead to single-point of failure • No backup technique is implemented in case of failure of the controller
FNR [29]	• Provides content delivery network (CDN)-like enhancement in NDN • Bloom Filter reduces storage and transmission overhead • Standard Bloom Filter reduces announcement traffic • Calculates the content popularity using Zipf distribution [31] • Local hashing table prevents repeated data location • Scalable • Lightweight	• Implements standard Bloom Filter • Maintaining multiple Bloom Filters occupies more memory • Centralized model may lead to a single point of failure • CS stores many data structures which increases memory overhead • CS saves one pointer to each content of Top-N Subset
BRCC [32]	• A technique to optimize the cache performance using collaborative caching • Implements sum-up counting Bloom Filter • Maintains caching information table to store the caching information • Reduces data content redundancy • Increases data content hit rate, diversity, and searching rate and accuracy • High cache hit ratio • Low Average Route Delay • Enhances data content matching rate • Decreases searching time	• Employs SCBF, which has a higher memory consumption • Does not improve Interest packet • Storing cache data of all neighbors makes the CIT saturated quickly • CIT performs frequency write operation to update the cache data periodically • Periodic update of cache data requires periodic flushing of SCBF

CIB entry saves node interface ID, type of the neighbor node, and a stable Bloom Filter [28]. The stable Bloom Filter is associated with the interface. It saves the node summary for quick searching. Every controller periodically sends the stable Bloom Filter to one-hop neighbors controllers. Stable Bloom Filter is transmitted using a specific interest packet. After receiving a new stable Bloom Filter, the controller updates the corresponding CIB entry.

Fetching the Nearest Replica (FNR) [29] is a technique to employ content delivery network-like [30] enhancement in NDN. It calculates the content popularity using Zipf distribution [31]. It partitioned the CS into two parts based on the content popularity, Top-N Subset and Heavy-Tailed Subset. Top-N Subset is a subset of size N which contains the top N popular content determined by using the Zipf law. This set satisfies the majority of requests. The contents of this set are inserted into a Bloom Filter. Bloom Filter reduces the storage and transmission overhead. Heavy-Tailed Subset has a big size and satisfies very few data requests. FNR deploys a Track Server (TS) for each Internet service provider. It handles the collection and synchronization of replica data. TS constructed the replica list by merging Top-N Subsets of all routers. Hence, the replica list is also the collection of all Bloom Filters. For the purpose of updating the Top-N Subset, the router requests TS to retrieve the latest Bloom Filter. Then TS updates its replica list and later synchronizes with every node. TS itself updates replica lists when the router does not send update requests for a long time. FNR implements a standard Bloom Filter to decrease announcement traffic. FNR is a centralized model which has the issue of the single point of failure.

Bloom Filter-based Request Node Collaboration (BRCC) [32] is a technique to optimize the cache performance. It implements a sum-up counting Bloom Filter (SCBF). BRCC maintains two data structures, SCBF and Caching Information Table (CIT). This technique is based on collaborative caching. Collaborative caching decreases the data content redundancy and enhances the data content diversity and hit rate. It also decreases data content duplication and makes it more diverse. It helps BRCC in having a high cache hit ratio and enhances First Hop Hit Ratio (FHHR). Moreover, it reduces the average hop count. The neighbor nodes periodically exchange their cached data content. CIT saves the caching information. BRCC only enhances data packet caching. It does not improve internet packet caching. BRCC has a modified data packet. The packet has a Caching Index (CI), caching age (CA), and Caching Remain Age (CRA). CI is a caching flag which is initially set to 0. CI is set to 1, indicating the data content is cached in the data packet. CA records the lifespan of data packets. CRA stores the remaining lifespan of the data packet. When a new data packet arrives with a saturated CS cache, then the packet is replaced with a packet having CRA value 0. In case of unavailability of CRA value 0 packet, then a new packet is replaced with the packet having the lowest CRA value. Furthermore, in case of the availability of multiple packets for replacement, the Least Recently Used (LRU) policy is used for replacement. SCBF stores the content name that is cached in CIT. It helps in quick searching of CIT. Before searching in CIT, the content is queried in SCBF. If SCBF returns true, then CIT is searched to retrieve the data. Otherwise, it is concluded that the content is not cached.

11.5 Pending interest table

Pending Interest Table (PIT) [33] is a data structure to save the pending interest. PIT is responsible for saving the interest name and the interface of the interest packet. When a node receives a data packet, the PIT is searched to retrieve the interface. If PIT returns $True$, then the data packet is forwarded to the interface, and the PIT entry is deleted. PIT is a large-sized data structure to prevent overflow because the speed of incoming interest packets is very high. It has high-frequency read and write operations. Hence, another requirement of PIT is highly efficient operations (insert, search, and delete) to cope up with high-speed incoming packets. Every application follows a different mechanism to transmit its packets which influences the size and processing speed of PIT. PIT executes the ENM algorithm for searching the content name of the interest packet. ENM algorithm determines an exact match of the interest packet with the PIT entry. However, in the case of a data packet PIT executes the All Name Prefix Matching (ANPM) algorithm for matching the content name. ANPM retrieves any pending interest having a similar content name to the prefix of the data content. Each PIT entry is attached with a timer to delete entries after the timeout. It helps in preventing saturation of the data structure. It also prevents simple flooding attacks [34]. PIT has important responsibilities, and features in NDN communication [33]. Some of the important features are:

- **Content concentration.** It only stores content. PIT routes the packets without including source or destination address.
- **Security.** Lack of source or destination address increases the difficulty of an attack.
- **Deduplication.** PIT ignores duplicate data.
- **No looping.** PIT saves unique data which prevents looping.
- **Multipath routing.** No looping helps in sending interest packets through multiple interfaces.
- **Multicasting data packets.** PIT forwards the data packet to multiple consumers who have requested the same content.
- **Detecting data packet loss.** PIT waits for the data time out. If the data packet is not received within the time out. It is identified as data loss.

The rest of the section includes a review of various techniques that implement Bloom Filter to enhance PIT. Table 11.4 illustrates the features and limitations of the techniques.

MaPIT [35,36] is a technique that implements Mapping Bloom Filter (MBF) to enhance PIT. MBF consists of a packet store and an index table. MaPIT consists of MBF and CBF. MBF and CBF are constructed in on-chip and off-chip memory, respectively. The index table consists of a Bloom Filter and a Mapping Array (MA). MA provides the offset address of the packet store. Initially, all bits of Bloom Filter and MA are set to 0. The whole Bloom Filter is partitioned into different parts, and each part is mapped to one bit of MA. The content is hashed by hash functions. The hashed value provides the bit location in Bloom Filter. Those bit locations are set to 1. Also, for the part which is set to 1, the corresponding bit in MA is set to 1. A router first searches CS for an interest packet. If CS returns false, then the packet is searched in the index table. Again, if the index table returns $False$, then the interest packet is saved

TABLE 11.4 Features and Limitations of Bloom Filter based techniques for Pending Interest Table.

Technique	Features	Limitations
MaPIT [35]	• Implements Mapping Bloom Filter to enhance PIT • MaPIT consists of MBF and CBF • MBF and CBF are constructed on on-chip and off-chip memory, respectively • Reduces on-chip memory consumption • Easy to deploy on faster memory chip • Reduces the false positive probability	• Average performance • Maintains many data structures • MBF implements standard Bloom Filter
Hardware Accelerator [37]	• A hardware accelerator • Adopted software and hardware co-design approach • Incorporates an on-chip Bloom Filter and an off-chip linear chained hash table • Name ID table saves all distinct content name IDs • Efficient PIT lookup • Bloom Filter reduces lookup time of hash table • Reduces FIB workload • PIT lookup and management is independent of software • Improves throughput	• Requirement of per-packet update in the hash table • Saturated PIT takes more packet processing time • Implements standard Bloom Filter
FTDF-PIT [38,39]	• A technique to enhance the PIT performance by implementing FTDF [38,39] • Each slot of FTDF is a bit vector • FTDF saves the information of the interface number • Quick and efficient deletion • Low data structure construction time • Time complexity of insertion, query, and retrieving interface number order of $O(1)$ • Space and cost-efficient PIT • High security against DDoS attacks • PIT memory independent of length of content name	• FTDF is influenced by hard collision • FTDF requires more memory because each slot is a vector

in FIB. The footprint is stored in MBF and CBF. MA provides the offset address to store the packet information in the packet store. If interest exists or a false positive response is received from the Bloom Filter, then the packet store is updated and sent to FIB. PIT first searches the index table's Bloom Filter after receiving the data packet. If Bloom Filter returns $True$, then PIT searches the packet store. In case the data is absent from the packet store, then data is blocked and the algorithm deletes the footprint from CBF. On the contrary, if the data is present in the packet store, then the packet is transmitted using the faces saved in the packet store. The footprint and record are removed from CBF and the packet store, respectively. Finally, Bloom Filter and CBF are synchronized.

Yu and Pao [37] present a Hardware Accelerator. It adopted a software and hardware co-design approach. It incorporates an on-chip Bloom Filter and an off-chip linear chained Hash Table (HT). PIT has a name ID table (nidT) to save all distinct content name IDs (nid). The presence of content in nidT does not require reference to FIB. This decreases the FIB workload. Hardware accelerator helps PIT lookup and management to be independent of software. Bloom Filter is deployed in on-chip DRAM for the faster searching of the hash table, whereas SRAM stores the full hash table. During a lookup operation, the hardware first searches the Bloom Filter. In case Bloom Filter returns $True$, then the hardware traverses the list of keys mapped to the corresponding bucket. If the key is found, then it is PIT-hit, and the content is deleted. If the bucket becomes empty after the deletion, then the Bloom Filter is reset to 0. PIT-miss occurs when the Bloom Filter returns $False$. In the Lookup Interest (LI) command, hardware searches for the content. The above procedure is followed for searching. The address of the data is returned in the case of PIT-hit. However, the content is inserted into the Bloom Filter in the absence of the content.

Fast Two-Dimensional Filter Pending Interest Table (FTDF-PIT) [38,39] is a technique to enhance the PIT performance by implementing Fast Two-Dimensional Filter (FTDF) [38,39]. Each slot of FTDF is a bit vector. The number of bits of the vector is equal to the number of faces. Each bit of the vector corresponds to a face. A received interest packet is searched in CS and, if not found, mapped to FTDF-PIT. During insertion in FTDF-PIT, first, the interest packet is hashed. Then the hashed value is used by the quotient technique of FTDF to obtain the row and column of FTDF-PIT. In this slot, the corresponding face bit is set to 1. FTDF-PIT caches all data packets coming from any interface in CS. After receiving a data packet, the content name is hashed to get FTDF-PIT slot position. The slot location provides the required face. The data packet is forwarded to all the interfaces whose bit positions are set to 1.

Then these bits are reset to 0. FTDF-PIT has low FPP due to hard collision. It uses a collision-resistant hash function to solve the hard collision issue.

11.6 Forwarding information base

Forwarding Information Base (FIB) [7] is a data structure maintained by the router for storing the next-hop and other related information of every reachable node name prefix. The router refers to FIB for forwarding the interest packet to the next node. If the interest packet is absent in PIT, then the packet is forwarded to FIB. FIB executes Longest Name Prefix Matching (LNPM) for name matching. LNMP helps in obtaining the most accurate forwarding information. LNPM searches against FIB entries for the longest prefix matching with the interest packet. If an FIB entry is found, then the packet is forwarded to the next hop, and a new entry is inserted into PIT. The forwarding decision is taken based on the forwarding policy, or the interest packet is broadcast to all outgoing interfaces. FIB follows color schemes to determine the status of each interface. The color scheme considers three colors: green, yellow, and red [40]. The green color indicates the interface can receive data. The yellow color indicates it is unconfirmed whether the interface can receive data or not. The red color indicates the interface cannot receive data. The yellow color is assigned to a newly added interface. After confirmation that the interface can receive data, the color changes to green; otherwise, to red. The interface status is checked periodically, and based on the changes, the assigned color is updated. Some fields for individual name prefixes in FIB are routing preference, status, stale time, round trip time, and rate limit. The routing protocol assigns the value of routing preference based on routing policy. NDN routing protocol announces the name prefixes added to FIB during routing convergence time. FIB does not delete the name prefix for a stale time period after the deletion of the name prefix from routing. Status is the color code which identifies the interface's performance. FIB has a high read operation frequency, but less write operation frequency. FIB does not play any role in the case of data packets. The rest of the section presents a review of the Bloom Filter-based FIB techniques. Table 11.5 highlights the features and limitations of the techniques.

TABLE 11.5 Features and Limitations of Bloom Filter-based techniques for Forwarding Information Base.

Technique	Features	Limitations
Lee et al. [41]	• Efficient forwarding engines that perform quick name lookups in FIB • Implements a hashing-based Name Prefix Tree (NPT) and a Bloom Filter • Non-chip Bloom Filter reduces access to off-chip hash table • Enables fast look-up	• Implements standard Bloom Filter • Every tree node has a Bloom Filter
BFAST [42]	• A data structure implemented in FIB for faster indexing • Consists of a hash table, a CBF, and *k* auxiliary CBFs Increases parallelism • First-Rank-Indexing reduces memory consumption • Incremental update • Bloom Filter reduces searching time	• Maintains many data structures • Many Bloom Filters lead to overall higher false positive and false negative probabilities • Counter overflow may occur in CBF • Size of CBF is four times that of a standard Bloom Filter • True response of CBF is more costly than the false response
BBS [44]	• Algorithm for efficient content name lookup in FIB • Follows correct path searching while maintaining a low memory requirement • Avoids backtracking • Storing virtual name prefixes in Bloom Filters reduces memory consumption • Efficient with longer prefix and large database • Bloom Filter size is dynamic • Time complexity of BBS is less compared to binary search of hash table	Many lookup operation is performed per request which increases time complexity
MaFIB [36]	• Technique that uses MBF [35] to enhance FIB • MBF is deployed in on-chip memory whereas CBF is deployed in off-chip memory • Low on-chip memory consumption • Low FPP compared to conventional FIB • Supports delete operation	• MBF has high false positive and false-negative probabilities • CBF for off-chip memory occupies high memory

Lee et al. [41] proposed two efficient forwarding engines that perform quick name lookups in FIB. It implements a hashing-based Name Prefix Tree (NPT) and a Bloom Filter. Content name is inserted into the Bloom Filter. First algorithm processes all tree nodes for which the Bloom Filter returns $True$. Name Prefix Tree with Bloom Filter (NPT-BF) is the second algorithm that queries the Bloom Filter to determine the longest matching length. The processing starts from the tree nodes with the longest possible length. After searching the tree node, if no match is found, then the tree nodes are tracked back. NPT is stored in an off-chip hash table, and Bloom Filter is stored on-chip. Every NPT node is stored in a hash table (HT). A hash index is generated for each interest packet by comparing the content name against the HT entry. The search continues till a matching entry is found; no matching entry or search equals the input length. Bloom Filter is searched first before searching the HT. It reduces unwanted HT accesses. If Bloom Filter returns $True$, then HT is searched to retrieve the output face. When Bloom Filter returns $False$, NPT-BF performs backtracking of tree nodes until a matching entry is found.

Bloom Filter-Aided haSh Table (BFAST) [42,43] is a data structure proposed for faster FIB indexing. BFAST consists of a hash table, a CBF, and k auxiliary CBFs. Each CBF slot is associated with a hash table slot. CBF helps in balancing the hash table load. The content name is hashed by k hash functions, which provide the CBF slot locations. Each hash function is also associated with an auxiliary CBF (aCBF). Content is inserted into CBF using the slot locations. The content is also inserted into the hash table slot associated with the CBF slot having the lowest count among the slot locations. The content is also inserted into an aCBF. The aCBF is determined by the hash function, which has returned the slot location, which has the lowest count. The bit location of aCBF is obtained by performing a modulus operation on the hash value generated by its associated hash function. First, CBF is searched during the query of the content in FIB. If it is absent in CBF, then it is concluded that it is also absent in the hash table, so BFAST returns $False$. In case it is present in CBF, all aCBFs are searched.

Bloom-Filter assisted Binary Search (BBS) [44] is an algorithm for efficient content name lookup in FIB. It follows correct path searching while maintaining a low memory requirement. The algorithm avoids backtracking. A Bloom Filter is deployed at each branching point for the detection of prefixes. Each hash table is associated with a Bloom Filter. Searching for a prefix requires three lookups. First, lookup is performed on the Bloom Filter. In case the content is absent in a Bloom Filter, then BBS searches in its right subtree. If the content is absent in the right subtree's Bloom Filter, then BBS searches in the left subtree. Searching the right and left subtrees is the second and third lookup, respectively. Insertion of a content name in FIB involves saving the content in the hash table and the corresponding Bloom Filters. Bloom Filter has a dynamic size which depends on the total number of current length-component name prefixes.

MaFIB [36] is a technique that uses MBF [35] to enhance FIB. MBF is deployed in on-chip memory, whereas CBF is deployed in off-chip memory. Each MBF has a corresponding CBF. Each MBF stores one length name prefix, which speeds up the query operation. MaFIB maintains a packet store to save packet forwarding information. MA value provides the offset address to access the packet store. MaFIB supports delete operation. It is achieved by using timely synchronization between the CBF and MBF. FPP of MaFIB is equal to the sum of the MBFs, which is lower than the conventional FIB.

11.7 Security

The design of NDN architecture provides some basic security. Every data packet has a cryptographically signed producer key [45] to verify the signature. The signature is a message and name, or payload. A consumer accepts a packet after verifying the signature. It is not compulsory for intermediate routers to verify the signature because the verification increases processing overhead. Content signature provides three basic securities: Data integrity, Origin authentication, and Correctness [45].

1. **Data integrity.** Valid signature ensures intact content.
2. **Origin authentication.** Verification of signature using the producer's public key makes it easier for any node to verify the content signature.
3. **Correctness.** Signature ties the content to payload. It helps to securely determine whether the data packet content is the same as the requested data.

The design of NDN architecture also solves some basic DoS (denial of service), and DDoS (distributed denial of service) attacks [46]. Transmission of the data packet is initiated after receiving a request. Moreover, interests

requesting the same content are merged into a single packet. In addition, the number of PIT entries and timeout records helps to analyze and detect attack behaviors. But NDN has many other issues such as [45,46]:

- **Privacy.**
  - **Name privacy.** The interest packet has a hierarchically structured name which makes the name vulnerable.
  - **Cache privacy.** Periodically the neighbors access each other's cache content.
  - **Content privacy.** Data packet content is not encrypted, which makes them vulnerable.
  - **Signature privacy.** The signature identifies the producer and its organization which violates the producer's privacy.
- **Trust.** The public key provided to all nodes to verify the signature may not be trusted.
- **Interest flooding.** A specific producer is targeted by an attacker and flooded with a large number of interest packets. These packets are stored in the PIT. But the huge number increases the difficulty in identifying the legitimate interest packets by PIT. The content requests are of three types: static, non-existent, and dynamically-generated content. The storage or removal of the content in PIT is determined by the type of content. Static content is kept in cache and the interest packets are not forwarded. The router removes invalid requests in case of non-existent contents. But the content remains in PIT till timeout. Interest packets having dynamically-generated content are forwarded to the producer. It occupies the router's PIT and wastes bandwidth.
- **Content/cache poisoning.** If an attacker sends corrupted content, CS caches that content. Upon request, this corrupted content is forwarded in a data packet.

The rest of the section presents a review of the NDN techniques to solve some security issues. Table 11.6 illustrates the features and limitations of these techniques.

TABLE 11.6 Features and Limitations of Bloom Filter-based Security techniques for NDN.

Technique	Features	Limitations
Chen et al. [47]	• A Bloom Filter-based access control mechanism • Protects access to data contents • Bloom Filter prevents unauthorized consumers to access encrypted data contents • Reduces bandwidth	• Maintains multiple Bloom Filters • A Bloom Filter is constructed for each site • Implements standard Bloom Filter
Guo et al. [48]	• An anti-pollution algorithm against the false-locality pollution • Each cached content of CS is associated with a counter and PathTracker • Efficient against address spoofing • One hash function in Bloom Filter reduces time complexity	• Saving path information exposes the Internet structure to attackers • Saving path information increases processing and network overhead • Increases router overhead • Not efficient against address-based attack
CPPM-DAM [49]	• Provides security against cache privacy snooping • Determines legitimate users • Permutation indicator helps in determining the difference between the packet with high probability	• Permutation indicator is an overhead for new content • Increase in attack intensity increases the round trip time • Fewer hash functions leads to the possibility of guessing the permutation by attackers • Small matrix size increases FPP • Permutations need to be saved for future use
Wu et al. [50]	• An access control algorithm for an effective security solution for Smart Homes • Implements CBF to save authorized users • Improves user identity filtering mechanism • Improves user logout mechanism • Reduces total delay • Improves security of the entire system	• CBF has higher memory consumption

Chen et al. [47] proposed a Bloom Filter-based access control mechanism. Bloom Filter is used to identify the authorized users. A Bloom Filter is constructed for each site which saves the authorized users. The digests of all public keys present in the active user tables are inserted into the Bloom Filter, which is piggybacked with a data packet after completion of the Bloom Filter construction. The data packet is sent to the router. The router uses Bloom Filter to pre-filter interest packets from users absent in the active user tables. Periodically, an updated Bloom Filter is sent to the router. A virtual subscription is given to mobile users. When an interest packet is sent by a new user

recently added to a network, the packet gets rejected. The user resends the interest packet, which triggers an update of Bloom Filter on the producer site.

Guo et al. [48] presents an anti-pollution algorithm against the false-locality pollution. Each cached content of CS is associated with a counter and PathTracker. PathTracker is a data structure that stores the number of unique paths traveled by the interest packet corresponding to each content. The counter saves the hit frequency of content. The counter is flushed after every pollution examination. First, CS is searched for an interest packet. If found, its corresponding counter is incremented by 1, and the PathTracker saves the traveled path information. A cache of content that crosses a threshold value indicates a popular content or an attack. The saving of path information exposed the Internet structure to attackers. Moreover, its storage increases processing and network overhead. Hence, in case a cache miss occurs, then its path information is deleted. PathTracker is implemented using probabilistic counting with stochastic averaging and Bloom Filter. One hash function is used in the Bloom Filter.

Cache privacy protection mechanism (CPPM-DAM) [49] is a Bloom Filter-based security mechanism to prevent cache privacy snooping. Bloom Filter has C k-dimensional matrices, where C refers to the number of classes for content popularity, and k is the number of independent hash functions. Each class maintains a k-dimensional matrix to save content which belongs to that class. CPPM-DAM randomly selects n hash functions between $n!$ permutation of hash functions. The interest packet adds a new string field to the packet content name to include the permutation indicator. Same packets have the same permutation. The permutation indicator and the content name are extracted from the interest packet. The permutation indicator generates n addresses. It is concluded that CS contains the content when the cells of n addresses in the matrix are 1. If anyone's cell is 0, then the content is absent in CS. Then it is inserted into the CS by setting the n addresses in the matrix to 1.

Wu et al. [50] proposed an access control algorithm for an effective security solution for Smart Homes. The algorithm implements CBF to save authorized users. It focuses on the effective utilization of cache. The cache helps in data processing for user requests such as cancellation of user privileges, improving user logout, and reducing delays. In a smart home, the home manager is the home administrator, which is responsible for performing functions to grant user privileges. The algorithm has four stages. CBF is initialized in the first stage. The second stage is the user application permission stage. The user signs the interest packet using a private key and forwards it to the home manager. The home manager saves the user information in the user registry after successful authentication of the user. Then the home manager sends the transform key and registration confirmation in a data packet to the user. The third stage is data access by users. The users are permitted to access data in two ways, caching and not caching. Caching in intermediate routers is caching the data, and not caching is not caching the data. The data keys are converted to a ciphertext and forwarded to authorized users. The fourth stage deals with user logout, which occurs in two situations, active and passive cancellation. When the home manager removes the users from the registry based on time, it is passive cancellation. The user removal information is broadcasted to the intermediate routers. Then the intermediate router updates its CBFs based on the information.

11.8 Discussion

An NDN router consists of many data structures to help in smooth packet forwarding. These data structures need to be capable of handling interest/data packets at a frequency of millions per second. Among these, many are duplicate packets. Hence, the application of Bloom Filter in NDN is greatly explored. From this chapter, it can be concluded that all data structures of NDN are exploring Bloom Filter to enhance performance. However, one main concern is the underestimation of Bloom Filter issues. As illustrated by Table 11.7, except for a few, every technique is implementing a standard Bloom Filter. The standard Bloom Filter has high FPP, and the technique does not implement any additional mechanism to reduce FPP. The standard Bloom Filter does not permit a delete operation to prevent the occurrence of false negatives. But, some techniques require delete operations, for instance, in Guo et al. [48]. The frequency of flow of interest packets is millions per second. Hence, Bloom Filter becomes saturated in a short time interval. But, it has kept the old packets due to restrictions on executing delete operations.

CBF was proposed to permit Bloom Filter to execute delete operations. However, implementing CBF is more costly in terms of memory requirements compared to standard Bloom Filter. Hence, other CBF variants can be implemented, for instance, ternary Bloom Filter [56]. Many NDN techniques construct a single Bloom Filter for each interface/faces, for instance, Chen et al. [47] and Mun et al. [15,16], or requires multiple Bloom Filters (e.g., BADONA [57], NameFilter [58], FNR [29]). These techniques maintain many Bloom Filters and perform frequent construction

TABLE 11.7 Comparison of NDN techniques: FP, false positive; FN, false negative; Method, method used to reduce the number of false positives; Component, NDN component where Bloom Filter is deployed; *, no details how CBF is implemented nor any counting mechanism.

Paper	Year	Bloom Filter	Method	FP	FN	Component
MaPIT [35]	2014	Mapping [35]	MBF	✓	✓	PIT
Dai et al. [26]	2014	Conventional*	None	✓	X	CS
Chen et al. [47]	2014	Conventional	None	✓	X	CS
CCNxTomcat [51]	2014	CBF	None	X	✓	Cache and Web Server
CCFS [27]	2015	Stable [28]	Stable BF	✓	X	CS
Guo et al. [48]	2016	Conventional	None	✓	X	Cache
FNR [29]	2016	Conventional	None	✓	X	CS
Lee et al. [41]	2016	Conventional	Increases Bloom Filter size	X	X	FIB
Hardware Accelerator [37]	2016	Conventional	Increase number of hash functions	X	✓	PIT
Mun et al. [15,16]	2017	Stateful [17]	None	✓	X	Packet
BFAST [42]	2017	CBF	None	X	✓	FIB
CT-BF [52]	2017	Conventional	None	✓	X	CS
BFR [20]	2017	Conventional	None	✓	X	Packet
BBS [44]	2017	Conventional	None	✓	X	Lookup Engine
MaFIB [36]	2017	Mapping [35] and CBF	None	X	✓	FIB
Watano et al. [19]	2018	Conventional	None	✓	X	Router
BRCC [32]	2018	Sum-up Counting [32]	Sum-up table and hash-based Look-up Table	✓	✓	Cache
Bourtsoulatze et al. [18]	2018	Conventional	None	✓	X	Packet
B-MaFIB [53]	2018	CBF	None	✓	✓	FIB
Pull-based BFR [22]	2019	Conventional	None	✓	X	Packet
FTDF-PIT [38,39]	2019	Fast Two Dimensional [38,39]	Fingerprint	X	X	PIT
BFH^{2}M^2 [54]	2019	CBF, Mapping [35] and Attenuated [55]	None	X	✓	FIB
Wu et al. [50]	2019	CBF	None	✓	✓	Packet
Marandi et al. [23]	2019	Conventional	None	✓	✓	Packet

and deletion of Bloom Filters. But maintaining many Bloom Filters increases the burden on the NDN router because it has limited memory. In addition, the NDN router has to maintain three data structures: CS, PIT, and FIB.

CS has a high read and write operation frequency. Deploying Bloom Filter before the buffer improves efficiency. Bloom Filter is searched before searching in CS. The time complexity of Bloom Filter for query operation is $O(k) \approx O(1)$. When Bloom Filter returns $False$, CS is not searched, and the interest packet is directly forwarded to PIT. The size of CS depends on the router and the number of edges in the network. Deploying Bloom Filter is a tiny overhead because Bloom Filter occupies less memory. CS requires an efficient cache replacement policy. CS implements skip lists which helps in maintaining order in the data storage. But skip lists have a high query time complexity. Caching decisions require performing frequent query operations. Overall, implementing a skip list is costly. Bloom Filter is more efficient than the skip list. Thus, Bloom Filter is appropriate for CS because it is capable of storing huge numbers of inputs while requiring tiny memory.

The incoming interest packet speed is very high in PIT. The constant insertion time complexity of Bloom Filter helps to cope with the high-speed insertion of interest packets. The PIT data structure size is large to save the packets without overflow. Therefore, implementing Bloom Filter in PIT is not much of an overhead. Another important point is that PIT deletes entries. Thus, PIT requires a Bloom Filter that permits deletion operation while other Bloom Filter issues are less severe. Some examples are Deletable Bloom Filter [59] and Bloofi [60].

11.9 Conclusion

The inability of IP networking to meet the exponentially increasing users and their demands from the Internet is encouraging researchers to explore new networking architectures. NDN is the future Internet architecture. The design of NDN architecture maintains many data structures for efficient handling of packets and security. This also provides scope to Bloom Filter to enhance the performance of NDN by associating in all data structures. This chapter

discussed the role of the Bloom Filter in improving the performance of NDN. This chapter includes the discussion on the NDN data structures: CS, PIT, and FIB. These data structures have different requirements in terms of memory and packet processing. Using this chapter, it is proved that Bloom Filter is capable of fulfilling all requirements of these data structures and enhancing the performance.

References

[1] V. Jacobson, D.K. Smetters, J.D. Thornton, M.F. Plass, N.H. Briggs, R.L. Braynard, Networking named content, in: Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies, ACM, 2009, pp. 1–12.
[2] C. Fang, H. Yao, Z. Wang, W. Wu, X. Jin, F.R. Yu, A survey of mobile information-centric networking: research issues and challenges, IEEE Commun. Surv. Tutor. 20 (3) (2018) 2353–2371, https://doi.org/10.1109/COMST.2018.2809670.
[3] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J.D. Thornton, D.K. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos, et al., Named data networking (NDN) project, Relatório Técnico NDN-0001, Xerox Palo Alto Res. Center-PARC 157 (2010) 158.
[4] NSF, National science foundation, https://www.nsf.gov/publications/pub_summ.jsp?ods_key=nsf13538. (Accessed 26 June 2019).
[5] IRTF, Internet research task force, https://irtf.org/. (Accessed 26 June 2019).
[6] GreenICN, Architecture and applications of green information centric networking, http://www.greenicn.org/. (Accessed 26 June 2019).
[7] D. Saxena, V. Raychoudhury, N. Suri, C. Becker, J. Cao, Named data networking: a survey, Comput. Sci. Rev. 19 (2016) 15–55.
[8] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang, et al., Named data networking, ACM SIGCOMM Comput. Commun. Rev. 44 (3) (2014) 66–73.
[9] A. Afanasyev, J. Burke, T. Refaei, L. Wang, B. Zhang, L. Zhang, A brief introduction to named data networking, in: MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM), IEEE, 2018, pp. 1–6.
[10] G. Xylomenos, C.N. Ververidis, V.A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K.V. Katsaros, G.C. Polyzos, A survey of information-centric networking research, IEEE Commun. Surv. Tutor. 16 (2) (2013) 1024–1049.
[11] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, B. Ohlman, A survey of information-centric networking, IEEE Commun. Mag. 50 (7) (2012) 26–36.
[12] P. Gasti, G. Tsudik, E. Uzun, L. Zhang, DoS and DDoS in named data networking, in: 2013 22nd International Conference on Computer Communication and Networks (ICCCN), IEEE, 2013, pp. 1–7.
[13] N. Aloulou, M. Ayari, M.F. Zhani, L. Saidane, G. Pujolle, Taxonomy and comparative study of NDN forwarding strategies, in: 2017 Sixth International Conference on Communications and Networking (ComNet), 2017, pp. 1–8.
[14] A. Tariq, R.A. Rehman, B. Kim, Forwarding strategies in NDN based wireless networks: a survey, IEEE Commun. Surv. Tutor. (2019) 1, https://doi.org/10.1109/COMST.2019.2935795.
[15] J.H. Mun, H. Lim, Cache sharing using Bloom filters in named data networking, J. Netw. Comput. Appl. 90 (2017) 74–82.
[16] J.H. Mun, H. Lim, Cache sharing using a Bloom filter in named data networking, in: 2016 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), 2016, pp. 127–128.
[17] F. Bonomi, M. Mitzenmacher, R. Panigrah, S. Singh, G. Varghese, Beyond Bloom filters: from approximate membership checks to approximate state machines, ACM SIGCOMM Comput. Commun. Rev. 36 (4) (2006) 315–326.
[18] E. Bourtsoulatze, N. Thomos, J. Saltarin, T. Braun, Content-aware delivery of scalable video in network coding enabled named data networks, IEEE Trans. Multimed. 20 (6) (2018) 1561–1575, https://doi.org/10.1109/TMM.2017.2767778.
[19] H. Watano, T. Shigeyasu, Interest re-route control according to degree of similarity on cached contents using Bloom filter on NDN, in: L. Barolli, F. Xhafa, J. Conesa (Eds.), Advances on Broad-Band Wireless Computing, Communication and Applications, Springer International Publishing, Cham, 2018, pp. 230–240.
[20] A. Marandi, T. Braun, K. Salamatian, N. Thomos, BFR: a Bloom filter-based routing approach for information-centric networks, in: 2017 IFIP Networking Conference (IFIP Networking) and Workshops, 2017, pp. 1–9.
[21] A. Marandi, T. Braun, K. Salamatian, N. Thomos, A comparative analysis of Bloom filter-based routing protocols for information-centric networks, in: 2018 IEEE Symposium on Computers and Communications (ISCC), 2018, pp. 00255–00261.
[22] A. Marandi, T. Braun, K. Salamatian, N. Thomos, Pull-based Bloom filter-based routing for information-centric networks, in: 2019 16th IEEE Annual Consumer Communications Networking Conference (CCNC), 2019, pp. 1–6.
[23] A. Marandi, T. Braun, K. Salamatian, N. Thomos, Network coding-based content retrieval based on Bloom filter-based content discovery for icn, in: ICC 2020–2020 IEEE International Conference on Communications (ICC), 2020, pp. 1–7.
[24] Z. Li, Y. Xu, B. Zhang, L. Yan, K. Liu, Packet forwarding in named data networking requirements and survey of solutions, IEEE Commun. Surv. Tutor. 21 (2) (2019) 1950–1987, https://doi.org/10.1109/COMST.2018.2880444.
[25] S.K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, S. Shenker, Less pain, most of the gain: incrementally deployable ICN, in: Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM, SIGCOMM'13, ACM, New York, NY, USA, 2013, pp. 147–158.
[26] H. Dai, Y. Wang, H. Wu, J. Lu, B. Liu, Towards line-speed and accurate on-line popularity monitoring on NDN routers, in: 2014 IEEE 22nd International Symposium of Quality of Service (IWQoS), 2014, pp. 178–187.
[27] N. Aloulou, M. Ayari, M.F. Zhani, L. Saidane, A popularity-driven controller-based routing and cooperative caching for named data networks, in: 2015 6th International Conference on the Network of the Future (NOF), 2015, pp. 1–5.
[28] F. Deng, D. Rafiei, Approximately detecting duplicates for streaming data using stable Bloom filters, in: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, SIGMOD'06, ACM, New York, NY, USA, 2006, pp. 25–36.
[29] J. Cao, D. Pei, X. Zhang, B. Zhang, Y. Zhao, Fetching popular data from the nearest replica in NDN, in: 2016 25th International Conference on Computer Communication and Networks (ICCCN), 2016, pp. 1–9.

[30] J. Choi, J. Han, E. Cho, T. Kwon, Y. Choi, A survey on content-oriented networking for efficient content delivery, IEEE Commun. Mag. 49 (3) (2011) 121–127, https://doi.org/10.1109/MCOM.2011.5723809.

[31] L.A. Adamic, B.A. Huberman, Zipf's law and the Internet, Glottometrics 3 (1) (2002) 143–150.

[32] R. Hou, L. Zhang, T. Wu, T. Mao, J. Luo, Bloom-filter-based request node collaboration caching for named data networking, Clust. Comput. 22 (2019) 6681–6692, https://doi.org/10.1007/s10586-018-2403-9.

[33] H. Dai, B. Liu, Y. Chen, Y. Wang, On pending interest table in named data networking, in: Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS'12, ACM, New York, NY, USA, 2012, pp. 211–222.

[34] Mengjun Xie, I. Widjaja, Haining Wang, Enhancing cache robustness for content-centric networking, in: 2012 Proceedings IEEE INFOCOM, 2012, pp. 2426–2434.

[35] Z. Li, K. Liu, Y. Zhao, Y. Ma, Mapit: an enhanced pending interest table for NDN with mapping Bloom filter, IEEE Commun. Lett. 18 (11) (2014) 1915–1918, https://doi.org/10.1109/LCOMM.2014.2359191.

[36] Z. Li, K. Liu, D. Liu, H. Shi, Y. Chen, Hybrid wireless networks with FIB-based named data networking, EURASIP J. Wirel. Commun. Netw. 2017 (1) (2017) 54, https://doi.org/10.1186/s13638-017-0836-0.

[37] W. Yu, D. Pao, Hardware accelerator to speed up packet processing in NDN router, Comput. Commun. 91–92 (2016) 109–119, https://doi.org/10.1016/j.comcom.2016.06.004.

[38] R. Shubbar, M. Ahmadi, A filter-based design of pending interest table in named data networking, J. Netw. Syst. Manag. 27 (2019) 998–1019, https://doi.org/10.1007/s10922-019-09495-y.

[39] R. Shubbar, M. Ahmadi, Efficient name matching based on a fast two-dimensional filter in named data networking, Int. J. Parallel Emerg. Distrib. Syst. 34 (2) (2019) 203–221, https://doi.org/10.1080/17445760.2017.1363202.

[40] M.M.S. Soniya, K. Kumar, A survey on named data networking, in: 2015 2nd International Conference on Electronics and Communication Systems (ICECS), 2015, pp. 1515–1519.

[41] J. Lee, M. Shim, H. Lim, Name prefix matching using Bloom filter pre-searching for content centric network, J. Netw. Comput. Appl. 65 (2016) 36–47, https://doi.org/10.1016/j.jnca.2016.02.008.

[42] H. Dai, J. Lu, Y. Wang, T. Pan, B. Liu, Bfast: high-speed and memory-efficient approach for NDN forwarding engine, IEEE/ACM Trans. Netw. 25 (2) (2017) 1235–1248, https://doi.org/10.1109/TNET.2016.2623379.

[43] H. Dai, J. Lu, Y. Wang, B. Liu, BFAST: unified and scalable index for NDN forwarding architecture, in: 2015 IEEE Conference on Computer Communications (INFOCOM), 2015, pp. 2290–2298.

[44] D. He, D. Zhang, K. Xu, K. Huang, Y. Li, A fast and memory-efficient approach to NDN name lookup, China Commun. 14 (10) (2017) 61–69, https://doi.org/10.1109/CC.2017.8107632.

[45] T. Chatterjee, S. Ruj, S.D. Bit, Security issues in named data networks, Computer 51 (1) (2018) 66–75, https://doi.org/10.1109/MC.2018.1151010.

[46] W. Ding, Z. Yan, R.H. Deng, A survey on future Internet security architectures, IEEE Access 4 (2016) 4374–4393, https://doi.org/10.1109/ACCESS.2016.2596705.

[47] T. Chen, K. Lei, K. Xu, An encryption and probability based access control model for named data networking, in: 2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC), 2014, pp. 1–8.

[48] H. Guo, X. Wang, K. Chang, Y. Tian, Exploiting path diversity for thwarting pollution attacks in named data networking, IEEE Trans. Inf. Forensics Secur. 11 (9) (2016) 2077–2090, https://doi.org/10.1109/TIFS.2016.2574307.

[49] Y. Zhu, H. Kang, R. Huang, A cache privacy protection mechanism based on dynamic address mapping in named data networking, KSII Trans. Int. Inf. Syst. 12 (12) (2018).

[50] R. Wu, B. Cui, R. Li, Research on access control of smart home in NDN (short paper), in: H. Gao, X. Wang, Y. Yin, M. Iqbal (Eds.), Collaborative Computing: Networking, Applications and Worksharing, Springer International Publishing, Cham, 2019, pp. 560–570.

[51] X. Qiao, G. Nan, W. Tan, L. Guo, J. Chen, W. Quan, Y. Tu, CCNxTomcat: an extended web server for content-centric networking, Comput. Netw. 75 (2014) 276–296.

[52] R. Zhang, J. Liu, T. Huang, T. Pan, L. Wu, Adaptive compression tree based Bloom filter: request filter for NDN content store, IEEE Access 5 (2017) 23647–23656, https://doi.org/10.1109/ACCESS.2017.2764106.

[53] Z. Li, Y. Xu, K. Liu, X. Wang, D. Liu, 5G with B-MaFIB based named data networking, IEEE Access 6 (2018) 30501–30507, https://doi.org/10.1109/ACCESS.2018.2844294.

[54] I.-H. Bae, Design and evaluation of a Bloom filter based hierarchical hybrid mobility management scheme for Internet of things, in: K.J. Kim, H. Kim (Eds.), Mobile and Wireless Technology 2018, Springer Singapore, Singapore, 2019, pp. 3–15.

[55] F. Liu, G. Heijenk, Context discovery using attenuated Bloom filters in ad-hoc networks, in: T. Braun, G. Carle, S. Fahmy, Y. Koucheryavy (Eds.), Proceedings 4th International Conference on Wired/Wireless Internet Communications, WWIC 2006, in: Lecture Notes in Computer Science, Springer, 2006, pp. 13–25.

[56] H. Byun, J. Lee, H. Lim, Ternary Bloom filter replacing counting Bloom filter, in: 2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), 2016, pp. 1–4.

[57] A. Abidi, S. Mettali Gammar, F. Kamoun, W. Dabbous, T. Turletti, Towards a new internetworking architecture: a new deployment approach for information centric networks, in: M. Chatterjee, J.-n. Cao, K. Kothapalli, S. Rajsbaum (Eds.), Distributed Computing and Networking, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 519–524.

[58] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, Q. Dong, Namefilter: achieving fast name lookup with low memory cost via applying two-stage Bloom filters, in: 2013 Proceedings IEEE INFOCOM, 2013, pp. 95–99.

[59] C.E. Rothenberg, C.A.B. Macapuna, F.L. Verdi, M.F. Magalhaes, The deletable Bloom filter: a new member of the Bloom family, IEEE Commun. Lett. 14 (6) (2010) 557–559, https://doi.org/10.1109/LCOMM.2010.06.100344.

[60] A. Crainiceanu, D. Lemire, Bloofi: multidimensional Bloom filters, Inf. Sci. 54 (Supplement C) (2015) 311–324, https://doi.org/10.1016/j.is.2015.01.002.

This page intentionally left blank

CHAPTER

12

Enhancement of software-defined networking using Bloom Filter

12.1 Introduction

Convention network architecture, i.e., IP-based Internet was designed for text-based networking. However, with the advancement of technology, the Internet is currently transmitting audio and video data along with the text. The Internet-of-Things (IoT) has increased the connectivity of people with others by providing Internet services to small devices such as smartphones and smartwatches. It leads to an exponential increase in data volume, i.e., Big data, and creates a huge network traffic issue. Cloud and mobile computing have paved the path for mobile Internet where the hosts are constantly traveling, thus providing smooth and high-quality services is needed while the network nodes change with respect to the host. It makes the network pattern and conditions change rapidly and dynamically [1]. Furthermore, with the passage of time, everyone accepts the current technology and increases their dependency on the Internet. For example, streaming services promise high-quality services that require constant transmission of high-quality video and audio. But audio and video data consumes more bandwidth and requires more network resources. Another example is remote monitoring of patients using smart medical devices (e.g., heartbeat recording sensor). The patient's information is transmitted and collected in the server to monitor the patient. If any abnormality is detected, the patient and medical professionals are informed for further action. This information of patients is a time-constrained task. Delay in the transmission of data may be fatal to the patient. Thus, a smooth and low latency network connectivity is required. IP-based Internet is unable to accommodate the changing Internet requirements.

In the future, the network bottleneck will further increase because the requirement from the Internet is going to increase, for instance, in the advent of 6G network services. 6G promises to provide a global network on land, in air, and underwater [2]. It will also change the IoT to IoE (Internet-of-Everything). IoE connects every device to the Internet where the devices are not smart but rather intelligent. Another critical point is that the exponential increase in the data volume is leading to the advent of Big Data 2.0 [3]. Thus, there is a requirement for a change in the Internet architecture that can handle the increasing requirement of the Internet and be highly scalable in adding new features.

12.2 SDN architecture

The SDN has three layers, namely, the application, control, and data layer [4] (see Fig. 12.1). SDN layers are also called SDN planes.

1. *Application Layer.* This layer accommodates SDN applications to provide various services such as security, policy implementation, and network management. The Controller uses a northbound open interface to interact with this layer.
2. *Control Layer*. The control layer consists of Controller nodes responsible for controlling the whole SDN network. The control nodes (say, Controller) are logically connected, but the nodes are distributed over the network. The network continuously generates information based on network traffic and sends these to the Controller. It helps the Controller to have a global view of the network. Moreover, these data are analyzed to generate information for developing new security policies and detecting anomalies. These security policies are later implemented in the network to enhance the network's security. Moreover, the control layer provides various hardware abstractions to SDN applications. The Controller uses the East/WestBound open interface to interact with other controller nodes.

Bloom Filter https://doi.org/10.1016/B978-0-12-823520-1.00018-3

Copyright © 2023 Elsevier Inc. All rights reserved.

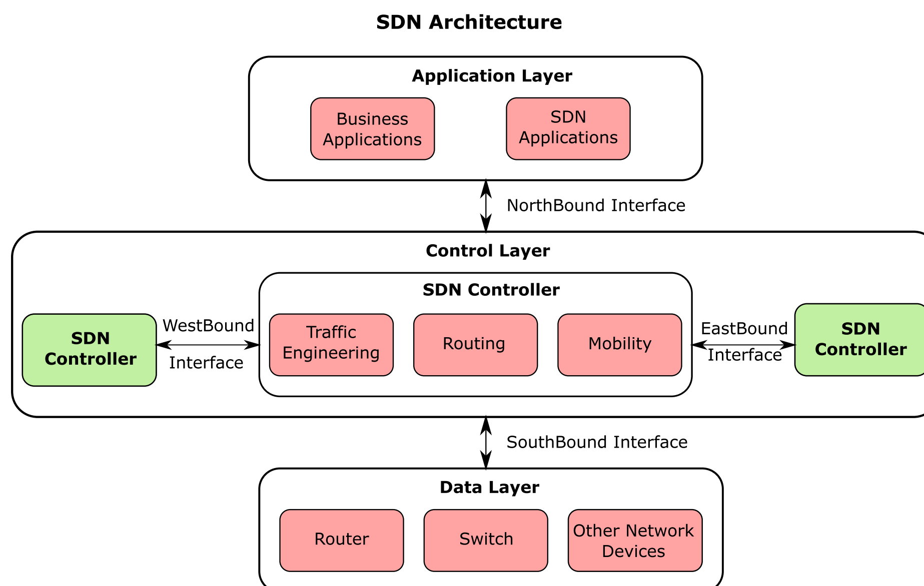


FIGURE 12.1 SDN Architecture.

3. *Data Layer.* This layer is also called the infrastructure layer [4] and consists of forwarding elements. The forwarding elements are physical and virtual switches. They are reachable through an open interface, which permits packet switching and forwarding. The forwarding elements follow the instructions provided by the Controller to forward the packets. The instructions are defined by the southbound interface, which also defines protocol oblivious forwarding (POF) [5]. The Controller uses the southbound open interface to interact with this layer.

12.2.1 Interfaces

The Controller has three open interfaces to interact with the same or other layers. The three interfaces are Northbound, Southbound, and East/WestBound.

1. *Northbound.* The northbound interface is used to interact with the application layer. It provides provisioning services, network topology, and configuration retrieval. Furthermore, it helps in path computation, loop avoidance, and security [6].
2. *Southbound*. The southbound interface interacts with the forwarding planes, i.e., the data layer. Some examples of the protocols for southbound interface are OpenFlow [7], OpFlex [8], and Open vSwitch Database Management Protocol (OVSDB) [9]. Two well-known southbound protocols are Forwarding and Control Element Separation (ForCES) [10], and Openflow [11]. OpenFlow is the first and most widely used protocol to connect Controllers and the forwarding elements. It was standardized by the Open Networking Foundation (ONF) [12].
3. *East/WestBound.* East/WestBound is used for interaction between the Controller nodes. The main requisite of this interface is advanced data distribution techniques [1].

12.3 Control layer

The control plane of SDN consists of Controller nodes. The Controller is a software control program that is the brain of the SDN network. The main responsibility of SDN is constantly coordinating with forwarding elements to manage the flow of traffic. The control layer is also called a Network Operating System (NOS) because it is the control logic and provides an abstract global network view to the application layer [1]. The Controller is of two types based on their numbers, i.e., centralized or distributed Controller. The centralized Controller is a single Controller that monitors and manages the whole network. Some examples are NOX-MT [13] and Beacon [14]. Distributed

Controllers consist of multiple Controllers physically distributed in the network, but they communicate with each other to maintain a global view of the network.

12.3.1 Architecture

The architecture of the Controller affects the performance of SDN. Suppose there is a single Controller and a huge amount of traffic situation occurs, then the Controller is unable to obtain an optimal route because it will exceed its capacity. Furthermore, the Controller will be in a bottleneck which may continue for a whole week [15]. Thus, multiple Controllers are present in the control plane, which are distributed physically but logically connected as one unit. Three types of Controller architecture are proposed, namely, multi-core, logically centralized, and entirely distributed [16].

Multi-Core Controllers. A single controller manages the SDN network, but it has multiple cores to speed up and handle a huge number of tasks. However, scalability and single point of failure are the main issues in such an architecture [17].

Logically Centralized Controllers. The control plane has multiple Controllers distributed physically. But they maintain a consistent view of the whole network by sharing information with each other. Some examples are OpenContrail [18] and ONOS [19]. When a Controller's local state changes, it is synchronized with other Controllers. The presence of multiple Controllers improves the performance of the network. However, the synchronization among the Controllers consumes huge network resources.

Completely Distributed Controllers. The architecture is similar to logically centralized Controllers, where the Controllers are physically distributed in the network. Moreover, they also synchronize with each other to maintain a consistent global view of the network. However, they try to reduce the overhead of the Controller state synchronization.

12.3.2 Network monitoring

Network monitoring has five steps, namely, collection, preprocessing, transmission, analysis, and presentation [20]. Data collection and preprocessing are the responsibility of the data layer. In the preprocessing step, the collected data are aggregated and converted into some specific statistical format. This step itemizes and also helps in identifying the source of the data. This step is important because it eliminates more garbage data and reduces the data processing overhead. The data is transmitted from the Data layer to the Controller to perform analysis in the transmission step. Data analysis is performed to produce statistics and determine specific events. The analysis algorithms are executed in the application or control layer. However, the analyzed data are forwarded to Controllers for adaptation. This step has four common functions: traffic statistics, anomaly detection, traffic engineering, and fault management. Traffic statistics consider packet loss, throughput, latency, and link utilization. The anomaly detection concerns SDN network security. Fault management handles two issues, namely, fault detection and fault recovery. Traffic engineering involves determining the shortest path and optimizing link utilization. The presentation step converts the analyzed data into some presentation format such as graphs, statistics, etc. Topology graphical user interface (GUI) is used to visualize the network topology using network graphs. Management interfaces represent the flow tables, review real-time traffic information, and inform the operators regarding elephant flows or unexpected system errors.

12.4 Data layer

In this section, both OpenFlow and ForCES data plane architecture is discussed.

12.4.1 OpenFlow

The data plane of OpenFlow [11] consists of switches where the switch has a fixed architecture, unlike ForCES (see Fig. 12.2). An OpenFlow switch consists of three tables, channels, and protocol types. The three types of tables are meter, group, and flow table [21]. The flow table stores the flow records, along with the associated actions. A single switch may contain multiple flow tables to pipeline the operation. Sometimes the flow table forwards the

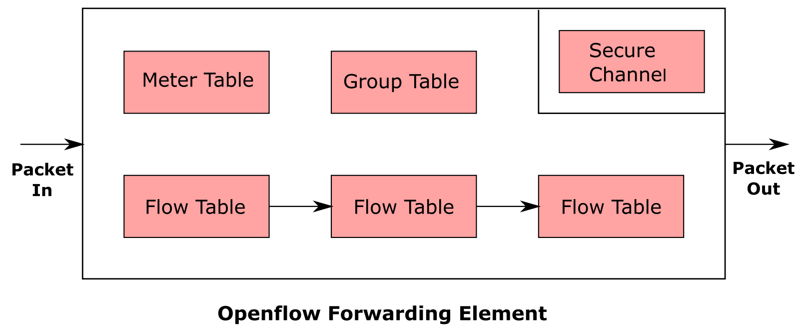


FIGURE 12.2 OpenFlow Forwarding Element.

packet to the group table or meter table. The group table initiates some additional actions in one or more flows. When a packet is forwarded to the meter table, performance-related actions are initiated on the flow. The channel is a secure channel to connect the switch with the Controller. OpenFlow protocol bestows standardized communication between the switch and the Controller. The flow record stores the flow records, which the Controller defines. Each record has three fields: packet header, action, and statistics. The packet header matches the packet with the entries stored in the flow table. An action determines the procedure for packet processing used on similar packets. Statistics stores statistics associated with the flow, for instance, flow counter. OpenFlow switch has three responsibilities: (a) packet forwarding to a particular port or multiple ports, (b) packet encapsulation and forwarding the packet to the Controller, and (c) discarding of packet. OpenFlow switches are of two types, namely, pure and hybrid. Pure OpenFlow switch is responsible for following the flow table for packet forwarding and communicating with the Controller. The hybrid switch has another additional responsibility of supporting autonomous operation similar to a conventional Ethernet switch in case of the absence of a Controller.

When an OpenFlow switch receives a packet, the packet header matches the flow table entry. Packets are processed based on the priority defined by the Controller. When a match is found, the flow counter is incremented, and the action associated with the matched flow entry is executed for packet processing. In case the first packet of a new flow is received by the switch, then the packet is encapsulated in the OpenFlow Packet-In message and transmitted to the Controller using the secure connection. The Controller produces a new forwarding rule which will be followed for the new flow and sent to the switch. On the contrary, the Controller can also instruct the switch to drop the packet. The Controller uses the OpenFlow Packet-Out message to send the packet and the actions. The switch stores the forwarding rule in the flow table. OpenFlow data plane has scalability and performance issues because most of the work is performed by the Controller and the switch forwards the packets.

12.4.2 ForCES

RFC 5812 [10] first proposed data plane architecture that defines ForCES protocol. ForCES forwarding element is called a logical functional block (LFB), a packet processing functionality. Some examples of LFB are classifiers, shapers, and meters. One LFB is responsible for a single task. LFB helps in using fewer LFBs to complete the complex packet forwarding task. During packet processing, some information in the packet is modified, or new information is produced, which is stored as metadata. Hence, metadata represents the state of the packet. In some cases, LFB deletes the metadata and forwards the packet without any metadata. A single forwarding element contains multiple LFBs. Fig. 12.3 illustrates a forwarding element having multiple LFBs. Each LFB has many input and output ports. Input port takes packet and metadata, but any value (i.e., packet or metadata) can be empty. One output port of an LFB is connected to a particular input port of another LFB to avoid ambiguity. Some LFB also has a single output port.

12.4.3 Review

This section presents a review on SDN techniques that have used Bloom Filter to solve some problems or enhance the network's performance. Table 12.1 highlights various features and limitations of the SDN techniques.

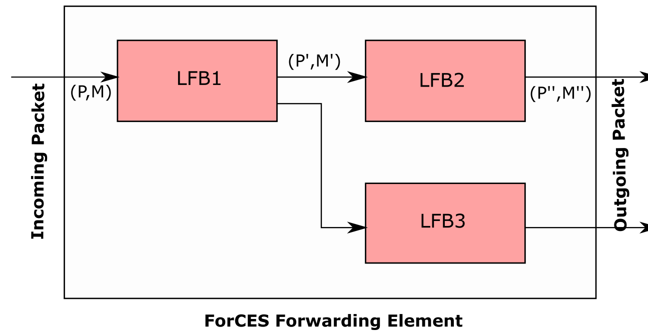


FIGURE 12.3 ForCES Forwarding Element. LFB1, LFB2, and LFB3 are the logical functional blocks present in the forwarding element (Note that the arrangement of the LFBs are not fixed as shown in the figure); P, P′, and P″ are the packets taken as input by LFBs while M, M′, and M″ are the metadata taken as input by LFBs.

TABLE 12.1 Features and Limitations of SDN security techniques.

Technique	Features	Limitations
MPBF-TVHT [22]	• Performs two verification to reduce FPP • All operations on Bloom Filter are performed in parallel • Hash table is used to reduce FPP • CAM records the insert collision • Checking in hash table and CAM is performed in parallel	• FPP increases with increase in valid bits in mask vector • FPP increases with less valid bits in mask vector • Maintains multiple Bloom Filters • Implements standard Bloom Filter • Increase in number of items and FPP requires more CAM memory • Does tradeoff between FPP and memory
Challa et al. [23]	• Reduces the frequency of switch and Controller communication • Arrangement of MBF in Column-Major order optimizes the logical shift operations • Stores information of past events	• Maintains multiple Bloom Filters • Uses standard Bloom Filter • Uses many data structures
2MVeri [24]	• Verifies the control-data plane consistency • Determines faulty switch	• Verification fails if Bloom Filter tag passes same switches but in different order compared to correct tag • Inverse of characteristic matrix should exist • False positive response results in missing the recognition of a faculty switch • Verification is a overhead
QBF [25]	• Performs selective caching • Selective caching reduces memory consumption • Deletes Bloom filters to remove older elements without introducing false negatives • Suitable for large sized flow • Maintains record of old flows	• Not suitable for mice flow • Maintains multiple Bloom Filters • Implements standard Bloom Filter
Huang et al. [26]	• Fast packet classification algorithm • Bloom Filter reduces the number of unnecessary paths • Multiple R*-Trees reduce the number of necessary paths	• Multiple R*-Trees are constructed for low priority rules • Bad performance in firewall rule tables • Implementing Bloom Filter reduces performance • Maintains multiple Bloom Filters • Implements standard Bloom Filter
Parzyjegla et al. [27]	• Storage of subscription information in the Bloom Filter enhances security • Switches use Bloom Filter to determine the forwarding decision	• Large number of subscriptions for a notification requires more Bloom Filters • Large number of subscriptions sends more notification packets • Implements standard Bloom Filter

continued on next page

Masked Parallel Bloom Filter with Twice Verification Hash Table (MPBF-TVHT) [22] is a Bloom Filter-based technique for multi-protocol query for SDN switch. The Masked Parallel Bloom Filter (MPBF) comprises multiple Bloom Filters. The item has multiple fields, and each field is stored in a different Bloom Filter. Each field is hashed by k hash functions when an item is received. The k hash values are k locations of the cells in Bloom Filter. These locations are

TABLE 12.1 (*continued*)

Technique	Features	Limitations
TSA-BF [28]	• Segmented aging Bloom Filter automatically ages and deletes content • Bloom Filter reduces communication cost • Avoids cold cache effects • Encoding of ternary prefix-rules into binary codewords eliminates rule-set expansion • Codewords prevent false registration in Bloom Filter • Performs few memory accesses • Reduces power waste • Partial delete operation reduces false negative probability • Partial delete operation reduces the number of segment accesses	• Delete operation still generates false negatives • Maintains many data structures • Maintains multiple Bloom Filters
Abbasi et al. [29]	• Reduces memory accesses • Reduces memory occupancy • Leaf-pushed KD tree has less search time compared to KD tree	• Maintains multiple Bloom Filters • The number of Bloom Filters depends on the tree size • Implements standard Bloom Filter
LBF [30]	• Combining RNN and Bloom filter enhances efficiency of packet classification • Bloom Filter helps in re-verification of negative outcomes	• The number of hash table accesses depends on the FPP of LBF • FPP of LBF depends on RNN and Bloom Filter size • Lower memory footprint gives high FPP • FPP of LBF adversely affect the match-searching speed
DCM [31]	• Flow monitoring • Load balancing • Per-flow monitoring • Memory efficient • Scalable • Two stage Bloom Filters monitor the rules • Admission Bloom Filter reduces the number of packets checked by the action Bloom Filter	• Reconstruction of Bloom Filter is performed periodically • Maintains many action Bloom Filters • If Bloom Filter takes more memory then accuracy reduces because CountSketch has less memory • Requires delete operation on Bloom Filter
NDN-SF [32]	• Improves name searching • Improves content forwarding • Reduce processing time • Reduces the number of packets in the network	• Response time increases greatly in case the content is forwarded to a different domain • Implements standard Bloom Filter
BloomFlow [33]	• False positive free • Being false positive free prevents bandwidth utilization overhead • Performance is good in networks with sparse multicast groups	• Maintaining a false-positive-free Bloom Filter requires additional query operation overhead • Maintains multiple Bloom Filters • Bloom Filter length is limited to 320 bits to accommodate it in IP header • Change in the topology requires change in the multicast tree • Cannot support very large network

set to 1. There is a mask vector of the bit length equal to the number of Bloom Filters. Each bit is given to each Bloom Filter. This vector also affects the result of the Bloom Filter. If a bit of the mask vector is 1, then the state of Bloom Filter is valid; otherwise invalid. This mask vector defines a rule. Change in mask vector indicates implementation of a new rule; hence, the previous information is discarded. This technique follows two verification procedures. It performs the first verification is checking with Bloom Filters. All the Bloom Filter results are multiplied with the corresponding bit value of the mask vector. Then all these results are summarized to obtain a single value called $EXIST$. The second verification involves a hash table. The hash table contains some information to determine false positive responses. The hash table contains $EXIST$, KEY, and $RESULT$. The KEY is the item. $RESULT$ is the action taken. During insertion, the new item is checked in Bloom Filters. Then the new item is checked in the hash table. If the new item $EXIST$ is 0 and KEY does not match, then the new item is inserted into the Bloom Filters. If $EXIST$ is 1 and KEY matches, then the new item exists, and it is ignored. In case $EXIST$ is 0 and KEY matches, then $EXIST$ is a false positive response. In case $EXIST$ is 1, and KEY does not match, then Content Addressable

Memory (CAM) is used to store the insert collision record. The result is forwarded to the hash table and CAM in parallel during the second verification.

Challa et al. [23] proposed a flow entry restriction strategy using Multiple Bloom Filters (MBF). MBF helps in efficient management of the flow table. The OpenFlow switch uses Ternary CAM (TCAM), static RAM (SRAM), and specialized Application Specific Integrated Circuit (ASIC). TCAM contains the flow table and is responsible for packet processing. SRAM contains the MBF and stores information about flows. The hash functions are stored in ASIC. The recentness and frequency determine the priority of a flow called importance value. It is calculated by performing the left shift operation on an array storing the importance value. When the OpenFlow switch generates a packet, the flow table uses the header and determines its corresponding rule. In parallel, the header is forwarded to the ASIC. The hash functions present in ASIC hashes the header. The hashed values give the locations of the Bloom Filter cells. All Bloom Filters are checked. If the flow is present, then the importance value of the flow is incremented. When the flow table is full, low priority flows are removed by retrieving the importance value. MBF consists of multiple Bloom Filters (say, B Bloom Filters). Each Bloom Filter stores information for a unit time period (say, T). Hence, MBF stores information during a BT-long time interval. Let the time be 0. Now the first Bloom Filter stores the information of the flow. After time T, the second Bloom Filter stores the information. Similarly, operations on other Bloom Filters are followed. A bitmask is used to determine the current Bloom Filter. The importance value is decremented after time T, which is called importance decay.

2MVeri [24] is a framework for verifying the control-data plane consistency. This technique embeds a Bloom Filter and a two-dimensional array (say, 2-D array) in the packet header. The 2-D array stores two positive random numbers of 32 bits. The Bloom Filter is 16-bit long. These two together are called the tag. When a switch is initialized, the Controller assigns a 2×2 characteristic matrix. The characteristic matrix contains positive numbers. When the packet crosses a switch, the tag is updated. The 2-D array is multiplied by the switch matrix, and the switch ID is inserted into the Bloom Filter. When the exit switch receives the packet, it retrieves the tag and forwards it to the Controller, which obtains a correct tag using matrices and switch IDs. Then the Controller compares the correct tag and the Bloom Filter tag. The correct tag is determined before receiving the Bloom Filter tag. A switch's actual existence is determined by comparing the existence of the switch in both the Bloom Filter and in the product with the inverse of the characteristic matrix. If it does not exist in any of them, then the switch is a faculty switch.

Queue Bloom Filter (QBF) [25] consists of multiple Bloom Filters to reduce the memory consumption of the cache. QBF is a time series queue that removes elements while avoiding false positives. After a fixed time period (say, T), a new Bloom Filter is queued into QBF. Current element's flow is inserted into the most recent Bloom Filter. QBF maintains a fixed number of Bloom Filters. After exceeding the maximum number, the oldest Bloom Filter is dequeued. It helps to delete elements without introducing false negatives. EGRESS is a router. When EGRESS receives a packet, it constructs data pieces, i.e., prefix pattern. The data pieces are queried to all Bloom Filters. If all Bloom Filters return $False$, the data piece is inserted into QBF. Otherwise, the number of hits of the flow is counted. If the number of counts is more than a threshold value, it is assumed that the flow is forwarding duplicate content. Then the content is cached.

Huang et al. [26] proposed a fast packet classification algorithm using R*-Tree-based Bitmap Intersection. This algorithm is further improved by using Bloom Filter and multiple R*-Trees. The paths are classified into necessary and unnecessary paths. The necessary paths are those paths that lead to results. The unnecessary paths are those paths that do not lead to results. The Bloom Filter reduces the number of unnecessary paths taken. The multiple R*-Tree helps in reducing the number of necessary paths by constructing different R*-Trees. The nodes of the R*-Tree are rules for packet classification. The rules are further classified to form the children node. Thus, the R*-Tree of rules is constructed. Bitmap Intersection is used to search the R*-Tree. The Bloom Filters are deployed in higher levels of the R*-Tree, such as level 1 or level 2. The children's rules are inserted into the Bloom Filters. When searching the R*-Tree, the Bloom Filter helps to determine the path that can be avoided as it does not contain the required rules. A node contains many Bloom Filters. Each Bloom Filter saves a field of the rule. The information about various fields of rules is retrieved from the packet header. Then these fields are checked in the Bloom Filters. If at least one Bloom Filter returns $False$, then that path is ignored. This algorithm using Bloom Filter has bad performance because the unnecessary paths have less influence, but more computation is performed for avoiding them.

Parzyjegla et al. [27] proposed a notification routing protocol. Each notification stores a Bloom Filter. The notification contains the matching subscription information. OpenFlow switch refers to the Bloom Filter to determine the forwarding decisions. Bloom Filter stores the subscription matched by a notification. The Controller is responsible for determining the rules and installing them. It is achieved by analyzing the network topology. A new client informs the Controller about its aliveness and role. The publisher forwards an advertisement about the notifications that are

produced by it. A subscriber constructs a subscription mentioning their interested notifications. Both advertisements and subscriptions are transferred to the Controller, and the Controller analyzes the advertisements, subscriptions, and network topology to determine the routing rules. The Controller informs every publisher about the subscriptions that match a notification. Each OpenFlow switch port maintains a flow table. The flow table stores entries of all active subscriptions of the client whose notification can reach the port. When a packet is received, it is searched in the flow table. If some entries match, the notification packet is transferred from the corresponding port. Bloom Filter stores all subscriptions matched by the notification. Then the publisher embeds the Bloom Filter in the notification packet. A saturated Bloom Filter gives more false positives. Hence, the subscriptions are partitioned when the number of Bloom Filter inserts crosses a threshold value. The subscriptions are classified into two or more disjoint sets of smaller sizes. These sets are stored in different Bloom Filters. The Bloom Filter helps the switches in transferring the notification to the subscriber.

Ternary Segmented Aging Bloom Filter (TSA-BF) [28] is a dynamically updatable Bloom Filter. Segmented Aging BF (SA-BF) is proposed for dynamic updates while occupying less memory. It consists of two buffers, active and warm-up. Both buffers are constructed in segment form. When a packet is received, SA-BF executes a query operation. If the packet is present, the packet is forwarded to the TCAM lookup table. Otherwise, the packet is inserted into SA-BF. The count of the number of insertions is recorded by a counter. The update operation is executed when the counter reaches a threshold value. An insertion operation performs two insertion operations, one in the active buffer and the other in the warm-up buffer. During an insertion operation, the packet is hashed by k hash functions. The hash values give k cell locations in each of k segments in SA-BF. The k cell locations are set to 1. During a query operation, the packet is queried in the active buffer. The packet is hashed by k hash functions. The hash values give k locations in k segments. If the locations have bit value 1, then the packet is present; otherwise absent. The delete operation in SA-BF has two procedures, partial delete and cyclic update. A pointer is used to point a segment in the active buffer. When the delete operation is executed, the pointed segment is first replaced by its corresponding segment in the warm-up buffer. Then, the partial delete operation is executed on the replaced warm-up segment. The packet is hashed by one hash function, which gives a location, and it is reset to 0. Then the segment pointer moves to the next segment in the active buffer. In case the segment is the last segment, the pointer points to the first segment. TSA-BF handles ternary data of OpenFlow switches. TSA-BF has Ternary Prefix-tagging Encoder (TPE), rule preprocessor, and packet preprocessor. Rule preprocessor is responsible for insertion in cooperating Bloom Filters. Packet preprocessor is responsible for a query operation. The prefix is referred to as rules. When a rule is inserted, first, the mask length is extracted from it. TPE uses the prefix and prefix length to generate a unique codeword. If the rule is a prefix, then the most significant bit (MSB) of the codeword is set to 1. If the rule is an exact match, then MSB is set to 0. The codeword is inserted into the cooperating Bloom Filters.

Abbasi et al. [29] proposed an enhanced KD-tree using Bloom Filter and leaf-pushing. The rules/prefix are stored in a tree structure. The root is partitioned using the first MSB bit of the source prefix. If a source prefix has MSB bit as 0, it is inserted into the left subtree; otherwise, it is the right subtree. Next level partitioning is performed using the destination address. This procedure is followed till the inclusion of all rules in the KD tree. A leaf pushed KD tree pushes all the prefixes from the internal nodes to the leaves. Hence, all prefixes are stored in the leaves. Each leaf represents a range of prefixes. The rules are stored in the hash table. A Bloom Filter is stored in every node. Bloom Filter inserts the prefixes of the node. When a packet is received, a substring is retrieved from the packet, which is the combination of source and destination address. The length of the substring is equal to the longest prefix stored in the KD-tree. This substring is queried to the root of the tree. If the Bloom Filter responds in true, then the substring is queried to the Bloom Filter of its subtrees. The Bloom Filter that responds $True$, the substring is queried in the subtree Bloom Filter. Whereas if Bloom Filter responds $False$, that subtree traversing is terminated. This is followed till the leaf node, which contains the pointer to the database. The pointer contains the address of the required rule in the hash table. The rules are retrieved from the hash table using the pointer.

Learned Bloom Filter (LBF) [30] is a packet classification algorithm that combines recurrent neural network (RNN) learned model and Bloom Filter. The RNN helps to classify the addresses into positive and negative sets. The positive set includes the rules that give possible outcomes. The negative set includes those that do not have any rules. After receiving a packet, the source and destination addresses are retrieved from the packet header. Both the addresses are searched in the tree in parallel. The matched results of source and destination addresses are cross-combined and given to RNN, which generates positive and negative sets. If RNN concludes a rule as positive, then the rule is inserted into the hash table. Otherwise, the rule is inserted into both the hash table and Bloom Filter. The negative set may contain false results. Hence, the elements of the negative set are queried to Bloom Filter for re-verification. If an element is absent from Bloom Filter, it is removed from the negative set and added to the positive set. Then the

negative set is forwarded to the Controller. The elements of the positive set are hashed. If the elements are present in the hash table, then rules are searched using the remaining three fields of the packet header. A linear search is performed in this search because the search domain is small.

The Distributed and Collaborative Monitoring system (DCM) [31] is a system that performs flow monitoring in the switches, load balancing, and per-flow monitoring. It is memory efficient and scalable. DCM implements two-stage Bloom Filters for monitoring the rules while using less memory. The Controller is responsible for installing, updating, and constructing the two-stage Bloom Filters in the data plane of switches. Controllers construct the admBFs and actBFs based on the current flow information. Then the Bloom Filters are encapsulated in control messages and forwarded to the switches. The Controller also periodically updates the Bloom Filters based on the condition of the network flows. The Controller follows Real-time Addition and Periodical Reconstruction (RAPR). When a new flow is detected, the Controller informs the switches to update their Bloom Filter to support the new flow. After a time period, the Controller reconstructs all Bloom Filters. When a flow terminates, the Controller cannot delete the flow from the Bloom Filter because it increases the false negative probability. Also, ignoring the update saturates the Bloom Filter and increases the FPP. During reconstruction, the Bloom Filter size is based on the current status of the number of flows. Admission Bloom Filter (admBF) and action Bloom filters (actBFs) are the first and second stages of the two-stage Bloom Filter, respectively. admBF stores flow information to determine whether a flow is under monitoring or not. actBFs determine the monitoring action. An actBF is constructed for every action. The wild card rule initiates action to an aggregate of flows. When a packet is received by a switch, first, it is checked whether the packet matches a wild card rule. If it does, then the action is followed, and the packet does not use Bloom Filter. Otherwise, it is checked in admBF. If admBF returns $False$, then the packet is forwarded for further processing. Otherwise, the packet is checked in actBFs. Each packet is checked in all actBFs. admBF helps in reducing the number of packets processed by the actBFs. It reduces resource consumption. DCM also performs load balancing. In case there are fewer flows, then, instead of all, some switches are assigned to the flows. This increases efficiency because every packet processing switch has to construct and maintain actBFs. When the number of flows crosses a threshold value, some more switches are considered for packet processing. This balances the load in switches.

Named Data Networking based on SDN and Bloom Filter (NDN-SF) [32] is a forwarding strategy for efficient content delivery and enhancement of efficient resource consumption. The whole network is partitioned into areas or domains in the NDN-SF network. Each domain has a Controller, NDN nodes, and some switches. The communication between Controller and switch follows OpenFlow protocol. The switches contain Switch Forwarding Information Base (S-FIB), Switch Content Store (S-CS), and Switch Pending Interest Table (S-PIT). S-FIB contains the information about the set of ports against each data name. It is updated and modified by the Controller. S-CS is the cache that stores the most popular content. S-PIT stores the information about the data names and their corresponding requesting interfaces. NDN nodes use three Bloom Filter tables: FIB-Bloom, PIT-Bloom, and CS-Bloom tables for a faster name lookup process. FIB-Bloom, PIT-Bloom, and CS-Bloom tables are constructed using the Bloom Filter with Counter (BFC) to the S-FIB, S-PIT, and S-CS, respectively. BFC is an array of m bits. Initially, all cells are set to 0. When inserting a prefix, it is hashed by k hash functions. The k hash values give k cell locations. These locations are set to 1. When content is forwarded to S-FIB, S-PIT, or S-CS; first, the content is searched in their respective BFC. If the BFC returns $True$, then the prefix is searched in the table. Otherwise, the content is forwarded to the Controller to search in other domains.

BloomFlow [33] is a Bloom Filter-based multicast approach for SDN networks. The Controller assigns a 16-bit integer unique identifier to every interface in the network. Moreover, the Controller uses a tag to identify the multicast packets that have Bloom Filter shim headers. Changes in network topology or multicast workload are detected by installing a flow-in forwarding element (FE) that transmits all the Internet Group Management Protocol (IGMP) packets to the Controller. The change requires a particular multicast source/group pair traffic to re-route. When the Controller determines a change, the Controller calculates a multicast tree that connects the multicast source and destination nodes. The link in the multicast tree is associated with an interface identifier. These identifiers are classified into disjoint sets based on the distance (number of hop(s)) from the tree root. Each set of identifiers is inserted into a Bloom Filter. The Bloom Filter uses k hash functions, and m is the Bloom Filter array length. The disjoint set is called the inclusion set because these identifiers are inserted into the Bloom Filter. Another set is constructed for each inclusion set, called the exclusion set. The exclusion set contains the identifier that should not be added to the Bloom Filter. After insertion of the identifiers into the Bloom Filter. The exclusion set identifiers are queried into the Bloom Filter. This confirms a false-positive-free Bloom Filter. The k and m values are encoded using Elias gamma encoding [34] into a binary format. The encoded values are concatenated to the Bloom Filter to have a shim stage. All shim stages are concatenated together to construct a single shim header. The Controller installs a single flow

table entry in every FE. When an FE encounters a packet with the multicast tag, it checks with the flow table. The *m* and *k* values are first extracted when extracting the Bloom Filter. The rest is Bloom Filter. It is copied to a cache for further processing. Left-shift operations are performed to extract the first Bloom Filter. Similarly, all the Bloom Filters are extracted. When the root FE receives the packet, it adds the shim header to the packet. Then it forwards the packet to the next hop. When a node has a distance less than the depth of the multicast tree, the packets are forwarded according to the shim header. Also, the packets are forwarded to direct connected nodes. The identifiers in the inclusion set are inserted, and identifiers of the exclusion set are queried to achieve a false-positive-free Bloom Filter. However, in case any identifier of the exclusion set is inserted into the Bloom Filter, then how it is removed from the Bloom Filter is not mentioned. Standard Bloom Filter does not permit the delete operation. Then to remove the exclusion set identifier(s) Bloom Filter needs to be reconstructed.

12.5 Issues and challenges

- **Single point of failure.** When the network maintains a single Controller, then the failure of the Controller will impair the whole network.
- **Synchronization Overhead.** When the SDN network has multiple Controllers, they have to maintain a common global view of the network. Thus, they communicate with each other using the East/WestBound Interface to update about any changed local state of the Controller. However, this consumes network resources and leads to heavy overhead.
- **Scalability.** Increase in traffic will constantly keep the Controllers in a bottleneck. Thus, it forces to include more Controllers in the network.
- **Privacy.** Different SDN domains may follow different privacy policies. Some SDN domains are also dedicated to a specific number of customers, which defines their own policies [21].
- **Interoperability.** Distributed Controllers are maintained in the SDN network to overcome many issues. However, among the Controllers' interoperability issues exist because the Controllers are responsible for different network domains while operating in different technologies. The Controllers communicate with each other using the East/WestBound interface, but it has a deficiency of an open standard.
- **Reliability.** The separation of control and data layer has raised doubt on the reliability of the SDN network. The Control layer centrally controls the whole network; hence, in case of failure due to any reason can collapse the whole networking system [1].
- **Overhead.** Forwarding elements constantly collect network data and transmit it to the control layer. However, these activities are overhead on the networking system. But, if these activities are not performed, then the Controller will not be able to maintain the global view of the network and cannot rectify any issues arising due to lack of information regarding the network situation.

12.6 Security

SDN has two vulnerable points from where the attack is possible, one is Controller, and the other is Switches or forwarding element [16]. The Controller can be tricked by making the attacker as a legit. After entering the network, attackers can attack the three interfaces, i.e., NorthBound, SouthBound, and East/WestBound. The application running in the application layer has special permission to access remote information resources and also controls the network functioning. Therefore, it makes applications a vulnerable point in the SDN network. Third-party organizations develop majority applications instead of Controller vendors. Thus, it makes authentication applications a major challenge in SDN [35]. Furthermore, SDN is an emerging technology; hence, it lacks standardized security mechanisms. Some techniques are available to ensure that the SDN programs are working properly. For example, VeriCon [36] is a verification system to determine the correctness of the programs executed by the Controller. The rest of the section presents a review on some security SDN techniques based on the Bloom Filter. Table 12.2 highlights the features and limitations of the security SDN techniques based on the Bloom Filter.

BloomStore [37] is a dynamic Bloom Filter-based secure rule-space management scheme. Each Controller has a public key and a Bloom Filter. The Bloom Filter saves all active OpenFlow switches. Also, all OpenFlow switches have a Bloom Filter, which saves all active Controllers. Each OpenFlow switch has a private key for authentication

TABLE 12.2 Features and Limitations of SDN security techniques.

Technique	Features	Limitations
BloomStore [37]	• Dynamic traffic management • Secure and dynamic communication between control and data planes • Double and partitioned hashing reduces overall computation and communication cost • Partitioned hashing results in less interhash collisions	• BloomStore maintains multiple Bloom Filters • Increase in level increases the size of the Bloom Filter • Query operation requires querying multiple Bloom Filters
Gupta et al. [38]	• Detects and attenuates DNS-based DDoS attacks • Third-party solution prevents changing the infrastructure of the organization • Acts as barrier against the malicious packets passing through the network of the organization	• One common Bloom Filter is not maintained for all organization by the ISP • Switch has to maintain one Bloom Filter for each organization • Implements standard Bloom Filter • Small switch size limits the number of Bloom Filters and their size
Li et al. [39]	• Security technique mainly for man-in-the-middle attacks • Insertion of packet information into Bloom Filter is a low overhead • Bloom Filter is a bit array, hence determining the difference in Bloom Filters is easy • Using another instance helps in hiding the presence of the monitor process from the attackers	• Technique fails if all switches are attacked • Technique fails if the attacker modifies packet information not stored in Bloom Filter • Attack detection is late • Controller has to handle many Bloom Filters • Bloom Filter gives high FPP in large flows • Big-size Bloom Filter increases overhead of the network
Xiao et al. [40]	• Detects abnormal flow • Abnormal flow information is stored in the Bloom Filter • Two modules help in both insertion of information and detection of abnormal attacks	• Implements standard Bloom Filter • Attack prevention is not performed by the switch • Attack prevention is undertaken by the Controller

and a unique ID. The global Controller maintains a master Bloom Filter which saves all possible entries (i.e., Switches and Controllers). BloomStore maintains multiple Bloom Filters. The size of the first Bloom Filter (say, m_0) is a multiple of the number of hash functions (say, k) where the multiple is a prime number (say, p). The Bloom Filter is divided into k buckets using partition hashing. A new Bloom Filter is constructed when the Bloom Filter crosses a threshold value. The size of the new Bloom Filter is $m_l = m_{l-1} + (\frac{m_0}{k})^l$ where l is the level of the Bloom Filter. A new threshold value is calculated for the new Bloom Filter. Insertion of ID is performed on the newest constructed Bloom Filter. In insertion operation, the ID is hashed by $(H_1(ID) + (a-1)H_2(ID)) \mod \eta^l$ where $H_1()$ and $H_2()$ are hash functions, $1 < a < k$, and η^l is a dynamic parameter whose initial value is p. The ID is hashed k times with different a values. The hashed values give the locations in the Bloom Filter. These locations are set to 1. Each Bloom Filter maintains a counter which stores the frequency of the queries whose result is present in the Bloom Filter. After a time interval, all counters are checked. If the counter value is less than a threshold value, then its respective Bloom Filter is refreshed. Initially, all the OpenFlow switches are inserted into the master Bloom Filter. This scheme follows two-step security checks, data plane and control plane. When an OpenFlow switch communicates with a Controller, the authentication of the switch is checked in the Bloom Filter of the Controller. If present, then the private key is matched. If it matches, then the second security check is followed. The ID and public key are concatenated and queried with the Controller. If it is present, then the switch is allowed to communicate. If the switch is not authenticated, then the new switch is added to the Controller using a three-way handshake protocol. First, the ID and private key of the switch are checked in the master Bloom Filter. If it is present, it verifies the switch is a legal network switch. Then the Controller shares its public key with the switch and inserts it into its Bloom Filter. If it is absent from the master Bloom Filter, the master Controller is consulted for authentication. During a query operation of a Controller's Bloom Filter, the most recent Bloom Filter is searched first. The ID is hashed, and the hash values give the location of cells in the Bloom Filter. These cells are checked; if all are 1, then the switch is present; otherwise, it is absent. The counter value determines the sequence of the Bloom Filter checking. The worst query time complexity is $O(kXa)$.

Gupta et al. [38] proposed a third-party solution against Domain Name System (DNS) based DDoS attacks. This technique implements Bloom Filter. An organization takes Internet services from multiple Internet Service Providers (ISPs). There is a Bloom Filter shared among all ISPs. Bloom Filter stores the legitimate packet information. The packet information is the source address, source port number, destination address, and destination port number. When the OpenFlow switch receives a packet, the packet information from the header is extracted. Then the information is hashed by k hash functions. The k hashed values give k cell locations in the Bloom Filter. These cell

locations are checked. If all are 1, then the packet is legitimate, and it is allowed to enter the network of the organization. Otherwise, the packet is malicious, and it is discarded.

Li et al. [39] proposed a lightweight security technique based on Bloom Filter. This technique is proposed for the IoT-Fog network. Bloom Filter is used to determine any modification in a packet. Bloom Filter is present in a switch. The packets of a flow are first forwarded to Bloom Filter. Bloom Filter inserts the packet information. Many switches are present along the path of a flow. All these switches insert the packet information in their Bloom Filter. Later, all the Bloom Filters are forwarded to the Controller, which verifies whether all the Bloom Filters are the same. If Bloom Filters are different, then it is concluded that the packet is modified. A monitor process is installed in the fog node to prevent packet modification at the first switch. This process also inserts the packet information in the Bloom Filter. Later the monitor process forwards the Bloom Filter to another instance of the Cloud. This instance is responsible for forwarding the Bloom Filter to the Controller. Using another instance helps in hiding the presence of the monitor process from the attackers. The Controller checks the Bloom Filters; when it finds any trace of attack, it informs the administrator about the possible switch(es). The switches do not perform attack detection. The technique performs a late attack detection.

Xiao et al. [40] proposed a real-time link attack detection system. It is a two-module detection framework. The two modules are Collector and Detector. Both modules can be in the Controller node, otherwise in different nodes. SDN has a flow table that stores flow statistical features such as packetCount, durationSeconds, byteCount, etc. When the utilization ratio of a link appears abnormal, the flow table is scanned to determine the abnormal link. When the values of the statistical features are more than their respective threshold values, the flow is considered abnormal. The statistical information of the abnormal link is stored in Bloom Filter., which is later forwarded to the detection module, responsible for sniffing the network. The possible suspect packets are collected, and their IP features are extracted. These packets are classified using Bloom Filter. The IP features are queried to the Bloom Filter. If Bloom Filter returns $True$, then the packet is an abnormal flow. Finally, the abnormal flow information is forwarded to the Controller.

12.7 Discussion

In this section, Table 12.3 highlights some features of all the SDN techniques based on the Bloom Filter reviewed in the chapter. Moreover, some discussion is presented on Bloom Filter in the context of SDN.

TABLE 12.3 Additional details regarding the SDN technique: Bloom Filter, name of Bloom Filter variant used in the technique; Reduce FP, technique used to reduce the number of false positives; Purpose, purpose of the technique.

Technique	Network Type	Bloom Filter	Reduce FP	Remark
BloomStore [37]	IP	Standard	No	Authentication
MPBF-TVHT [22]	IP	Standard	Hash table	Multi-query protocol
Challa et al. [23]	IP	Standard	No	Flow management
2MVeri [24]	IP	Standard	Calculation using characteristic matrix	Control-data plane consistency, Determines faulty switch
QBF [25]	IP	Standard	No	Cache management
Huang et al. [26]	IP	Standard	No	Packet classification
Parzyjegla et al. [27]	Publish/Subscribe	Standard	Multiple Bloom Filters	Notification routing
TSA-BF [28]	SDN-TCAM	Standard	No	Packet classification
Abbasi et al. [29]	Intelligent vehicular	Standard	No	Prefix matching
LBF [30]	IP	Standard	No	Packet classification
Gupta et al. [38]	IP	Standard	No	DNS-based DDoS attack
DCM [31]	IP	Standard	No	Flow monitoring
NDN-SF [32]	Information-centric networking	Standard	No	Packet forwarding
Li et al. [39]	IoT-Fog	Standard	No	Security
Xiao et al. [40]	Distributed data center	Standard	No	Abnormal flow detection
BloomFlow [33]	IP	Standard	No	Multicast
RRBF [41]	Information-centric networking	Standard	Multiple Bloom Filter	Load balancing

SDN is a centralized network where the hardware and software are separated to facilitate their individual development. The data and control planes are separated to enhance network management and control. However, the

Controller has to bear the huge burden of controlling and managing the network. Bloom Filter has a simple architecture; hence, it requires small memory. It favors deploying many Bloom Filters in the main memory. Bloom Filter also has simple operations with constant time complexity. Hence, Bloom Filter will be able to maintain the pace in filtering the huge packet traffic. Bloom Filter helps in reducing some of the burdens before analysis of the Controller.

Table 12.3 compares the various techniques proposed for SDN network. It highlights the network type it is proposed for, implemented Bloom Filter variant, the solution to reduce FPP, and the purpose fulfilled by the technique. One point that can be observed from Table 12.3 is that all techniques use standard Bloom Filter. Standard Bloom Filter is the simplest Bloom Filter; however, it has high FPP. Hence, some solutions need to be implemented to reduce FPP. On the other hand, other techniques have not implemented any additional solution to reduce FPP, except a few. Standard Bloom Filter can maintain the desired FPP, but the number of hash functions needs to be increased to achieve that. Of course, Bloom Filter enhances the performance, but Bloom Filter is an overhead in case of true positives. Thus, increasing the number of hash functions increases the computational time, which increases the overhead.

Flow monitoring is an important task. It helps the network nodes to prepare or determine the type of action to perform. Many techniques use Bloom Filter for flow monitoring. However, these techniques use a single Bloom Filter per flow. Hence, the techniques have to maintain many Bloom Filters. For example, MPBF-TVHT [22] uses multiple Bloom Filters for the storage of different packet information. SDN can explore Bloom Filter for multiple sets [42]. This is a variant of Bloom Filter, where a single Bloom Filter stores the information of different sets. The query operation to such Bloom Filter can respond approximately whether an item is present in one set but absent from another. Many such Bloom Filter variants are proposed [42]. Single Bloom Filter reduces the memory overhead. However, such Bloom Filters have high FPP and suffer from misclassification.

There is also another variant of Bloom Filter that can be explored. Some techniques use trees for faster searching. Furthermore, the techniques use Bloom Filter as nodes to speed up the searching further. However, Bloom Filters are unable to store data. Hence, either the node has both the Bloom Filter and the data or Bloom Filter and pointer to the data in the database. Thus, a large tree has to invest in a large memory space for many Bloom Filters. Therefore, a new variant of Bloom Filter is required to store the Bloom Filter tree information in a single data structure.

12.8 Conclusion

This chapter presents the role of Bloom Filter in the software-defined network. This chapter initially provides a short elaboration on the SDN architecture. SDN partitions the data and control plane for efficient network management and control. SDN is a centralized network where the Controller takes the responsibility of network management and monitoring. Bloom Filter's simple architecture helps in deploying multiple Bloom Filters in the main memory. Moreover, Bloom Filter's simple operations help in fast filtering and insertion of packet information for flow monitoring. This chapter highlights the great potential Bloom Filter has further to enhance the efficiency and performance of the SDN network.

References

[1] F. Bannour, S. Souihi, A. Mellouk, Distributed SDN control: survey, taxonomy, and challenges, IEEE Commun. Surv. Tutor. 20 (1) (2018) 333–354, https://doi.org/10.1109/COMST.2017.2782482.

[2] S. Nayak, R. Patgiri, 6G communication: envisioning the key issues and challenges, EAI Endorsed Trans. Internet Things (2020) 166959, https://doi.org/10.4108/eai.11-11-2020.166959.

[3] S. Nayak, R. Patgiri, T.D. Singh, Big computing: where are we heading?, EAI Endorsed Trans. Scalable Inf. Syst. 7 (27) (2020), https://doi.org/10.4108/eai.13-7-2018.163972.

[4] Y. Jarraya, T. Madi, M. Debbabi, A survey and a layered taxonomy of software-defined networking, IEEE Commun. Surv. Tutor. 16 (4) (2014) 1955–1980, https://doi.org/10.1109/COMST.2014.2320094.

[5] B.A.A. Nunes, M. Mendonca, X. Nguyen, K. Obraczka, T. Turletti, A survey of software-defined networking: past, present, and future of programmable networks, IEEE Commun. Surv. Tutor. 16 (3) (2014) 1617–1634, https://doi.org/10.1109/SURV.2014.012214.00180.

[6] S. Singh, S. Prakash, A survey on software defined network based on architecture, issues and challenges, in: 2019 3rd International Conference on Computing Methodologies and Communication (ICCMC), 2019, pp. 568–573.

[7] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, J. Turner, OpenFlow: enabling innovation in campus networks, SIGCOMM Comput. Commun. Rev. 38 (2) (2008) 69–74, https://doi.org/10.1145/1355734.1355746.

[8] M. Smith, M. Dvorkin, Y. Laribi, V. Pandey, P. Garg, N. Weidenbacher, Opflex control protocol, IETF, Apr. 2014, https://datatracker.ietf.org/doc/html/draft-smith-opflex-00.

[9] L. Hao, B. Ng, Y. Qu, Dynamic optimization of neighbor list to reduce changeover latency for Wi-Fi networks, in: Proceedings of the 2017 International Conference on Telecommunications and Communication Engineering, ICTCE '17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 20–24.
[10] R. 5812, Internet research task force, https://tools.ietf.org/html/rfc5812. (Accessed 15 September 2020).
[11] E. Kaljic, A. Maric, P. Njemcevic, M. Hadzialic, A survey on data plane flexibility and programmability in software-defined networking, IEEE Access 7 (2019) 47804–47840, https://doi.org/10.1109/ACCESS.2019.2910140.
[12] ONF, Open networking foundation, https://www.opennetworking.org/. (Accessed 15 September 2020).
[13] S. Fichera, M. Gharbaoui, P. Castoldi, B. Martini, A. Manzalini, On experimenting 5G: testbed set-up for SDN orchestration across network cloud and IoT domains, in: 2017 IEEE Conference on Network Softwarization (NetSoft), 2017, pp. 1–6.
[14] P.T. Congdon, P. Mohapatra, M. Farrens, V. Akella, Simultaneously reducing latency and power consumption in OpenFlow switches, IEEE/ACM Trans. Netw. 22 (3) (2014) 1007–1020, https://doi.org/10.1109/TNET.2013.2270436.
[15] P. Sun, J. Li, Z. Guo, Y. Xu, J. Lan, Y. Hu, Sinet: enabling scalable network routing with deep reinforcement learning on partial nodes, in: Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos, SIGCOMM Posters and Demos '19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 88–89.
[16] K. Nisar, E.R. Jimson, M.H.A. Hijazi, I. Welch, R. Hassan, A.H.M. Aman, A.H. Sodhro, S. Pirbhulal, S. Khan, A survey on the architecture, application, and security of software defined networking: challenges and open issues, Int. Things 12 (2020) 100289, https://doi.org/10.1016/j.iot.2020.100289.
[17] Z. Guo, S. Zhang, W. Feng, W. Wu, J. Lan, Exploring the role of paths for dynamic switch assignment in software-defined networks, Future Gener. Comput. Syst. 107 (2020) 238–246, https://doi.org/10.1016/j.future.2019.12.008.
[18] Q. Gao, W. Tong, S. Kausar, L. Huang, C. Shen, S. Zheng, Congestion-aware multicast plug-in for an SDN network operating system, in: Softwarization and Caching in NGN, Comput. Netw. 125 (2017) 53–63, https://doi.org/10.1016/j.comnet.2017.04.050.
[19] K. Nisar, H. Hasbullah, A.M. Said, Internet call delay on peer-to-peer and phone-to-phone VoIP network, in: 2009 International Conference on Computer Engineering and Technology, vol. 2, 2009, pp. 517–520.
[20] M. Cheikhrouhou, J. Labetoulle, Efficient instrumentation of management information models with SNMP, in: Proc. IEEE/IFIP NOMS, 2000.
[21] W. Stallings, Software-defined networks and OpenFlow, Internet Protoc. J. 16 (1) (2013) 2–14.
[22] D. Yuan, X. Yang, X. Shi, B. Tang, Y. Liu, Multi-protocol query structure for SDN switch based on parallel Bloom filter, in: 2014 International Conference on Information and Communication Technology Convergence (ICTC), 2014, pp. 206–211.
[23] R. Challa, Y. Lee, H. Choo, Intelligent eviction strategy for efficient flow table management in OpenFlow switches, in: 2016 IEEE NetSoft Conference and Workshops (NetSoft), 2016, pp. 312–318.
[24] K. Lei, K. Li, J. Huang, W. Li, J. Xing, Y. Wang, Measuring the control-data plane consistency in software defined networking, in: 2018 IEEE International Conference on Communications (ICC), 2018, pp. 1–7.
[25] K. Sasaki, A. Nakao, Packet cache network function for peer-to-peer traffic management with Bloom-filter-based flow classification, in: 2016 18th Asia-Pacific Network Operations and Management Symposium (APNOMS), 2016, pp. 1–6.
[26] D.-F. Huang, C. Chen, M. Thanavel, Fast packet classification on OpenFlow switches using multiple R*-Tree-based bitmap intersection, in: NOMS 2018–2018 IEEE/IFIP Network Operations and Management Symposium, 2018, pp. 1–9.
[27] H. Parzyjegla, C. Wernecke, G. Mühl, E. Schweissguth, D. Timmermann, Implementing content-based publish/subscribe with OpenFlow, in: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC'19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1392–1395.
[28] S.-C. Kao, D.-Y. Lee, T.-S. Chen, A.-Y. Wu, S.-C. Kao, D.-Y. Lee, A.-Y. Wu, T.-S. Chen, Dynamically updatable ternary segmented aging Bloom filter for OpenFlow-compliant low-power packet processing, IEEE/ACM Trans. Netw. 26 (2) (2018) 1004–1017, https://doi.org/10.1109/TNET.2018.2813425.
[29] M. Abbasi, H. Rezaei, V.G. Menon, L. Qi, M.R. Khosravi, Enhancing the performance of flow classification in SDN-based intelligent vehicular networks, IEEE Trans. Intell. Transp. Syst. 22 (7) (2021) 4141–4150, https://doi.org/10.1109/TITS.2020.3014044.
[30] M. Yang, D. Gao, C.H. Foh, Efficient packet classification with learned Bloom filter in software-defined networking, in: ICC 2021 – IEEE International Conference on Communications, 2021, pp. 1–6.
[31] Y. Yu, C. Qian, X. Li, Distributed and collaborative traffic monitoring in software defined networks, in: Proceedings of the Third Workshop on Hot Topics in Software Defined Networking, HotSDN'14, Association for Computing Machinery, New York, NY, USA, 2014, pp. 85–90.
[32] A. Kalghoum, S.M. Gammar, L.A. Saidane, Towards a novel forwarding strategy for named data networking based on SDN and Bloom filter, in: 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA), 2017, pp. 1198–1204.
[33] A. Craig, B. Nandy, I. Lambadaris, P. Koutsakis, BloomFlow: OpenFlow extensions for memory efficient, scalable multicast with multi-stage Bloom filters, Comput. Commun. 110 (2017) 83–102, https://doi.org/10.1016/j.comcom.2017.05.018.
[34] P. Elias, Universal codeword sets and representations of the integers, IEEE Trans. Inf. Theory 21 (2) (1975) 194–203.
[35] A. Akhunzada, A. Gani, N.B. Anuar, A. Abdelaziz, M.K. Khan, A. Hayat, S.U. Khan, Secure and dependable software defined networks, J. Netw. Comput. Appl. 61 (2016) 199–221, https://doi.org/10.1016/j.jnca.2015.11.012.
[36] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, A. Valadarsky, VeriCon: towards verifying controller programs in software-defined networks, SIGPLAN Not. 49 (6) (2014) 282–293, https://doi.org/10.1145/2666356.2594317.
[37] A. Singh, S. Batra, G.S. Aujla, N. Kumar, L.T. Yang, Bloomstore: dynamic Bloom-filter-based secure rule-space management scheme in SDN, IEEE Trans. Ind. Inform. 16 (10) (2020) 6252–6262, https://doi.org/10.1109/TII.2020.2966708.
[38] V. Gupta, A. Kochar, S. Saharan, R. Kulshrestha, DNS amplification based DDoS attacks in SDN environment: detection and mitigation, in: 2019 IEEE 4th International Conference on Computer and Communication Systems (ICCCS), 2019, pp. 473–478.
[39] C. Li, Z. Qin, E. Novak, Q. Li, Securing SDN infrastructure of IoT-Fog networks from MitM attacks, IEEE Int. Things J. 4 (5) (2017) 1156–1164, https://doi.org/10.1109/JIOT.2017.2685596.
[40] P. Xiao, Z. Li, H. Qi, W. Qu, H. Yu, An efficient DDoS detection with Bloom filter in SDN, in: 2016 IEEE Trustcom/BigDataSE/ISPA, 2016, pp. 1–6.
[41] Ö. Szabó, C. Simon, Round-robin Bloom filters based load balancing of packet flows, Infocommun. J. 8 (3) (2016) 13–19.
[42] L. Waikhom, S. Nayak, R. Patgiri, A survey on Bloom filter for multiple sets, in: Modeling, Simulation and Optimization, Springer, 2021, pp. 775–789.

CHAPTER

13

Impact of Bloom Filter in wireless network

13.1 Introduction

Wireless communication refers to the transmission of information among the nodes that are not connected physically. The communication medium in wireless is electromagnetic waves; commonly used electromagnetic waves are radio waves. Some wireless applications are cellular telephones, GPS, satellite television, smart homes, etc. The nodes are mostly small devices having limited energy, resources, etc. Designing routing protocols for such a network is challenging because apart from an efficient and high-performance technique, low energy and resources factors also need to be considered [1]. Various factors that influence the routing protocol design are wireless means of communication, energy storage, data reporting and aggregation model, degree of fault tolerance, and node deployment approach.

Wireless sensor networks (WSNs) [2] are networks consisting of sensors for collecting information about their environment. Some examples of its application are environmental monitoring (e.g., temperature, humidity) [3,4], file exchange, etc. A mobile ad-hoc network (MANET) [5] is a network consisting of mobile and wireless devices. Typically, MANET networks are constructed for a specific purpose and for a short time. A MANET network focuses on the area of interest. A few examples of its applications are battlefield communication [6], search and rescue [7], etc. Vehicular Ad hoc Network (VANET) [8] is a special case of MANET where the nodes are vehicles, Road-Side Units (RSU), and On-Board Unit (OBU) [9]. VANET aims to collect traffic data and transfer information to the vehicles for safe and fast passage. The Internet-of-Things (IoT) consists of wired and wireless devices. It can be for public or private usage. Some applications of IoT are smart homes [10], smart schools [11], etc.

All wireless communication has the same issues as mentioned above. The focus is on Bloom Filter to solve these issues. Hence, this chapter elaborates on the role of Bloom Filter to enhance the performance of wireless communication: WSN, MANET, VANET, and IoT. Moreover, the chapter also presents the review of many techniques based on Bloom Filter to solve wireless communication issues.

13.2 Wireless sensor networks

Wireless sensor networks (WSNs) [2] consist of various sensors as nodes. The main focus of a WSN is to collect environmental information. The sensors concurrently sense, collect, process, and transfer the data. Some examples of WSN applications are disaster and health care areas providing relief, file exchange, environmental monitoring (e.g., temperature, humidity). WSNs are low in installation cost, and small sensors can be distributed over a wider area and are fault tolerant. However, WSN has lots of issues such as error-prone transmissions, low bandwidth, fewer resources due to small size, etc. One main issue is power supply [12]. The sensors have a limited power supply; hence they are unable to provide complex processing. The rest of the section presents a review of various techniques based on the Bloom Filter to enhance the performance of the WSN. Table 13.1 highlights the features and limitations of the WSN techniques.

Choi et al. [13] proposed a code-based discovery protocol for cellular device-to-device (D2D) communications. It efficiently determines the nodes with similar applications among many devices. The scheme is designed for OFDMA-based cellular networks. In this network, the time and frequency radio resources are allocated to the devices by a base station which is the central node. The time is partitioned into slots, and frequency is partitioned into subchannels. When a node becomes visible within a certain distance range, the discovery protocol detects the node and identifies other nodes having the same applications. The discovery protocol determines any changes, i.e., entry or exit of a node within the protocol coverage for each time slot. A discovery code has a set consisting of all application names in a

https://doi.org/10.1016/B978-0-12-823520-1.00019-5

Copyright © 2023 Elsevier Inc. All rights reserved.

TABLE 13.1 Features and Limitations of Wireless Sensor Network Techniques.

Technique	Features	Limitations
Choi et al. [13]	• Efficiently determines the nodes with similar applications among many devices • Bloom Filter maximizes the bandwidth efficiency • Application information is send to the nodes having same applications • Avoids unnecessary broadcasting of messages • Performance of Bloom Filter is better compared to hashing	• Standard Bloom Filter • Bloom Filter gives high FPP in case a large number of devices are present within the coverage area
Sanchez-Hernandez et al. [14]	• CBF protects the topological information • CBF operation is fast • Dehydration operation helps in determining the feasibility of transmitting the packets to a destination • CBF reduces the memory requirement for collecting all nodes' topological information	• When the network is tightly connected then the overhead of the protocol increases • Protocol has broadcast storm problem • CBF requires more memory, hence, consumes more memory in the packet
Talpur et al. [15]	• Implementation of Bloom Filter reduces the energy consumption • Bloom Filter helps in full network coverage • Bloom Filter helps in fast determination of common neighbor nodes • Reduces communication overhead by reducing the number of packet transmission	• Receiving duplicate packets is not completely prevented • Implements standard Bloom Filter
Moualla et al. [16]	• Saves bandwidth • Reduces signaling overhead • Privacy is maintained by implementing Bloom Filter	• Implements standard Bloom Filter • A node does not save the most popular data if it is dissimilar with the node having most popular data
OSR [19]	• Reduces length of source route field in packet header • Enhances scalability • Enhances reliability • Introduces opportunistic routing into the source routing • Bloom Filter compresses source-route path • In case of failure of a node, other alternative parent nodes are selected to resume the downward forwarding • Does not require false positive recovery scheme	• During failure of a node, OSR selects an alternative which increases delay of the communication • Implements standard Bloom Filter
BFRP [1]	• Replaces routing table by Bloom Filter • Bloom Filter takes low memory • Considering Bloom Filter with fewer 1s reduces FPP and longest path traversal • Some sensors sleep to reserve energy	• Implements dynamic Bloom Filter which has many Bloom Filters • Implements standard Bloom Filter
BFRE [17]	• Reduces redundant traffic data • Bloom Filter helps in compression of information • Reduces traffic • Increase in the Bloom Filter size reduces the packet retransmission	• Increase in the number of data chunks increases the overhead of Bloom Filter transmission • High fingerprint matching by Bloom Filter inversely affects the performance • Implements standard Bloom Filter
CSPR [18]	• Storing both path length and hop count in the Bloom Filter differentiates among packets having same source node but takes different paths • Invulnerable to topology dynamics and lossy links • Implements optimization techniques to reduce the representation space and reduce the sparsity of unrecovered path vectors • Path reconstruction depends on a small number of received packets	• Introduces only small and fixed overhead in annotating each packet • Implements standard Bloom Filter

node. Bloom Filter performs mapping of the application name to discovery code. Initially, all cells of Bloom Filter are set to 0. The application names are inserted into the Bloom Filter. An application name is hashed by k hash functions. The hashed values provide the cell position, which is set to 1. When a node receives discovery code from another node, it checks whether it has any common application(s). While checking for an application name in the discovery

code, the same procedure is followed to obtain the cell positions. If all cell positions are 1, that application also runs in the node. Otherwise, if at least one cell position is 0, then that application does not run in the node. Hashing is also used for mapping the application name to discovery code. However, the performance of the discovery protocol with Bloom Filter is better compared to hashing with respect to false positive probability.

Sanchez-Hernandez et al. [14] presented a routing protocol for broadcasting topological information using CBF. CBF contains the topological information which is collected while traversing the nodes. The identity of a node is inserted into CBF. All CBFs are of the same size. The architecture of CBF is the same (i.e., each cell is associated with a counter). However, during insertion, instead of incrementing the counter of the cell by 1, the counter is set to a fixed value (say, c). In the CBF, three operations are performed: insertion, degradation, and addition. In a degradation operation, the counter value of CBF is decremented by 1 when CBF is becoming stale or it is traversing away from the source node. In addition to operation, a new CBF is constructed by combining two CBFs. Each cell of the new CBF is assigned to the maximum value among the same corresponding cells of both CBFs. Every node maintains two Bloom Filters. Let the Bloom Filters be BF_1 and BF_2; BF_1 contains the node's topological information, which remains constant; BF_2 also contains the node's topological information, but it changes based on degradation due to time and received update. Every node broadcasts its BF_2 periodically after a fixed time interval. When a node receives a Bloom Filter from the neighboring node, it performs the addition operation to merge its own and received information. Also, the receiver node performs the degradation operation. First, the node performs a degradation operation on its own BF_2. Then performing XOR operation between the received BF_2 and its modified BF_2. Moreover, every node's BF_2 is modified by applying degradation operation periodically. The receiver node also calculates the new delivery probability. A node forwards a packet to its neighbor if the sum of the probability of reaching the destination node and a constant threshold is more than the probability of the current node. If the node calculates a higher probability, the receiver node adds its identifier into a list. The list is an array of identifiers of nodes present in the packet. The receiver node also creates a new list and copies the identifiers from the received packet. Then the receiver node sends a request to the sender node with a list which is an array of packet identifiers. It indicates the list of identifiers of nodes whose messages the receiver node wants. Then, the sender sends back the messages.

Talpur et al. [15] presented a broadcasting protocol based on Bloom Filter. This protocol reduces the number of packet transmissions, which reduces energy consumption. The basic idea of the protocol is that a node sends the broadcasting packet to its neighbors. It minimizes the receiving of duplicate packets by a single node. The network has an Initial Source Node (ISN) and an end node. The selection of the ISN is performed randomly. The end node is a base station responsible for connecting the sensor node to the central network. The central node performs processing and makes decisions. In the protocol, first, the neighbor discovery phase is implemented. After this phase, a tree is constructed to discover the neighbor nodes. All nodes construct a neighbor filter consisting of the list of their neighbors. The broadcasting starts from the ISN transmitting two Bloom Filters, namely, Neighbor Filter (NF) and Urgent Member Filter (UMF), to one-hop neighbors. NF contains the list of neighbors, and UMF contains the list of nodes that require urgent services from the central node. Initially, the UMF is empty. Based on some condition or threshold level, the ISN becomes the urgent node and adds itself into the UMF. Then it forwards the filters to its neighbors. When the neighbor node sends the broadcasting packet, it first queries NF to prevent sending the packet to the node already sent by ISN. The neighbor node adds itself to the UMF in case it requires urgent services. Otherwise, the neighbor forwards its NF, and the modified UMF received from ISN. This broadcasting of the packets continues till the packet reaches the end node.

Moualla et al. [16] dealt with a cache management algorithm based on the Bloom Filters. The Bloom Filter is used for storing the user cache summaries. The cache is maintained based on the social distance among users and the popularity of the data. The algorithm is proposed for content-centric networking. The popularity of the data is determined by knowing the cached content of other nodes. This is achieved by using Bloom Filter for storing the cache summary. Every node periodically broadcasts its Bloom Filter. Social proximity among the users is based on similar cache content. After receiving all the Bloom Filters of other nodes, the receiver node determines the nodes with high similarity. First, the receiver node determines peer nodes, which refer to the nodes with high similarity of data with the receiver node. Other nodes are assigned with a score. The receiver node queries its cache content with every Bloom Filter. With every match of the cached content, the score is incremented. The nodes with the highest score are concluded as the peer nodes. Then the receiver node determines the popular files in the cache. The receiver node checks the Bloom Filters of the peer nodes. The file is assigned a score. With every matching file, the file score is incremented. The files with the highest scores are kept in the cache, and other files are considered during cache replacement.

Opportunistic Source Routing protocol (OSR) is proposed to enhance scalability and reliability in heterogeneous WSN. OSR uses Bloom Filter to encode the downward source route in OSR efficiently. OSR uses an adaptive path Bloom filter where the length of Bloom Filter depends on the number of hops of the route. If the number of hops is greater than the maximum length of Bloom Filter possible for an encoded source route in bytes (say, L), then the Bloom Filter array size is $8L$. Otherwise, the size is eight times the number of hops. Instead of storing complete network information for downward routing decisions, due to OSR, a node stores the one-hop direct child node information. It reduces memory requirements for storing complete network information. When a packet traverses the network, the node's address is inserted into the Bloom Filter. When a node receives the packet, the packet is queried into its child node address in the Bloom Filter. If present, then the packet is forwarded to that node. If the Bloom Filter gives a false positive, and the node is forwarded to another child of the node, it becomes another route for the packet. Hence, OSR does not require a false positive recovery scheme.

Bloom Filter-based routing protocol (BFRP) [1] is a routing protocol for large-scale sensor networks. The network replaces the routing table with a Bloom Filter. BFRP uses a dynamic Bloom Filter. Dynamic Bloom Filter consists of many standard Bloom Filters. Initially, one Bloom Filter is present. When the FPP of the Bloom Filter crosses a threshold value, a new Bloom Filter is added in the dynamic Bloom Filter. The new Bloom Filter becomes active, i.e., items are inserted into this Bloom Filter. In the case of query operation, all Bloom Filters are searched. If any of the Bloom Filters returns $True$, then the item is present in the dynamic Bloom Filter. Every node maintains a Bloom Filter instead of a routing table. The Bloom Filter contains the nodeID of the neighboring node. Nodes need to remain updated about their neighbors. Hence, nodes periodically send update messages to indicate neighboring nodes to forward their Bloom Filter. After receiving the Bloom Filter, OR operation is performed between the present and received Bloom Filters to update the new information. The sender node has the Bloom Filter of the destination node during packet transmission. The sender queries its neighbor node's nodeID to the Bloom Filter. If Bloom Filter returns $True$ for any node, then the packet is forwarded to that node. If multiple neighbor nodes match, then the packet is forwarded to the node whose Bloom Filter has fewer 1s. It reduces FPP and avoids long path traversal. BFRP implements a clustering algorithm to cluster nodes to prevent costly long-distance transmission. Each node calculates its coverage-aware cost based on a formula. When a node wants to become the cluster head, it broadcasts an announcement message to inform others of it being the cluster head. The node waits for a time period called the activation period, whose length is equal to the coverage-aware cost. When other nodes receive this message, they send their announcement message and wait for their respective activation period. If a node does not receive any other announcement message within the activation period, it becomes a cluster node because it has the least coverage-aware cost. In the case of multiple nodes having the same coverage-aware cost, a node is randomly selected. After selection, other nodes send a JOIN message to join with the nearest cluster head. The JOIN message contains the Bloom Filter of the node. After collecting all Bloom Filters, the cluster head performs OR operation to merge all the information into a single Bloom Filter.

Bloom-Filter-aided Redundancy Elimination (BFRE) [17] is a traffic data filtering technique. The cache consists of data chunks, and it has a fingerprint as metadata. A reference value is added to the last bits of the fingerprint. Each set of data chunks has a Bloom Filter. The Bloom Filter stores the fingerprint of the data chunk set. Initially, the sender sends the original packet to the receiver. The receiver selects some data chunks from the packets and determines their fingerprint. The fingerprints are queried to all the Bloom Filters in the data chunk set. The fingerprint is hashed by k hash functions. The hash value provides the cell location in the Bloom Filter. If all cells are 1, the fingerprint is present in the Bloom Filter. If a Bloom Filter returns $True$ for several fingerprints, then that Bloom Filter is forwarded to the sender. The data chunk is forwarded to the expected set buffer. If no Bloom Filter is found, the sender transmits the original data. After the sender receives the Bloom Filter, it selects a chunk of data and determines its fingerprint. The fingerprint is queried in the Bloom Filter. If Bloom Filter returns $True$, then the data chunk is replaced by the metadata of the data chunk. Otherwise, the original data chunk is stored in the packet. Finally, the encoded packet is transmitted to the receiver. After receiving the encoded packet, it is decoded using the corresponding fingerprint on the receiver side. The fingerprint in the packet is replaced by the data chunk present with the receiver. If the receiver cannot find the corresponding data chunk, then the original data is requested by the receiver.

Compressive Sensing based Path Reconstruction (CSPR) [18] is a technique for path reconstruction in WSN. A Bloom Filter is embedded in the packet to store the nodeID and corresponding hop count information. CSPR uses a 3-tuple key to determine the path of the packet. The 3-tuple key is the source address, path length, and Bloom Filter. When the sender sends the packet, the path length and Bloom Filter are set to 0. When a node receives a packet, the path length is incremented. The product of the nodeID and hop count is hashed by k hash functions. The k hashed values provide the bit locations in the Bloom Filter, which are set to 1. The data station maintains a database

storing the path groups. Each entry in the database represents a path group indexed by the 3-tuple key. A path group consists of paths classified as a group. The paths are classified using the 3-tuple keys. First, the packet classification is performed based on the source address. The path groups are further classified based on the path length. The paths having the same source and path length are differentiated using the Bloom Filter. Packets with the same 3-tuple key travel the same path; hence, they are considered in the same path group. When a base station receives a packet, the 3-tuple key is extracted from the header and matched in the database. If a match is found, then the algorithm retrieves the path. If a match is found, but the path is not ready, then after accumulating a certain number of packets, the CSPR performs the path construction. Also, if no match is found, a new entry is created in the database.

13.2.1 Security

WSN uses the wireless medium for communication which makes the WSN vulnerable [2]. In a wired network, a node or device needs to follow some protocol for joining the network for communication. However, in WSN, a device located within the wireless signal range is eligible to communicate in the network. Therefore, WSN is prone to many attacks such as spoofing attack [20], eavesdropping attack [21], Man-in-the-Middle (MitM) attack [22], message falsification/injection attack [23], etc. Hence, WSN should focus on authenticity, confidentiality, integrity, and availability. Every device needs to authenticate itself before joining the network. The data in the network should be shared among the legal users to maintain confidentiality. A user should be given a key when joining the network to access encrypted data. The data should be stored in an encrypted form to protect the data from modification or deletion from malicious devices. Availability depends on the protection of all devices within the network to prevent denial of services. It also depends on the capacity of a WSN network to support many devices. Bloom Filter is also considered for improving the security of WSN. The rest of the section presents a review of various Bloom Filter-based techniques that boost WSN security. Table 13.2 presents the features and limitations of the WSN security techniques.

TABLE 13.2 Features and Limitations of Wireless Sensor Network Security Techniques.

Technique	Features	Limitations
BFAN [24]	• Bloom Filter with multiple MACs in delayed key disclosure schemes reduces the risk of DoS • All commitment keys are aggregated into a single vector • Aggregation of commitment keys into a single Bloom Filter reduces communication cost • XOR-Based Bloom Filter reduces collision • Reduces authentication delay • Authentication operation immediately starts after receiving packet instead of waiting for arrival of all commitment keys	• Requires additional energy for performing XOR operation • Associating a counter with each cell increases the memory requirement of the Bloom Filter
Kumar and Prasad [25]	• Bloom Filter is transmitted along with the message due to the small size • Bloom Filter helps in determining incorrect paths • Bloom Filter helps in provenance information verification	• Incorrect path is detected later by the controller • Implements standard Bloom Filter • Incorrect provenance information is detected later by the controller
Thesnim and Nithin [26]	• Determines the malicious node using the energy consumed by a node • Bloom Filter size determines malicious attack	• Compromises security • Reduces performance • Implements standard Bloom Filter
Bhatti and Saleem [27]	• Secret key generation depends on the shared Bloom Filter • Bloom Filter compresses the received frame information of a node • Bit array of Bloom Filter helps in performing intersection operation to determine the common frames • Small-sized Bloom Filter requires low bandwidth	• Bloom Filters are broadcasted • Implements standard Bloom Filter • FPP of Bloom Filter depend on the number of frames inserted • High traffic network requires to broadcast the Bloom Filter very frequently to prevent saturation of Bloom Filter

Bloom-Filter-based AuthenticatioN (BFAN) [24] is an authentication scheme based on Bloom Filter for authenticating sensors. This scheme assumes that the base station is responsible for the authentication process. It has sufficient power and memory to securely communicate with all sensors. Every sensor has a Bloom Filter and k hash functions. The sensors have a fixed location. The commitment keys are inserted into a Bloom Filter. The commitment keys are

aggregated and inserted into the Bloom Filter. The MAC and aggregated keys are transmitted in the same packet to reduce the number of packet forwarding. The Bloom Filter has m cells where each cell has an associated counter. The counter records the number of 1s in the cell. Initially, all cells of the Bloom Filter are set to 0. The key is hashed by the k hash functions. The hashed value provides the cell locations in the Bloom Filter. These locations and their respective counters are set to 1. Collision in Bloom Filter occurs when the hash functions generate the same cell locations for two different commitment keys. The XOR operation is performed on the hashed values to prevent collision. The counter is checked when the hash functions generate the same hash values. If the counter is greater than 1, then XOR operation is performed between these hash values. The receiver node also constructs a Bloom Filter of commitment keys. If both the receiver's Bloom Filter and the received Bloom Filter are the same, then the sender is authenticated.

Kumar and Prasad [25] presented a technique for secure delivery of packets by implementing Bloom Filter for encoding and verification of provenance. Every packet has a Bloom Filter, along with the message. The source node constructs the Bloom Filter and sets all cells of the Bloom Filter to 0. When a packet reaches a node, the node inserts its provenance information into the Bloom Filter. Finally, the packet reaches the controller. Then the controller performs verification. It queries Bloom Filter to determine whether the packet followed the correct path. In case the path is incorrect, then the controller modifies the network to remove the path followed by the packet. For provenance verification, the controller constructs a Bloom Filter. The controller uses its knowledge and inserts the provenance information of the nodes traversed by the packet. Then both the Bloom Filters are checked. If both are the same, then the provenance verification is successful.

Thesnim and Nithin [26] introduced trust evaluation to detect provenance forgery and packet drop attacks in the WSN. The technique uses an in-packet Bloom Filter (iBF) to store and transfer the provenance information. The base station records the provenance of all packets; hence, it reduces performance and compromises security. Bloom Filter is used for encoding of the provenance information. Each node is represented by a vertexID. The sender node constructs an iBF with all bits set to 0. It inserts its own vertexID into the iBF. Then the packet is forwarded to the next node. The node which receives the packet inserts its vertexID into the iBF. The base station finally receives the packet and retrieves the iBF. The base station performs two processes: provenance verification and provenance collection. The base station has the knowledge of the path followed by the packet. It queries the vertexID of the nodes present in the path for provenance verification. If iBF returns $True$ for all nodes, then no attack occurred. In case of a node failure, the provenance collection is performed. In this process, the base station uses the provenance graph and iBF to determine the new path. The provenance verification process is again performed after provenance collection. The presence of a malicious node increases the Bloom Filter size. Hence, the big-sized Bloom Filter indicates an attack in the network.

Bhatti and Saleem [27] proposed a low-cost and robust technique for key generation in MANET. Bloom Filter helps to enhance the technique while maintaining many constraints such as low energy and resources. The nodes in the network capture frames. After a certain number of frames, the frames are inserted into the Bloom Filter. Then the nodes broadcast their Bloom Filter to all other nodes. After receiving the Bloom Filters, every node performs an intersection operation on Bloom Filters, and the result is stored in a new Bloom Filter. Then all frames are queried to the new Bloom Filter to determine common frames among all frames. All the frames common to all nodes are saved in a set called SKGframes. A secret key is generated using the SKGframes by a non-key cryptographic hash function. The hash function is pre-shared by all legal nodes. Illegal nodes are not permitted to broadcast their Bloom Filter. Without sharing Bloom Filter, a node cannot obtain the secret key.

13.3 Mobile ad-hoc networks

Mobile ad-hoc network (MANET) [5] is an instant network consisting of mobile and wireless devices communicating with each other within a short range. The main purpose of MANET is to provide communication service anytime and anywhere without specifications to infrastructure. MANET applications are battlefield communication, search and rescue, robot-data acquisition, and mini-site operation. Wireless Sensor Network (WSN) is a network of small devices or nodes called sensors connected to form a centralized network. Its primary purpose is to sense the area of interest. Some applications of WSN are object tracking, fire detection, traffic monitoring, and biomedical applications. MANET remains close to humans, whereas WSN collects data about the environment. MANET is distributed, and WSN is centralized. WSN has a low data rate, but MANET has a high quality of service to transmit high-resolution data [28].

Vehicular Ad hoc Network (VANET) [8] is a MANET where the device or nodes are vehicles, Road-Side Units (RSU), and On-Board Unit (OBU) [9]. The primary purpose of VANET is to determine effective and efficient routes for vehicles with the aim of providing fast and safe passage. The RSU collects data about the traffic and forwards the data to the vehicles. The communication in VANET includes Vehicle-to-Vehicle (V2V), Vehicle-to-Infrastructure (V2I), and Infrastructure-to-Infrastructure (I2I). Some applications of VANET are parking and toll collection services, collision avoidance, sharing messages, pictures, and videos, traffic control, and safe driving. Bloom Filter is an excellent choice for the fast processing of messages. In the following section, a review is presented about various Bloom Filter-based techniques to increase the performance of VANET.

The rest of the section presents a review of various techniques based on the Bloom Filter to enhance the performance of the MANET and VANET. Table 13.3 presents the features and limitations of the MANET and VANET techniques.

TABLE 13.3 Features and Limitations of Mobile Ad-hoc Network and Vehicular Ad-hoc Network Techniques.

Technique	Features	Limitations
Oigawa and Sato [29]	• Bloom Filter reduces the number of IERP control packets • Bloom Filter is maintained by the peripheral nodes • Bloom Filter helps in determining the destination node • Organizing the nodes in a tree structure reduces the packet transmission	• Tree construction is an overhead • Tree structure of zone is dynamic • Network tree is constructed after a fixed time period • False positive response leads to loss of packet or long route • Short route also depends on the selection of the root node • Implements standard Bloom Filter
Bao et al. [30]	• Lightweight authentication scheme • Prevents active attacks • Maintains privacy • Bloom Filter recognizes illegal nodes • Implementing Bloom Filter compresses the information • Broadcasting of Bloom Filter has low cost	• Implements standard Bloom Filter • FPP results in labeling of a node as illegal • Requires regular broadcast of Bloom Filter with newly inserted illegal nodes
Jin and Papadimitratos [31]	• Pseudonym validation • Reduces computational resources used for safety and time-critical operations • Reduces communication overhead • Compressed BF-deltas reduce communication overhead during Bloom Filter update • Fake Pseudonym List helps in detecting false positive response of Bloom Filter • PCA maintains CBF to perform insertion and deletion operations • Vehicles download a standard Bloom Filter to reduce communication overhead	• Due to false positives of Bloom Filter, the technique performs another check on the pseudonym • Bloom Filter is updated after joining of a certain number of vehicles in the network

Oigawa and Sato [29] proposed a Bloom Filter-based zone routing protocol. The technique follows a tree structure of nodes for transmitting query messages. Each node has knowledge of its parent node and child node. The root node transmits a TreeQuery packet to the nodes present within the zone. The first node which receives the packet is made the parent node. This parent node receives the packet and forwards it to all the nodes within its zone. The root waits for a TreeReply message for a fixed time interval. After receiving Bloom Filters from the children nodes, the parent node inserts all information of itself and children nodes except the destination node into a single Bloom Filter. Then the Bloom Filter is embedded into the TreeAck packet and forwarded to the children nodes. The network is dynamic; hence, the tree construction is repeated after a fixed time interval. When a sender wants to send a packet, first, it determines the route to the destination node. The sender creates a RouteQuery packet and queries the destination nodeID into the Bloom Filters of parent and children nodes. If any Bloom Filter returns $True$, then the RouteQuery packet is forwarded in that direction. In case no Bloom Filter returns $True$, then the RouteQuery packet is forwarded to the neighbor tree node. The neighbor node follows the same procedure to determine the destination node. The destination node returns a RouteReply packet. The Route Reply packet contains the route information. After receiving the RouteReply packet, the packet from the sender follows the route mentioned in the RouteReply packet.

Bao et al. [30] presented a Bloom Filter-based lightweight authentication scheme using Timed Efficient Stream Loss-Tolerant Authentication (TESLA). TESLA is used for broadcasting packets. Bloom Filter authenticates the one-way key chain (OKC). OKC is required to authenticate packets received in the future from the node. A node sends its first packet with its private key. Roadside unit (RSU) constructs a log to store all the certificates received from all vehicles. The log is used to construct a Bloom Filter. When a vehicle joins the network, it requests the Bloom Filter. The RSU transfers the Bloom Filter to the newly joined vehicle. Bloom Filter is used in the revocation mechanism. When a vehicle determines an incorrect message, this incorrect message report is sent to certificate authorities (CA). CAs are responsible for transferring public and private key pairs to RSU and vehicles. After receiving the incorrect message, CA inserts the OKC of the illegal node into the Bloom Filter. Then this Bloom Filter is broadcast to all nodes. When a vehicle receives an OWC, it checks the Bloom Filter containing the illegal node OWC. If found, the vehicle ignores all requests from the illegal node.

Jin and Papadimitratos [31] proposed a Bloom Filter-based pseudonym validation technique. Pseudonymous Certification Authority (PCA) inserts all the valid pseudonyms issued to the vehicles in a CBF. Vehicles download the Bloom Filter, i.e., bit values excluding the counters. Then the vehicle performs a query operation to verify the pseudonym of other vehicles. In case Bloom Filter returns $False$, then the vehicle performs a fallback approach to verify the pseudonym. For Bloom Filter updating, the vehicle performs the update when it is not traveling; otherwise, it has received a certain number of pseudonyms but is not recognized by the Bloom Filter. For Bloom Filter update, compressed BF-deltas [32] are used to reduce communication overhead. PCA maintains a Fake Pseudonym List (FPL) to record the pseudonym passed by the Bloom Filter due to a false positive response. This list is broadcast by the PCA to all vehicles. Hence, a vehicle first checks a pseudonym in the Bloom Filter. If Bloom Filter returns $True$, then it is checked in the FPL. If FPL returns $True$, then the pseudonym is legal. Otherwise, it is a false positive response of Bloom Filter, and is ignored.

13.4 Internet-of-Things

Internet-of-Things (IoT) is an interconnection among many analog or digital devices which are homogeneous and heterogeneous in nature, providing an overlapping transmission range for efficient communication of information. IoT also includes both wired and wireless devices. For example, smart homes. The home appliances are connected with a centralized control system in a smart home. The home appliances are controlled by the central device, i.e., switch on, switch off, change the modes, etc. The home owner uses a mobile phone or tablet to pass commands to the central device. Some other applications are IoT in agriculture [33] and healthcare [34]. The rest of the section presents a review of various techniques based on Bloom Filter for improving the performance of IoT devices. Table 13.4 highlights the features and limitations of the IoT techniques.

Airmed [35] is a technique to self-correct the corruption in the application software in the IoT device. Bloom Filter detects the malware program from the benign application code. Every device performs a self-check periodically. If a device finds any corruption, it stops the execution of the application code and tries to download the code from the neighboring devices. The whole application code is partitioned into different chunks. The chunk information is stored in the Bloom Filter. A chunk is hashed by k hash functions. The hashed value gives the bit location in Bloom Filter, which is set to 1. During self-check, the chunks are queried to the Bloom Filter. In case Bloom Filter returns $False$, that chunk is a corrupted or modified code fragment. After determining the corrupted application code fragment, only that code fragment is downloaded from the neighboring devices. If Bloom Filter returns a false positive response for the corrupted code fragment, the device has to download the whole application code.

Hierarchical Bloom Filter-based Indexing (HBFI) [36] is a technique to index the capabilities of IoT devices. An ontology is constructed to represent all the IoT service capabilities. The internal nodes represent the categories of capabilities, and the leaf nodes represent the actual service capabilities. HBFI implements Hierarchical Bloom Filter (HBF) to store the capabilities. A Bloom Filter is constructed for each level of the ontology. All siblings are stored in the same Bloom Filter. Hence, a tree of Bloom Filters is constructed. During capability searching, query operation is performed on the root Bloom Filter. If found, then query operation is performed in the children Bloom Filters of the root. Similarly, the capability is searched.

Deng et al. [37] presented an abnormal traffic detection framework based on Bloom Filter. The technique mainly addresses the port scanning traffic using standard Bloom Filter and TCP flooding using CBF. The framework has three modules: recording, statistics, and detection. The recording module records the packet information. The statistics module inserts information into the Bloom Filters. The detection module determines the traffic state, i.e., normal

TABLE 13.4 Features and Limitations of Internet-of-Things Techniques.

Technique	Features	Limitations
Airmed [35]	• Self corrects the corruption in the application software • Decentralized • Decentralization prevents self-propagating malware • Bloom Filter detects the malware program from the benign application code • Requires low bandwidth for code exchange to repair corrupted devices • Memoryless self-check • Bloom Filter prevents download of whole application code	• False positive response leads to download of whole application code • False positive response leads to a waste of bandwidth • Maintains a single Bloom Filter for each application code • Implements standard Bloom Filter
HBFI [36]	• Unstructured P2P routing • Single Bloom Filter stores all sibling information • Bloom Filter speedups the capability search • Bloom Filter consumes less memory	• Bloom Filter size depends on the number of siblings • Maintains multiple Bloom Filters • Utilizes standard Bloom Filter
Deng et al. [37]	• Retrieves accurate information from real-time data • Utilizes both standard Bloom Filter and CBF • Addresses port scanning and TCP flooding traffic • Implementing Bloom Filter in every module speedups the attack detection process	• Maintains multiple Bloom Filters • Implements standard Bloom Filter which has high FPP • Implements CBF which requires more memory • CBF is used as a counter, but CBF with a large-sized counter requires a lot of memory
BUTI [38]	• Bloom Filter helps in easy classification of tags • Technique rapidly reduces the search space • Avoids tag–tag collision • Tags do require saving the Bloom Filter, which saves memory	• False positive response of the Bloom Filter labels some unknown tag as known • Implements standard Bloom Filter • Requires to frequently flush the Bloom Filter as in IoT network some devices are mobile
CoAP-PBF [39]	• Enhances the CoAP resource discovery • Devices receive compact, complete, and encrypted remote device information • Enhances security • Receives resource information in encrypted format • PBF occupies low memory in the publication message	• Same PBF is periodically broadcast even without any update • A device has to store PBF of all other devices • PBFs consume more memory in a large network having many devices • False positive response from PBF wastes the bandwidth
Dautov et al. [40]	• Device discovery scheme • Constructing tree of Bloom Filter reduces device searching time • Does not require broadcasting of the device search message	• Implements CBF • If a Bloom Filter produces a false positive then it gets passed to the upper-level Bloom Filter • Server has to maintain a big Bloom Filter • Appropriate Bloom Filter size needs to be considered to reduce FPP • Considering the same-size Bloom Filters leads to a waste of memory
Palmieri et al. [41]	• Anonymous routing • Spatial Bloom Filter hides the location and identity of the nodes • Prevents learning of network structure and topology • Encryption of routing information using a homomorphic encryption scheme • Preserves context privacy • Attacker cannot decrypt the file • Maintains privacy by disabling the attacker in learning about the receiver node • Attacker cannot learn about the sender of packet through a switch	• FPP results in incorrect delivery to a subnetwork • Broadcasting of the packet is a waste of resources • Gateway does not determine the receiver node • If attacker controls both the source node and routing layers then it can determine the receiver node

or abnormal. The framework maintains different statistics and detection modules for port scanning and TCP flooding. The statistics of port scanning include the total number of packets, the number of traffic usage protocols, and the number of ports accessed. The recording module detects the stream by using Bloom Filter. The module retrieves the source and the destination address after receiving the packet. The addresses are queried into the Bloom Filter. If Bloom Filter returns $True$, then the packet information is recorded in its corresponding record of that stream. If Bloom Filter returns $False$, the packet addresses are inserted into the Bloom Filter and forwarded to the statistics

module. The statistics module also maintains a Bloom Filter that uses source address, a destination address, and destination port number as identifiers to differentiate among the packets. When the packet arrives in this module, the identifier is queried to Bloom Filter. If Bloom Filter returns $True$, the packet is forwarded to the detection module. If Bloom Filter returns $False$, then its identifier is inserted into the Bloom Filter. The detection module maintains an array of the common port numbers. After receiving the packet, the detection module stores the number of ports required by the packet. If the number of ports is high, then it is concluded that traffic is abnormal due to port scanning attacks. The recording module for detecting TCP flooding is similar to the recording module for port scanning. The statistics module maintains two CBFs. The first CBF (CBF_1) keeps the count of the streams based on the destination address and port as an identifier. The second CBF (CBF_2) keeps the count of the streams based on the destination address, destination port, and packet length as an identifier. When a packet reaches the statistics module, the destination address and port are extracted and queried to CBF_1. If present in CBF_1, the counter is incremented. The packets are classified based on the packet length, i.e., zero or multiple of 10. The packets having a length as multiples of 10 are considered similar. The destination address, destination port, and packet length are queried to CBF_2. If present in CBF_2, the counter is incremented. The detection module extracts the two counts of the current stream from CBF_1 and CBF_2 based on the respective identifiers. Then determines the ratio of the counts. In case the ratio is small, the traffic is normal. If the ratio exceeds a certain threshold value, then the stream is a TCP flooding attack. Furthermore, if the stream has zero packet length, it is an SYN flooding traffic; otherwise, TCP flooding traffic.

Bloom Filter-based Unknown Tag Identification (BUTI) [38] is a Bloom Filter-based RFID tag identification of IoT devices. The devices in an IoT network are identified uniquely by using a radio frequency identification (RFID) tag. BUTI has two phases, known tag filtering and unknown tag collection. The reader constructs a Bloom Filter in the known tag filtering and inserts all the known tags into the Bloom Filter. The reader then broadcasts the Bloom Filter to all tags. When a tag receives the Bloom Filter, it queries its tag. If the Bloom Filter returns $True$, then the tag is labeled as known. Otherwise, the tag labels itself as unknown. The tags do not save the Bloom Filter. The unknown tag collection phase collects the unknown tags. The reader broadcasts the frame length and hash seed to tags. After receiving a message from the reader, the tag sends a short response. The reader constructs an indicator vector where each slot corresponds to a tag. Initially, the slots of the indicator vector are set to 0. If the reader receives a response from a tag, then the corresponding slot in the indicator vector is set to 1. The reader broadcasts the indicator vector. After receiving the indicator vector, the tag checks its slot. If it is 1, then the tag sends its ID to the reader; otherwise, ignores it. The technique can remove the labeling phase. If a tag determines itself as an unknown tag from the received Bloom Filter, then it can send its tag ID. This step completely removes the need for the indicator vector.

Constrained Application Protocol-Partitioned Bloom Filter (CoAP-PBF) [39] is a discovery scheme to enhance the CoAP resource discovery by IoT devices. PBF is a CBF variant where each cell is of 4 bits. Initially, every device constructs a service database and a PBF. A device inserts its resources into PBF, excluding the resource of request type. The deviceID and resourceID are concatenated and hashed by a hash function. The hashed value is partitioned into k number of partitions. The partition provides the counter location in PBF, which is incremented. After the construction of the PBF, it is attached to the publication message. The publication message is broadcast in the network to inform other devices about its liveness. All devices extract the PBF from the publication message. When a device requires a resource, it queries the PBF about the resources. If a PBF returns $True$, it means the required resource is present in PBF owner's device. Then the device sends a CoAP GET query to request the resource. When a resource of a device fails, the resource information is removed from its PBF. The delete operation of PBF is similar to the insertion operation, except the counter is decremented. The updated PBF is broadcast in the subsequent publication message transmission. When a PBF returns a false positive response, the device receives an error message against the CoAP GET query. Again the device has to query the PBF of other devices. Hence, a false positive response leads to a waste of bandwidth.

Dautov et al. [40] presented a property-based IoT device discovery scheme. The devices within a network are represented as a tree. The network is classified based on the subnetwork, subnetwork is classified based on subnetwork gateway, and finally, the subnetwork gateway has devices as children. The root is the server of the network. A Bloom Filter is constructed for each device. The device properties such as type, location, functionality are inserted into the Bloom Filter. Then all the sibling Bloom Filters are merged into a single Bloom Filter, which is maintained by the subnetwork gateway. Similarly, all the sibling Bloom Filters of the subnetwork gateways are merged to a single Bloom Filter for the subnetwork. Finally, all Bloom Filters of the network are merged, which is maintained by the server. During a device search, the server checks its Bloom Filter. If the Bloom Filter returns true, the device is searched in its children Bloom Filters, i.e., subnetwork Bloom Filters. If any Bloom Filter returns $True$, the search is performed in its children. The search is continued till the device is found or all intermediate nodes return $False$. When an inter-

mediate node returns *False*, all or some Bloom Filters have the false positive issue. The scheme does not elaborate on the actions that should be taken due to the deletion of a device. Moreover, the scheme does not discuss the size of the Bloom Filter. The size of the Bloom Filter should increase with the increase of the tree level to maintain a low FPP. Maintaining the same sized Bloom Filter leads to high FPP in levels if the size is small, on the other hand, using a large size is a waste of memory in lower levels.

Palmieri et al. [41] proposed a protocol for anonymous routing among the IoT nodes. The protocol uses Spatial Bloom Filter (SBF) [42]. The message transmitted in the anonymous routing protocol consists of two parts, header and payload. The header has the routing information, and a payload is encrypted. The payload is a packet from lower layers such as TCP and UDP. Every node in the network has a public/private key pair. A key distribution scheme is followed among the nodes to retrieve the public key from a known node. The header does not mention the sender and receiver IDs. All the sender and receiver node IDs are kept anonymous by sending homomorphically encrypted SBF. The routing layer knows the secret key, whereas the encrypted SBF and public key are transmitted to all nodes. The different subnetworks are the different sets in SBF, whereas the node IDs are the items in SBF. The k hash functions required for performing an operation on SBF are known by all nodes. When the sender and receiver are in different subnetworks, the sender node constructs SBF. It inserts all the node IDs in the network and subnetwork. Then SBF is encrypted using Paillier cryptosystem [43]. The encrypted SBF is included in the payload of the packet and transmitted. When the router receives the packet, it decrypts the filter to obtain SBF. The router queries the SBF to determine the receiver node. If SBF returns *True*, it is forwarded to the receiver node's subnetwork. When the gateway receives the packet, it broadcasts it to all the subnetwork nodes. When the receiver node receives the packet, it decrypts the filter using the secret key.

13.5 Discussion

Table 13.5 highlights a comparison of various techniques reviewed in this chapter based on some parameters. One major issue in wireless communication, i.e., WSN, MANET, VANET, and IoT, is that the devices or nodes are mobile. Hence, locating a node is difficult because the location of a node is changing. In a subnetwork, the nodes frequently leave or join the network. Thus, every node needs to remain updated about the network's topology. Furthermore, many messages need to be broadcast in the network, such as key sharing for authentication, provenance information, etc. Therefore, apart from information sharing, all nodes remain engaged in other algorithms of keeping themselves updated about the network topology. Bloom Filter is a great help in this issue. A bulky message consumes the bandwidth and memory of the receiver. However, a small-sized Bloom Filter helps in storing lots of information while utilizing low bandwidth for transmission. Moreover, broadcasting Bloom Filter in some periods is a cost-effective solution. Furthermore, the operation on Bloom Filter takes very little time; hence, inserting or retrieving information from Bloom Filter is very fast. Another issue of sensors in wireless communication is the limited power supply. Due to the power limitation, the sensors are unable to provide complex processing. The operation of Bloom Filter has constant time complexity; hence, the insertion or query of information into or from the Bloom Filter is fast. Therefore, the sensors or nodes require lower power to perform operations in Bloom Filter.

Similar to other applications, the wireless communication techniques also implement standard Bloom Filter. A standard Bloom Filter gives low FPP if its size is big or the number of hash functions is more than in other variants of Bloom Filter. The standard Bloom Filter is fast; however, more efficient Bloom Filter variants are available that give low FPP with a small size. A faster Bloom Filter is important because Bloom Filter mostly helps reduce the cost of broadcasting messages for various purposes. With limited hardware resources, energy, and computational resources, more emphasis should be given to implementing the fastest Bloom Filter to optimize the algorithms further.

13.6 Conclusion

This chapter discussed the solutions Bloom Filter provides for enhancing the performance of wireless communication. The chapter on wireless communication includes wireless sensor networks, Mobile Ad-hoc Network, Vehicular Ad-hoc Network, and the Internet-of-Things. The chapter also presented a review of many techniques based on the Bloom Filter proposed for wireless communication. All wireless communications have to dedicate a major part of the processing for periodically broadcasting various information to every node within the network. The dynamic

TABLE 13.5 Comparison of Wireless Communication Techniques: Variant, Bloom Filter variant; Reduce FP, technique used to reduce the number of false positives; Purpose, purpose of the technique.

Technique	Variant	Reduce FP	Purpose
Choi et al. [13]	Standard	No	Device Discovery Protocol
Sanchez-Hernandez et al. [14]	Counting	No	Broadcasting of topological information
Talpur et al. [15]	Standard	No	Broadcasting protocol
Moualla et al. [16]	Standard	No	Cache Management
OSR [19]	Standard	No	Routing
BFRP [1]	Dynamic	Adds new standard Bloom Filter	Routing
BFRE [17]	Standard	No	Redundancy Elimination
CSPR [18]	Standard	No	Path Information
BFAN [24]	Standard	XOR operation	Authentication
Kumar and Prasad [25]	Standard	No	Provenance verification
Thesnim and Nithin [26]	Standard	No	Provenance verification
Bhatti and Saleem [27]	Standard	No	Secret key generation
Oigawa and Sato [29]	Standard	No	Zone routing
Bao et al. [30]	Standard	No	Recognition of illegal nodes
Jin and Papadimitratos [31]	CBF and standard	Maintains a Fake Pseudonym List	Pseudonym Validation
Airmed [35]	Standard	No	Self-correction of the application code
HBFI [36]	Hierarchical	No	Service Discovery
Deng et al. [37]	Standard, Counting	No	Port scanning and TCP flooding attack Detection
BUTI [38]	Standard	No	RFID tag identification
CoAP-PBF [39]	Partitioned	No	Resource discovery
Dautov et al. [40]	Counting	No	Device discovery
Palmieri et al. [41]	Spatial	No	Security, Privacy

nature of the network means that the nodes and devices within a network are mobile. The nodes leave or join a network very frequently. Thus, various information is required for communication, such as authentication information, a secret key for encryption or decryption of the message, etc. This information needs to be broadcast periodically, but a bulky message consumes more bandwidth, and its processing requires more power. These issues are solved by the Bloom Filter, which, with its small size, stores lots of information. Therefore, a lot of information is broadcast in a single packet. Furthermore, Bloom Filter, with its low time complexity, requires low power for performing operations.

References

[1] S.M.S. Amiri, H.T. Malazi, M. Ahmadi, Memory efficient routing using Bloom filters in large scale sensor networks, Wirel. Pers. Commun. 86 (3) (2016) 1221–1240.

[2] M. Keerthika, D. Shanmugapriya, Wireless sensor networks: active and passive attacks – vulnerabilities and countermeasures, in: International Conference on Computing System and Its Applications (ICCSA-2021), Global Transit. Proc. 2 (2) (2021) 362–367, https://doi.org/10.1016/j.gltp.2021.08.045.

[3] K. Bouabdellah, H. Noureddine, S. Larbi, Using wireless sensor networks for reliable forest fires detection, in: The 4th International Conference on Ambient Systems, Networks and Technologies (ANT 2013), The 3rd International Conference on Sustainable Energy Information Technology (SEIT-2013), Proc. Comput. Sci. 19 (2013) 794–801, https://doi.org/10.1016/j.procs.2013.06.104.

[4] T. Ezzedine, A. Zrelli, Efficient measurement of temperature, humidity and strain variation by modeling reflection Bragg grating spectrum in WSN, Optik 135 (2017) 454–462, https://doi.org/10.1016/j.ijleo.2017.01.061.

[5] L.E. Jim, N. Islam, M.A. Gregory, Enhanced MANET security using artificial immune system based danger theory to detect selfish nodes, Comput. Secur. 113 (2022) 102538, https://doi.org/10.1016/j.cose.2021.102538.

[6] S. Kodam, N. Bharathgoud, B. Ramachandran, A review on smart wearable devices for soldier safety during battlefield using WSN technology, in: International Conference on Nanotechnology: Ideas, Innovation and Industries, Mater. Today Proc. 33 (2020) 4578–4585, https://doi.org/10.1016/j.matpr.2020.08.191.

[7] H. Wu, J. Wang, R.R. Ananta, V.R. Kommareddy, R. Wang, P. Mohapatra, Prediction based opportunistic routing for maritime search and rescue wireless sensor network, J. Parallel Distrib. Comput. 111 (2018) 56–64, https://doi.org/10.1016/j.jpdc.2017.06.021.

[8] B. Samra, S. Fouzi, New efficient certificateless scheme-based conditional privacy preservation authentication for applications in VANET, Veh. Commun. (2021) 100414, https://doi.org/10.1016/j.vehcom.2021.100414.

[9] M.A. Shahid, A. Jaekel, C. Ezeife, Q. Al-Ajmi, I. Saini, Review of potential security attacks in VANET, in: 2018 Majan International Conference (MIC), 2018, pp. 1–4.
[10] M. Nasir, K. Muhammad, A. Ullah, J. Ahmad, S. Wook Baik, M. Sajjad, Enabling automation and edge intelligence over resource constraint IoT devices for smart home, Neurocomputing 491 (2022) 494–506, https://doi.org/10.1016/j.neucom.2021.04.138.
[11] K.N. Qureshi, A. Naveed, Y. Kashif, G. Jeon, Internet-of-Things for education: a smart and secure system for schools monitoring and alerting, Comput. Electr. Eng. 93 (2021) 107275, https://doi.org/10.1016/j.compeleceng.2021.107275.
[12] S.P. Kamble, N. Thakare, S. Patil, A review on cluster-based energy efficient routing with hybrid protocol in wireless sensor network, in: 2014 International Conference on Computer Communication and Informatics, 2014, pp. 1–4.
[13] K.W. Choi, D.T. Wiriaatmadja, E. Hossain, Discovering mobile applications in cellular device-to-device communications: hash function and Bloom filter-based approach, IEEE Trans. Mob. Comput. 15 (2) (2016) 336–349, https://doi.org/10.1109/TMC.2015.2418767.
[14] J.J. Sanchez-Hernandez, R. Menchaca-Mendez, R. Menchaca-Mendez, J. Garcia-Diaz, M.E. Rivero-Angeles, J.J. Garcia-Luna-Aceves, A Bloom filter-based algorithm for routing in intermittently connected mobile networks, in: Proceedings of the 18th ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, MSWiM'15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 319–326.
[15] A. Talpur, T. Newe, F.K. Shaikh, A.A. Sheikh, E. Felemban, A. Khelil, Bloom filter based data collection algorithm for wireless sensor networks, in: 2017 International Conference on Information Networking (ICOIN), 2017, pp. 354–359.
[16] G. Moualla, P.A. Frangoudis, Y. Hadjadj-Aoul, S. Ait-Chellouche, A Bloom-filter-based socially aware scheme for content replication in mobile ad hoc networks, in: 2016 13th IEEE Annual Consumer Communications Networking Conference (CCNC), 2016, pp. 359–365.
[17] G. Park, Y. Shim, I. Jang, S. Pack, Bloom-filter-aided redundancy elimination in opportunistic communications, IEEE Wirel. Commun. 23 (1) (2016) 112–119, https://doi.org/10.1109/MWC.2016.7422413.
[18] Z. Liu, Z. Li, M. Li, W. Xing, D. Lu, Path reconstruction in dynamic wireless sensor networks using compressive sensing, IEEE/ACM Trans. Netw. 24 (4) (2016) 1948–1960, https://doi.org/10.1109/TNET.2015.2435805.
[19] X. Zhong, Y. Liang, Scalable downward routing for wireless sensor networks actuation, IEEE Sens. J. 19 (20) (2019) 9552–9560, https://doi.org/10.1109/JSEN.2019.2924153.
[20] B. Kannhavong, H. Nakayama, Y. Nemoto, N. Kato, A. Jamalipour, A survey of routing attacks in mobile ad hoc networks, IEEE Wirel. Commun. 14 (5) (2007) 85–91, https://doi.org/10.1109/MWC.2007.4396947.
[21] S. Lakshmanan, C.-L. Tsao, R. Sivakumar, K. Sundaresan, Securing wireless data networks against eavesdropping using smart antennas, in: 2008 the 28th International Conference on Distributed Computing Systems, 2008, pp. 19–27.
[22] U. Meyer, S. Wetzel, A man-in-the-middle attack on UMTS, in: Proceedings of the 3rd ACM Workshop on Wireless Security, WiSe'04, Association for Computing Machinery, New York, NY, USA, 2004, pp. 90–97.
[23] T. Ohigashi, M. Morii, A practical message falsification attack on WPA, Proc. JWIS 54 (2009) 66.
[24] B. Mbarek, N. Sahli, N. Jabeur, Bfan: a Bloom filter-based authentication in wireless sensor networks, in: 2018 14th International Wireless Communications Mobile Computing Conference (IWCMC), 2018, pp. 304–309.
[25] S. Pavan Kumar, K.S. Nandini Prasad, Novel approach for detecting malicious activity in WSN using securing scheme, in: 2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT), 2016, pp. 76–80.
[26] H. Thesnim, M.V. Nithin, Trust evaluation in detection of provenance forgery and packet drop attacks in WSNs, in: 2017 International Conference on Intelligent Computing and Control (I2C2), 2017, pp. 1–4.
[27] D.S. Bhatti, S. Saleem, Ephemeral secrets: multi-party secret key acquisition for secure IEEE 802.11 mobile ad hoc communication, IEEE Access 8 (2020) 24242–24257, https://doi.org/10.1109/ACCESS.2020.2970147.
[28] T.H. Hadi, MANET and WSN: what makes them different?, Int. J. Comput. Netw. Wirel. Commun. 7 (6) (2017) 23–26.
[29] Y. Oigawa, F. Sato, An improvement in zone routing protocol using Bloom filter, in: 2016 19th International Conference on Network-Based Information Systems (NBiS), 2016, pp. 107–113.
[30] S. Bao, W. Hathal, H. Cruickshank, Z. Sun, P. Asuquo, A. Lei, A lightweight authentication and privacy-preserving scheme for VANETs using TESLA and Bloom filters, ICT Express 4 (4) (2018) 221–227, https://doi.org/10.1016/j.icte.2017.12.001.
[31] H. Jin, P. Papadimitratos, Proactive certificate validation for VANETs, in: 2016 IEEE Vehicular Networking Conference (VNC), 2016, pp. 1–4.
[32] M. Mitzenmacher, Compressed Bloom filters, IEEE/ACM Trans. Netw. 10 (5) (2002) 604–612.
[33] J. Xu, B. Gu, G. Tian, Review of agricultural IoT technology, Artif. Intell. Agric. 6 (2022) 10–22, https://doi.org/10.1016/j.aiia.2022.01.001.
[34] N.S. Sworna, A.M. Islam, S. Shatabda, S. Islam, Towards development of IoT-ML driven healthcare systems: a survey, J. Netw. Comput. Appl. 196 (2021) 103244, https://doi.org/10.1016/j.jnca.2021.103244.
[35] S. Das, S. Wedaj, K. Paul, U. Bellur, V.J. Ribeiro, Airmed: efficient self-healing network of low-end devices, arXiv:2004.12442, 2020.
[36] H. Moeini, I.-L. Yen, F. Bastani, Efficient caching for peer-to-peer service discovery in Internet-of-Things, in: 2017 IEEE International Conference on Web Services (ICWS), 2017, pp. 196–203.
[37] F. Deng, Y. Song, A. Hu, M. Fan, Y. Jiang, Abnormal traffic detection of IoT terminals based on Bloom filter, in: Proceedings of the ACM Turing Celebration Conference – China, ACM TURC'19, Association for Computing Machinery, New York, NY, USA, 2019.
[38] D. Zhang, Z. He, Y. Qian, J. Wan, D. Li, S. Zhao, Revisiting unknown RFID tag identification in large-scale Internet-of-Things, IEEE Wirel. Commun. 23 (5) (2016) 24–29, https://doi.org/10.1109/MWC.2016.7721738.
[39] M.R. Khaefi, D.-S. Kim, Bloom filter based CoAP discovery protocols for distributed resource constrained networks, in: 2015 IEEE 13th International Conference on Industrial Informatics (INDIN), 2015, pp. 448–453.
[40] R. Dautov, S. Distefano, O. Senko, O. Surnin, Property-based network discovery of IoT nodes using Bloom filters, in: Q. Zu, B. Hu (Eds.), Human Centered Computing, Springer International Publishing, Cham, 2018, pp. 394–399.
[41] P. Palmieri, L. Calderoni, D. Maio, Private inter-network routing for wireless sensor networks and the Internet-of-Things, in: Proceedings of the Computing Frontiers Conference, CF'17, Association for Computing Machinery, New York, NY, USA, 2017, pp. 396–401.
[42] L. Calderoni, P. Palmieri, D. Maio, Location privacy without mutual trust: the spatial Bloom filter, in: Security and Privacy in Unified Communications: Challenges and Solutions, Comput. Commun. 68 (2015) 4–16, https://doi.org/10.1016/j.comcom.2015.06.011.
[43] P. Paillier, Public-key cryptosystems based on composite degree residuosity classes, in: International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 1999, pp. 223–238.

This page intentionally left blank

CHAPTER

14

Network security using Bloom Filter

14.1 Introduction

Designing a technique to provide security in a network is a challenging task. Regular network traffic has millions of packet transmissions at any instance. During an attack, requests come in millions and increase exponentially. Hence, the primary requirement of a security technique is high efficiency with faster processing of packets. Moreover, efficiency needs to be maintained during an attack. Another issue with the security technique is that it is an overhead to the main application of the network. This provides another requirement for the security technique, i.e., good performance while maintaining constraints such as fewer resources, reasonable bandwidth, faster response, etc.

Bloom Filter is an excellent choice for network security because it can satisfy the main requirements of a security technique. It is capable of filtering packets quickly. Hence, it is implemented to defend the applications against many attacks. For example, Bloom Filter can deduplicate the packets rapidly to protect against DoS and DDoS attacks. Bloom Filter has a tiny memory footprint; hence, it is kept in the main memory. This further increases the performance of the Bloom Filter and satisfies the second requirement of the security technique. Another critical point is that the Bloom Filter is platform-independent. Hence, it can be implemented in various applications with many diverse features.

This chapter explores the implementation of Bloom Filter to provide security, maintain privacy, and defend against many attacks. Furthermore, in the discussion section, we elaborate how Bloom Filter architecture and operations are sufficient to shield the application from various attacks. In addition, this chapter also review many security techniques based on Bloom Filter. The chapter is organized as follows: Section 14.2 elaborates on distributed denial of services (DDoS) and reviews some techniques to defend against it. Section 14.3 briefly explains the denial of services (DoS). It also includes reviews on various techniques based on Bloom Filter to safeguard against DoS. Section 14.4 discusses some attacks specifically targeting Bloom Filter and how Bloom Filter is immune against these attacks. Section 14.5 mentions various techniques based on Bloom Filter to improve the security of the network. Section 14.6 explains how Bloom Filter helps in protecting sensitive data against intruders. It also includes some reviews of such techniques. Section 14.7 illustrates a small experiment conducted to prove that Bloom Filter acts as a roadblock for the adversary's advancement. Section 14.8 discusses the measures taken by Bloom Filter to tackle the false positive and false negative issues. Finally, Section 14.9 concludes the chapter.

14.2 DDoS

In a Distributed Denial-of-Service (DDoS) attack, the attacker uses many virtual machines having different IP addresses to generate millions of requests per second to flood the victim node/system. Sometimes DDoS traffic is in the range of hundreds of gigabits [1]. The highest traffic during the DDoS attack in 2016 was 1 terabit per second [2]. The attack tries to keep critical resources (e.g., Internet link capacity, stack space of the protocol software, and server CPU capacity) engaged in the network services. Usually, attackers use Zombies or Botnet computers which are widely scattered, well organized, and remotely controlled. They send service requests concurrently and continuously. To prevent detection, they use spoofed IP addresses.

Two methods are used to initiate DDoS attacks:

1. **Vulnerability Attack.** It sends a few malformed packets to victim nodes.
2. **Flooding attack:** A flooding attack is a distributed denial-of-service attack that aims to make a server unavailable by flooding it with a set of requests to all available server's resources.

 https://doi.org/10.1016/B978-0-12-823520-1.00020-1

Copyright © 2023 Elsevier Inc. All rights reserved.

a. Network/Transport-level. The attack exhausts bandwidth, router processing capacity or network resources to disturb the connectivity of legitimate users.
b. Application-level. The attack exhausts the server resources such as sockets, disk/database bandwidth, CPU, sockets, I/O bandwidth, memory, etc., to disturb the connectivity of legitimate users.

Some features of DDoS that make the design of defense algorithms difficult are:

- **Huge traffic.** It is difficult and challenging for a node to handle huge traffic in a short span.
- **Attack from different geographical locations.** Usage of spoofed IP addresses makes the source of the attack appear as they are generated from different geographical locations. Hence, it hinders the design of an efficient traceback mechanism.
- **Multiple attack sources.** The attack comes from multiple sources where each source generates less traffic to appear as a legitimate user. It makes the design of IP address filtering inefficient.

14.2.1 Types of DDoS attacks

Some famous DDoS attacks are UDP Flood, TCP SYN Flood, SQL Slammer Worm attack and NTP attack, and DNS Amplification.

- **UDP Flood.** Huge number of packets are sent to the victim node. Examples of such an attack tools are Stacheldraht, TFN2K, and Shaft.
- **TCP SYN Flood.** The attacker uses non-existing and not used IP addresses to send SYN packets. Following the three-way handshake protocol, the server stores the request details in memory and waits for client confirmation. Without client confirmation, the memory remains occupied. A huge number of similar requests saturate the memory and prevent the server from serving legitimate users. Examples of such an attack tools are Stacheldraht, Shaft, TFN, TFN2K, and Trinity.
- **SQL Slammer Worm.** This sends a massive number of requests to random nodes [3]. The nodes sharing the same MS SQL vulnerability get infected and further infect other nodes. One example of such an attack tool is Trinoo.
- **DNS Amplification Attack.** In this attack, the attacker hacks the Domain Name System (DNS) to construct a big-size resource record (RR). From the victim node, the attacker sends a request for the big RR. Then, DNS uses huge bandwidth to serve the request and remains unavailable for legitimate users.
- **NTP Attack.** In this attack, the attacker saturates the bandwidth of the victim amplifier [4]. An amplifier is a node executing a protocol such as Network Time Protocol (NTP) that sends one or more larger response packets than the query itself. Before the attack, the attackers scan the network to determine the vulnerable amplifiers.

14.2.2 Defense techniques against DDoS

Saurabh and Sairam [5] proposed wrap-around counting Bloom Filter (WCBF) to improve the performance of packet marking technique to handle DoS/DDoS attacks. WCBF helps to reduce the number of packets required for IP traceback. WCBF is implemented in the edge router close to the packet source. The slot value of WCBF decides the mark that is inserted into the packet. WCBF stores the IP ID. When a packet reaches the edge router, WCBF is searched using IP ID. If absent, it is inserted into WCBF, and the counter is incremented by performing the modulo operation. Every router checks the matching of the number of hops the packet covered and its IP ID. If both match, then that router is eligible to mark the packet.

Kavisankar et al. [6] proposed a Bloom Filter-based Efficient SYN spoofing Detection and Mitigation Scheme (ESDMS) framework. The framework consists of Efficient SYN spoofing Detection Scheme (ESDS) and Efficient SYN spoofing Mitigation Scheme (ESMS). SYN spoofing is determined based on the comparison between trusted and a threshold value. Trusted value is calculated using TCP probing, whereas threshold value is computed using statistical testing. The trust value of each requested IP address is stored in the confidence table. The Bloom Filter stores trusted IP addresses. When a client sends an SYN request, ESMS checks in Bloom Filter and determines IP address availability using TCP probing. When both return $True$, ESMS responds with SYN with acknowledgment to the client and establishes the connection.

SkyShield [7] is a defense system for application-layer DDoS attacks. It consists of two stages, namely, mitigation and detection. The mitigation phase uses Bloom Filter, and the detection phase uses sketch. The mitigation phase uses two Bloom Filters, called whitelist and blacklist, for incoming requests filtering. Whitelist stores legitimate

nodes, and blacklist stores malicious nodes. Classification of nodes (i.e., legitimate or malicious) is determined using captcha techniques. When a request is received, the whitelist and blacklist check and determine the type of node. If the node is legitimate, then the request is forwarded to the detection phase; otherwise, the node is logged. If the node is absent in both Bloom Filters, then the node is directed to the captcha test. If the node passes the test, then it is stored in the whitelist; otherwise, relegated to the blacklist. Bloom Filter helps to classify the request without reverse calculation of their IP addresses. Periodically, both blacklist and whitelist are flushed to avoid blocking a user infinitely. Bloom Filter helps in filtering the incoming request from legitimate users without reverse calculation of their IP addresses. However, false positive responses allow access to malicious nodes.

Dodig et al. [8] implemented Dual Counting Bloom Filter (DCBF) to reduce incorrect detection of matching packets. The technique is used in the $SACK^2$ algorithm. In the technique, three counting Bloom Filters (CBF1, $\overline{CBF1}$ and CBF2) are used. CBF1 identifies ACK packets for each SYN/ACK packet. CBF1 obtains K locations, and their respective counter values are decreased. When a counter value reaches 0, it means ACK does not match the SYN/ACK packet, and further processing is dropped. If the first CBF1 gives a $True$ response (ACK-SYN/ACK packet matches), then $\overline{CBF1}$ repeats the same procedure to reduce FPP. After decreasing the counter values, if it reaches 0, it is correctly decided that the ACK-SYN/ACK packet does not match. CBF1 produces the destination IP address and the destination port number as output. CBF2 is used in detecting the presence of the attack. CBF2 takes the output of CBF1, similar to the input method.

Tseung and Chow [9] proposed a device to live capture the network data efficiently. Also, an anti-DDoS technique is integrated to capture forensic data during DDoS attacks. The anti-DDoS technique is based on machine learning (ML) and Bloom Filter. ML monitors traffic to determine whether the victim node is under attack. Upon detection, ML extracts attack features. Based on some selected features, Bloom Filters are constructed. The Bloom Filter is a multi-layer CBF. Each Bloom Filter stores packets with similar features. When Bloom Filter receives a packet, it hashes the packet to obtain the location of the slot and increment its counter. When the counter crosses a threshold, it concludes that the packet is malicious. The overflow issue of CBF is addressed by using a leaky bucket algorithm. The algorithm decrements the counter to permit the passage of legitimate packets.

14.2.2.1 Link flooding attack

The attacker achieves a link flooding attack by indirectly attacking the target server using other servers. The attacker searches for connectivity links between the target area and the Internet. These links are usually servers around the target server. Then the attacker sends a smaller number of legitimate requests to all surrounding servers. These servers forward the request to the target server. Eventually, the target server receives heavy traffic and slows down [10]. Attackers use coordinated bots to generate legitimate requests. To achieve this attack, the attacker has to control two conditions. The first is sending requests which are considered legitimate by the server, and the other is obstructing rerouting [11]. Once the malicious requests are detected, they are dropped. When the requests are rerouted, the traffic reduces. In this attack, the attacker sends legitimate requests, making the design of the defense technique difficult.

Xiao et al. [12] proposed a Bloom Filter defense technique against link flooding attacks in Software-Defined Networking (SDN). The technique consists of two modules, namely, collector and detector. Both modules can reside in the same or different nodes. The collector module scans SDN flow tables for IP header inspection of traffic flow. If the link utilization ratio is suspicious, the flow table is scanned to evaluate the statistical features to determine the abnormal flow. After detection of abnormal flows, the IP address and port number are stored in Bloom Filter. Bloom Filter reduces the storage overhead of storing flow information. The detector module has a sniffer to receive packets in real-time. After receiving packets, the IP features required to determine the link flooding attack are extracted. The packet information is checked in the Bloom Filter. If it is present, the flow is malicious, and the result is sent to the controller.

14.2.2.2 Interest flooding attack

In the Content-Centric Networking (CCN), the Interest packets are stored in the PIT and removed after forwarding. The DDoS attack is difficult in CCN because same name Interest packets are aggregated and stored as a single Interest [13]. However, in an Interest flooding attack, the attacker sends a massive number of Interest packets using different names. PIT stores all Interest packets, and the malicious packets occupy the entire memory of PIT and halt the forwarding of legitimate packets. For an Interest flooding attack, the attacker prepares content namespace by assigning value to all the sections of an Interest packet, such as content name, content version, and segment number.

For a DoS attack, the attacker increments the segment number. Moreover, the attacker increases the Interest packet size by increasing the content name for faster consumption of PIT memory.

Bloom filter-based Attack Mitigating (BLAM) [14] is a technique to detect attacks in IoT nodes accurately. Every IoT node maintains a Bloom Filter locally. Bloom Filter is deployed in between CS and PIT. When the IoT node receives an Interest packet, it is searched in CS. If present, its respective Data packet is forwarded to the sender; otherwise, checked in its local Bloom Filter. If present, then the packet is forwarded to PIT. Otherwise, the packet is dropped by concluding it as malicious. The IoT nodes send Bloom Filter inside the Ba-NACK packet for synchronization. When such a packet is received, the node copies the Bloom Filter, updates its own local Bloom Filter array, and then forwards the packet to the destination node. In the article, the technique has not provided a clear procedure for insertion of packet into the Bloom Filter. One issue of this technique is that Bloom Filter is not flushed, which leads to saturation and an increase in FPP.

14.2.2.3 Data flooding attack in ICN

In this attack, the attacker floods the nodes with a massive number of data packets. For this attack, the attacker requires the list of Path Identifiers (PID). PID is used to identify paths when the data packet takes the reverse path to reach the sender. Therefore, PID is not made public to prevent the attack.

LogDos [15] is a PID-based ICN mechanism to prevent data flooding attacks using Bloom Filter. It filters packets based on GET messages. It adds extra information to the GET message and sends it to the router to check the data packet. When an Interest packet reaches a node, that node adds its PID to the GET message, and the same is performed by every router between the consumer and producer node. Consumer node is the node that requests content or consumer of data. The producer node is the node that provides the content or produces the content. When the data packet returns, it uses the PID list in reverse order. In the reverse path, the routers check whether this packet has taken this path. This checking is called logging in LogDos. The memory consumption is reduced by using Bloom Filter to insert the extra information to the GET message. The SID and the list of PIDs uniquely identify the GET message; hence these are inserted into the Bloom Filter. The Service Identifier (SID) uniquely identifies the content in the ICN network. When the Bloom Filter becomes full, the FPP becomes 1, hence the router maintains two Bloom Filters. When the first become saturated and gives more false positives than permitted, the second Bloom Filter is used. During a query operation, both Bloom Filters are checked. When the second Bloom Filter becomes saturated, the first Bloom Filter is flushed, and new inputs are inserted. The Odd/Even Logging scheme is used to reduce the computational overhead of the router. In case the number of PIDs in a GET message is odd, only the routers at odd positions in the PID list perform the logging. Similarly, if the number in PID in a GET message is even, then Even Logging is performed. Only the routers at even positions in the PID list perform the logging. LogDos also performs dynamic logging, in which the logging process can be switched off in case there is no attack. A time period is fixed to switch-off logging. In case the number of invalid data packets exceeds a threshold value, the router determines an attack is happening. In such scenarios, the logging switch-off time is extended. See Table 14.1.

14.3 DoS

DoS is different from DDoS. DoS is the denial of service where a single system attacks the target node by generating many requests or packets and forwarding them to the target node. DoS is slower than DDoS because DoS has a single system to create packets, whereas DDoS has many systems to generate a huge number of packets simultaneously. Hence, the traffic generated by DoS is less compared to DDoS. It is easy to stop the DoS attack as it is a single system, whereas in DDoS, all systems need to be identified to stop the attack, and in many cases, the systems are scattered throughout the globe. The DoS/DDoS techniques require filtering requested operations received from the malicious nodes while consuming less time. Hence, majority techniques use Bloom Filter, for instance, those of Saurabh and Sairam [5] and SkyShield [7]. SkyShield [7] uses two Bloom Filters, namely, blacklist and whitelist. These Bloom Filters help to distinguish between legitimate and malicious requests. Blacklist is responsible for saving malicious nodes, whereas whitelist is responsible for saving legitimate nodes. Saurabh and Sairam [5] technique is based on packet marking. The wraparound counting Bloom Filter (WCBF) is used for packet marking to enhance the packet marking process.

The Bloom Filter has a small array size; henceforth, Bloom Filter is stored in RAM for faster processing. Compared to normal network traffic, the frequency of incoming packets increases manyfold during a DoS/DDoS attack. The Bloom Filter efficiency can also be enhanced by maintaining multiple Bloom Filters to perform parallel processing.

TABLE 14.1 Features and limitations of defense techniques against DDoS.

Technique	Features	Limitations
Saurabh and Sairam [5], 2016	• Reduces the number of packets for IP traceback • Usage of cyclic counter prevents counter overflow	• Increase in packet loss leads to increase in requirement of packets for IP traceback • Increase in collision leads to degradation in performance
Xiao et al. [12], 2016	• Works in real time • Bloom Filter helps in scanning flows in real time	• Saving flow statistics is overhead • Conventional Bloom Filter is used
Kavisankar et al. [6], 2017	• TCP probing enhances bandwidth utilization • Bloom Filter is periodically flushed to prevent saturation	• TCP probing has small transmission overhead
BLAM [14], 2018	• Lightweight • Less memory consumption	• No definition when to insert packet into Bloom Filter • Uses convention Bloom Filter • Bloom Filter is not flushed
SkyShield [7], 2018	• Bloom Filter filters the incoming request from legitimate users without reverse calculation of their IP addresses • Avoidance of retrieving exact IP addresses of malicious nodes avoids intensive computation process • Reuse of abnormal sketch improves efficiency • Dynamically determines the number of malicious nodes • Random aggregation property of sketches improves capability	• False positive response may block legitimate users • Hellinger distance may generate false alarms after the termination of attacks • Small detection time interval increases computation and false alarm rate • Fewer abnormal buckets in the hash table increases the number of false negatives • Increasing the number of abnormal buckets in the hash table increases the probability of false identification of legitimate users • Increase in threshold damping coefficient of Exponential Weighted Moving Average method allows malicious users
Dodig et al. [8], 2018	• $\overline{CBFI}$ helps in reducing matching false rate • Reduces incorrect detection of matching packets • Less memory consumed • Less processing required • Convenient for embedded systems	• Maintaining multiple Bloom Filter increases memory consumption • Counter overflow occurs • When the counter is half full, DCBF resets all slots
Tseung and Chow [9], 2018	• Self-learning • Advanced ML is used to reduce human intervention	• ML methods are highly computational • Bloom Filter is adjusted periodically • Allows malicious packet if incoming malicious packet rate is similar to normal packet rate
LogDos [15], 2020	• Prevents data flooding in ICN • Bloom Filter reduces the storage overhead at LogDos-Enabled routers • No communication overhead • Logging and verification is performed in the same router • Odd/Even Logging reduces computational overhead of the router • Dynamic logging reduces computational overhead • Even Logging has high attack verification probability compared to Odd Logging	•. Each router maintains two Bloom Filters • Dynamic logging is less effective • Bloom Filter size is based on GET message arrival rate • Bloom Filter size also depends on the number of routers and their link rate • Conventional Bloom Filter is used, which has high FPP

Regardless of the number of Bloom Filters, the overall memory consumption by the Bloom Filters is still low. As we know, the time complexity of the query operation of Bloom Filter is $O(1)$; for this reason, Bloom Filter can handle DoS/DDoS attacks smoothly without deteriorating the performance of the application.

14.3.1 Defense techniques against DoS

Halagan et al. [16] proposed a CBF-based TCP SYN Flood attack detection technique. The technique uses CBF to classify the attack as random, subnet, and fixed. An attack is random when each packet has a randomly generated spoofed source IP address. An attack is called a subnet if each packet has a specific subnet range generated spoofed

source IP address. Similarly, a fixed attack is defined as that using selected IP addresses. A modified CBF, called MCBF, is also proposed to reduce the memory consumption of CBF. MCBF maintains a single vector of counters. The technique maintains two tables, one for storage of source IP addresses and the other for storing destination IP addresses. Both tables use MCBF, which increments the counter when an SYN packet is received and decrements the counter when an ACK packet is received. The operations are performed in both tables. During the DoS attack, the Bloom Filter size increases quickly. When it crosses a threshold value, an alarm is raised. The distribution of values in MCBF is analyzed to determine the type of attack. MCBF is not scalable; hence, the Bloom Filter may become saturated and increase the FPP during the attack.

14.3.1.1 SIP

Session Initiation Protocol (SIP) is a signaling protocol for voice and video communication [17]. It also supports other multimedia applications. SIP is implemented in a trusted and pre-arranged environment that uses passwords for communication. However, SIP is vulnerable to many attacks such as call hijacking, Spam over Internet Telephony (SPIT), toll fraud, vishing, and DoS. Under a DoS attack, the most severe attack is an INVITE flooding attack, in which attackers send a massive number of INVITE requests to the server. This affects both SIP proxy and end-user. Users remain connected due to proxy for a while, which makes them vulnerable. The flooding attacks are classified into three types; (a) single-source flooding when a single attacker sends flooding INVITE requests, (b) multiple source flooding, i.e., DDoS attack, and (c) SIP reflecting flooding which uses spoofing to generate INVITE requests.

Wu et al. [18] proposed two schemes based on CBF against SIP flooding attacks. One scheme uses CBF, and the other uses PFilter. PFilter is a horizontally compressed CBF, which stores signatures of SIP messages. When a SIP message is received, the CBF is searched to check the signature, i.e., CBF does authentication. When SIP messages are received, the counter of CBF is incremented till it reaches a threshold value. After crossing the threshold, the SIP message is considered a malicious message. Its signature is deleted and stored on a blacklist. However, this scheme uses CBF, which consumes more memory and has an overflow issue. Therefore, PFilter is proposed, which has comparatively less length, making it lightweight. To prevent the attack, PFilter functions similar to CBF. Both schemes have a low threshold value which is inefficient in the case of the flash crowd. Moreover, in the case of multi-attributed flooding attacks, the cost for detection is higher because multiple Bloom Filters have to be maintained. See Table 14.2.

TABLE 14.2 Features and limitations of defense techniques against DoS.

Technique	Features	Limitations
Halagan et al. [16], 2015	• MCBF consumes less memory compared to CBF • Classifies the attack	• MCBF has overflow issue • MCBF consumes more memory compared to a conventional Bloom Filter • MCBF has a scalability issue • Not hard to bypass the defense of the technique
Wu et al. [18], 2017	• PFilter occupies less memory compared to CBF • PFilter is lightweight • Exponential Weighted Moving Average determines the dynamic threshold value to decide on SIP attack • In case of multi-attributes flooding attacks, PFilter has low detection cost	• Scheme I occupies more memory and has an overflow issue • Inefficient in case of false crowd • Cost is more in case of multi-attributes flooding attacks • PFilter performance is low in low-rate flooding attack

14.4 Defense against various network attacks

This section discusses the role of Bloom Filter in preventing network attacks. We also mention some network attacks that target Bloom Filter.

14.4.1 Pollution attack

Pollution attack is initiating a large number of insertion operations by the attacker aimed to increase FPP [19]. Inserting a vast amount of data into Bloom Filter makes the data structure saturated, eventually making the FPP

equal to 1. A solution for this issue is detecting the fullness of the Bloom Filter. When the Bloom Filter is detected as full, then another Bloom Filter is created, or the same Bloom Filter is dynamically readjusted. On the other hand, the continuation of this problem results in complete or huge occupancy of memory by the Bloom Filter. Pollution attack is possible in an uncontrolled environment, but it is very difficult in a controlled environment. Proper authentication and cryptography in a controlled environment create many layers of obstacles for attackers to attack the Bloom Filter.

14.4.2 Query-only attack

A query-only attack is forwarding a massive number of queries to Bloom Filter aiming to deteriorate the services [19]. However, the $O(1)$ time complexity of Bloom Filter makes it difficult for attackers to deteriorate the performance of the Bloom Filter. Fig. 14.1 validates that Bloom Filter is capable of handling massive requests without becoming an overhead for the security applications. Thus, Bloom Filter is immune to query-only attacks.

14.4.3 Deletion attack

A deletion attack deletes information from Bloom Filter on a large scale. Deleting from Bloom Filter results in removing legitimate users from Bloom Filter to fail them during the authentication process. This attack is possible in an uncontrolled environment but very difficult in a controlled environment, as discussed in the pollution attack. As we know, a deletion operation is not a legit operation in Bloom Filter. Hence, initiating the delete operation self-proclaims that the system is being attacked. Therefore, deletion attacks are impossible against Bloom Filter.

14.4.4 Brute force and dictionary attacks

Brute force and dictionary attacks primarily target password databases. The attacker tries to acquire the system password to gain control of the system. In a brute force attack, the attacker attempts to guess the password, whereas in a dictionary attack, the attacker attempts all possible passwords serial-wise like a dictionary. A dictionary attack is a type of brute force attack. Regrettably, Bloom Filter is not implemented against brute force attacks. The main reason is that the password database does not implement Bloom Filter. The issues with false positives of Bloom Filter may permit malicious persons to access a user's sensitive data. Similarly, due to false negative responses, a legitimate user may not be permitted to use his/her data, for example, banking. Thereupon, the users lose faith in the application.

14.5 Security

This section mentions various techniques based on Bloom Filter to improve the security of the network.

Jin and Papadimitratos [20] proposed a Bloom Filter-based pseudonym-validation technique to reduce computational overhead. A pseudonym is a short-term credential generated by Vehicular PublicKey Infrastructure (VPKI) using protocols [21]. It is used for authentication of messages and maintaining integrity along with privacy. Pseudonymous Certification Authority (PCA) issues pseudonyms in a domain. When a vehicle wants to pass a domain, it has to request pseudonyms from the corresponding PCA. This technique uses Bloom Filter to validate the pseudonyms. PCA constructs a CBF and inserts validly issued pseudonyms. Compressed Bloom Filter-delta [22] publishes pseudonym updates, i.e., insertion or deletion of pseudonyms to reduce update overhead. The vehicles download the standard Bloom Filter of corresponding CBF to verify their pseudonym. Standard Bloom Filter is downloaded because the vehicle does not require the counters to verify, and it occupies less memory compared to CBF. In case the pseudonym is absent, the vehicle can prefer the fallback approach. However, the computational overhead of the fallback approach is more compared to Bloom Filter. If the pseudonym is present in Bloom Filter, it is again checked in Fake Pseudonym List (FPL). FPL is a list of fake pseudonyms that are present in Bloom Filter. FPL reduces the damage caused by false positive responses of Bloom Filter. In case the pseudonym is present in FPL, it is reported to VPKI. FPL is periodically flushed to keep synchronized with the new version of Bloom Filter. The vehicle has to periodically download the latest version of Bloom Filter because new vehicles join the domain.

Gupta et al. [23] proposed a probabilistic data structure-based Intrusion detection system (IDS) called ProIDS. It uses a Bloom Filter and count-min sketch. The network administrator provides a set of suspicious nodes stored in the Bloom Filter. Count-min sketch counts the number of hits by the suspicious nodes for a fixed time period.

If the number of hits crosses a threshold within the time period, then the node is confirmed as a suspicious node and reported to the network administrator. Bloom Filter reduces storage usage and computational cost. Moreover, it improves the performance of membership testing. When a node requests a resource, first, it is checked in Bloom Filter. If present, the request is ignored, and count-min sketch increments the corresponding count of the number of hits. Flushing of Bloom Filter is not considered; hence, the list of suspicious nodes will never change. It is an issue because if a legitimate node got included into the list of suspicious nodes, it will never be acknowledged as a legitimate node.

Compressed Bitmap Index and traffic Downsampling (CBID) [24] is a Payload Attribution System (PAS). PAS is a tool mostly used for detecting criminals who committed cybercrime. It records and stores the network traffic for an extended time period. CBID consists of two techniques, namely, downsampling and flow determination. Downsampling filters junk traffic to reduce Bloom Filter insertion operation. Flow determination technique reduces exhaustive Bloom Filter query operations. It is based on a Bloom Filter and a compressed bitmap index. Blocks smaller than a fixed size are dropped. Only large blocks are inserted into Bloom Filter. The predefined size of the block is called the downsampling threshold. A high downsampling threshold reduces the FPP of Bloom Filter. But it also reduces the number of blocks that are queried, which increases FPP. CBID uses a Multi-Section Bloom Filter (MSBF) due to its high insertion rate. The whole Bloom Filter array is divided into sections where each acts as a Bloom Filter. During insertion, one hash function chooses one sub-Bloom Filter. Other hash functions hash the item and set the bit to 1 in the obtained locations in the chosen sub-Bloom Filter. CBID uses winnowing to select the block boundaries where it will be divided. The large blocks are checked in MSBF; if present, ignore, otherwise inserted into MSBF.

Goss and Jiang [25] proposed a distributed protocol to protect the firewall from both insider and outsider intruders. The firewall policies are obfuscated by distributing the policy evaluation function among many servers. The protocol integrates multi-party computation, secret sharing schemes, Bloom Filters and Byzantine agreement protocols. The distributed architecture requires a minimum of three servers to implement the protocol. However, the protocol fails when the number of failed systems is less than a threshold number. The firewall is distributed secretly using a secret sharing scheme to prevent a single server from tampering with the firewall data. The blacklist size and FPP determine the Bloom Filter size and the number of hash functions during firewall initialization. Then the secret sharing scheme is used to generate the secretly shared Bloom Filter. The secretly shared Bloom Filter contains a share of the firewall policy, which is sent to a server. Similarly, different Bloom Filters are sent to different servers to distribute the firewall policy. When a gateway receives a packet, its IP address is extracted and sent to the servers possessing the firewall policy share. All those servers check the IP address with their blacklist Bloom Filter and send the result to the gateway. In case all results are true, the packet is further processed, otherwise blocked. The computation is shared among the servers to reduce the computation of the gateway. After evaluation of the response of the Bloom Filter, the Byzantine agreement process is used to determine the overall decision regarding blocking or accepting the packet. However, it increases the number of communication bits.

Deng et al. [26] proposed Bloom Filter-based abnormality detection technique for port scanning and TCP flooding traffic. Standard Bloom Filter is used for port scanning and CBF for TCP flooding traffic. The port scanning detection technique has three modules: recording, statistics, and detection. The source and the destination address are retrieved in the recording module as the stream identifier. The identifier is checked with Bloom Filter; if present, the stream is transmitted to the statistics module. Otherwise, the stream is inserted into the Bloom Filter. The source address, destination address, and destination port are used as the stream identifier in the statistics module. The stream identifier is checked with the Bloom Filter; if present, the packet is transmitted to the final module, i.e., the detection module. Otherwise, the stream is inserted into Bloom Filter. The detection module has an array of common port numbers and information regarding the port access request of each stream. The stream identifier from the recording module is retrieved, and the identifier is concatenated with each common port number and checked in the Bloom Filter of the statistics module. If the stream identifier is present, then the number of ports corresponding to the stream is incremented. If the number of ports is vast, it is concluded as port scanning traffic. The abnormality detection technique for TCP flooding traffic also has three modules. The first module, i.e., the recording module, performs the same task as in the port scanning technique. The statistics module maintains two CBFs. The first CBF (say, CBF1) counts the number of streams using destination address and destination port as flow identifiers. The second CBF (say CBF2) counts the number of packets of specified length in the flow in which the destination address, destination port, and packet length are used as flow identifiers. When the packet arrives at the module, the flow identifier is checked with CBF1; if present, the counter of CBF1 is incremented. Likewise, SYN and FIN flag bits are checked; if any flag bit is 1, then the message length is 0. Otherwise, packet length is assumed as a multiple of 10. Similarly,

the packet lengths are considered to increase accuracy. Likewise, the flow identifier is checked in CBF2; if present, the counter is incremented. In the detection module, the number of messages from CBF1 and the number of current messages from CBF2 are retrieved. The ratio between the current messages and total messages is calculated. If the ratio is large, then it is concluded that there is TCP flooding traffic. See Table 14.3.

TABLE 14.3 Features and Limitations of Security Techniques.

Technique	Features	Limitations
Jin and Papadimitratos [20], 2016	• Reduces computational overhead • Compressed Bloom Filter-delta reduces update overhead • Downloads standard Bloom Filter of the CBF to reduce memory consumption • Computational overhead of fallback approach is more compared to Bloom Filter	• Periodically downloads the latest version of Bloom Filter • Maintains both Bloom Filter and a list, which increases memory consumption • Before downloading a new version of Bloom Filter, some valid pseudonym maybe rejected by Bloom Filter
Gupta et al. [23], 2017	• Bloom Filter reduces storage usage • Bloom Filter reduces computational cost • Bloom Filter improves the performance of membership testing	• Accuracy of Count-min sketch decreases with increase in the number of items • Conventional Bloom Filter is used • List of suspicious nodes is not changed
CBID [24], 2018	• Resolves the alignment problem • Downsampling reduces Bloom Filter insertion operations • Flow determination reduces exhaustive Bloom Filter query operations • High downsampling threshold reduces FPP • Multi-section Bloom Filter has high insertion rate	• Fewer blocks that are queried increases FPP • Bloom Filter size is large • Bitmap Index Table is larger than Bloom Filter, which reduces performance • Bitmap Index Table incurs compression cost
Goss and Jiang [25], 2018	• Data secure against local inspection and tampering from insider • Bloom Filter securely and obliviously evaluates the firewall criteria checking function • Distribution of the firewall policy evaluation function increases the tampering difficulty • Secret sharing scheme distributes the firewall data securely • Multi-party computation decreases the gateway computation	• Distributed architecture requires a minimum of three servers • Architecture fails when the number of failed systems is less than a threshold number • Gateway has to wait for a fixed time to decide whether to accept or block a packet • Servers are commodity hardware, thus the scheme has high failure rate • Multi-party computation increases the number of communication bits • Bloom Filter is not flushed, hence the legal IP address becomes illegal due to a false positive response
Deng et al. [26], 2019	• Packet length is assumed as a multiple of 10 to increase accuracy • Technique is simple • Bloom Filter increases the accuracy of abnormality detection	• CBF consumes more memory • Standard Boom Filter has more false positives

14.6 Privacy

Privacy protects the user data, for example, the services or sensitive data stored by the user. These services should be kept private even from the service provider's administrator. Despite that, the services' provider requires checking the service list to provide the acquired services of the user. Although this may be required, maintaining such a list violates the user's privacy rights and makes that list a vulnerable point. Furthermore, solving this issue while maintaining a good quality of service is challenging. Hence, Bloom Filter is implemented to store the service list. The service list is inserted into the Bloom Filter, which, before providing the service, checks the permission of the user to access the service. Bosch [27] proposed a Bloom Filter-based privacy-preserving geofencing service. Geofencing of a vehicle defines the permitted/restricted geographical areas. Bloom Filter stores the GPS locations, and it is searched when the driver requests a service. The Bloom Filter response determines whether to provide the service or inform the fleet operator in case the geographical area restriction is violated. Bloom Filter protects the privacy of the vehicle driver within the permitted geofence. Another side of the coin is that an optimal and fast searching algorithm is

required to minimize the service search overhead. The minimum time complexity for searching algorithms is $\log(n)$, but if the n value increases, the $\log(n)$ value will also increase. Considering this issue, Bloom Filter is an excellent choice for such applications.

Calderoni et al. [28] proposed a scheme to provide location privacy using SpatialBF. The geographical area is divided into many sub-areas based on some interest in the scheme. There is direct communication between the user and the service provider in the first scenario. The service provider constructs SpatialBF by inserting the sub-areas. Then SpatialBF is encrypted and sent to the user. The user inserts its current location to verify its presence. This scheme protects the user's current location privacy from the service provider. In the second scenario, there is a third party between the user and the service provider. The service provider creates the SpatialBF and sends it to the third party. The user sends its location to the third party, and the third party returns the response. This setup reduces communication costs, and the user does not require high computational resources.

Li et al. [29] proposed a top-k query processing scheme to maintain data privacy and integrity of query results. Bloom Filter is used in the data privacy technique. Top-k query processing is the query to search for k largest or smallest data items received from a particular sensed area—for example, queries for environmental information such as temperature, pollution index, or humidity. For data privacy, each data item is encrypted using a secret key by the sensor. Then the sink performs indexing. The index and encrypted data items are stored in the storage system. The data privacy scheme has four steps: approximating uniform data distribution, data partitioning, interval embedding, and index selection. Bloom Filter is used in the index selection. For a range, the set is divided into multiple disjoint prefixes such that after the union, it will give the same range. These prefixes are hashed by l hash functions and stored in Bloom Filter. When a data item is queried, the item is hashed by l hash functions to obtain the locations in the Bloom Filter array. If all locations have a bit value of 1, the item is present; otherwise, it is absent. The security is increased by hashing the prefixes using HMAC keyed-hash function. Then, these secured prefixes are inserted into Bloom Filter. This scheme has high FPP; hence, a method called press-on-towards is implemented.

Han et al. [30] proposed a CP-ABE (ciphertext-policy attribute-based encryption) scheme to provide privacy. The scheme uses Bloom Filter to protect the user's attribute values from attribute authority. Attribute authority generates master secret keys and public parameters and provides access to users based on service attributes. Attribute Bloom Filter (ABF) is used in the encryption phase. ABF hides attribute values in the ciphertext. ABF uses Garbled Bloom filters (GBF) to insert attributes. GBF [31] is a garbled version of conventional Bloom Filter. During insertion, the input item is divided into k parts using XOR-based secret sharing. Secret sharing is a method to divide a secret and send it to multiple users. The secret is retrieved when all parts are combined. In XOR-based secret sharing, the secret is divided into $r-1$ parts, and the last part is XOR of all $r-1$ parts. Secret retrieving is XOR of all r parts. GBF hash the item by k hash functions and generate k array locations. Each part obtained by XOR-based secret sharing is inserted into GBF in one of the hash locations. Similarly, all k parts are inserted into GBF. During query operation, the item is hashed to obtain k locations. When all k locations array values are XORed, and the result is the item, then the item is present in GBF. The input item in ABF is the concatenation of the attribute and the index number string. The index number is the location/index of the access policy matrix. The input item is inserted following the insertion operation of GBF. ABF and access matrix are transmitted with the ciphertext. After receiving the ciphertext, ABF checks the existence of attributes in GBF. ABF permits access to the user when the attributes of the user are present in GBF.

Bosch [27] proposed a privacy-preserving geofencing service. Some vehicles are allowed within certain geographical areas. Geofencing is territory tracking for these vehicles. Bloom Filter helps in territory tracking. The fleet operator constructs the Bloom Filter. The GPS location of the permitted geofence is encrypted and hashed by the hashing function. Then the hashed value is inserted into the Bloom Filter. When a vehicle requests a service, the service provider checks the vehicle's location in the Bloom Filter. If present, the services are provided. Otherwise, the service provider again checks the location after some time. Due to the FPP of Bloom Filter, the service provider waits for some fixed time and again checks the location in Bloom Filter. If, in this case, the vehicle is outside the geofence, then a report is sent to the fleet operator. Bloom Filter helps in maintaining privacy. Bloom Filter does not directly store the location; hence, the service provider or fleet operator does not know the vehicle's location. They become aware of the location when the vehicle is outside the geofence. Two Bloom Filter are maintained, namely, In(B) and Out(B). In(B) stores permitted geofence location, and Out(B) stores restricted geofence locations.

Xue et al. [32] proposed a data-matching technique where Bloom Filter helps in preserving privacy. Initially, ML algorithms reduce the dimensions of the dataset. To the compressed data, the neighbor list is added to increase the information. Then the data is hashed and inserted into the Bloom Filter. Noise is added into Bloom Filter to reduce membership attacks. XOR method adds random bits as noise. The output of the perturbed Bloom Filter is used to

train the classifiers. The dataset is converted into another domain using Bloom Filter to address the data range attack. Hence, it becomes difficult for attackers to decrypt the dataset. See Table 14.4.

TABLE 14.4 Features and Limitations of Privacy Techniques.

Technique	Features	Limitations
Calderoni et al. [28], 2015	• SpatialBF allows private computation of location-based information • Scheme provides location privacy • Scheme does not require trust among the user, service provider or third party • Third party presence reduces communication cost	• Direct communication between the user and service provider has a larger communication cost • In direct communication, a user requires more computational resources
Li et al. [29], 2017	• Multiple query is aggregated into a single query • Data integrity is achieved by executing a data partitioning algorithm • Secured by following the IND-CKA security model • Secured prefixes are inserted into Bloom Filter	• Maintains multiple Bloom Filters • Scheme has high FPP • False positive responses permit selection of non-top-k data item • False positive responses lead to a waste of bandwidth
Han et al. [30], 2018	• Hides key generation attribute values from attribute authority • Hides the access policy from server during searching • Oblivious transfer technique hides attribute values • ABF hides attribute values in the ciphertext • Offline encryption reduces the computation tasks	• ABF construction increases communication cost • Checking attributes with ABF before decryption adds extra overhead • Sending extra information for protection of attributes increases time cost
Bosch [27], 2018	• Bloom Filter maintains privacy • Same geofence vehicles share same Bloom Filter to reduce memory consumption	• One Bloom Filter for each vehicle • Maintenance of multiple Bloom Filters consumes more memory
Xue et al. [32], 2020	• Bloom Filter maintains privacy • Noise in Bloom Filter decreases membership attack	• Noise in Bloom Filter makes it vulnerable to differential attack • Increase in dimension increases FPP • Machine learning algorithms are highly computational

14.7 Evaluation

Fig. 14.1 shows that Bloom Filter acts as a roadblock for the adversary's advancement. It prevents many adversary attacks that try to access sensitive data. Moreover, it creates another complex barrier for the adversaries to overcome.

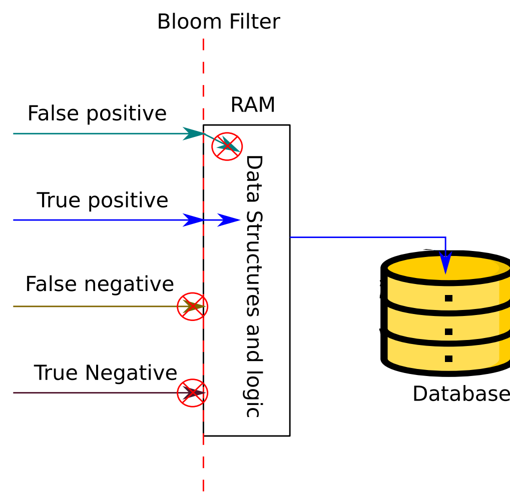


FIGURE 14.1 Querying Bloom Filter to get access to sensitive data. Bloom Filter is a barrier for adversaries.

We have experimented to evaluate the query protection capacity to protect sensitive data. We generated 30 million unique input items, out of which 10 million are considered as valid input and 20 million as attacks. This experiment is conducted in low-cost desktop hardware on Ubuntu 18.04.4 LTS operating system where CPU configuration is Intel Core i7-7700 CPU @ 3.60 GHz and RAM size is 8 GB. For evaluating the attack, we use 8L HFil Bloom Filter [33]. The desired false positive probability has been set to 0.01, which transcribes the memory requirements of 36 MB. In this experiment, by using Bloom Filter, out of 20M attacks, 54,835 attack requests are bypassed. As the 8L HFil takes 12.736031 seconds to filter all 30M queries, the Bloom Filter does not deteriorate the performance of the security of privacy systems. To access sensitive data, the 54,835 attacks have to bypass the next layer of security. Thus, even if there are 20 million attacks or DDoS, there is less load on the server. Therefore, Bloom Filter is a fortifier of security and privacy applications, but not a complete solution.

14.8 Discussion

Bloom Filter has two main issues due to false positives and false negatives. However, the probabilities of false positives and false negatives in the Bloom Filter are very low. This section discusses how these two issues do not affect the security of the network. Table 14.5 highlights some parameters of all the techniques discussed in this chapter.

TABLE 14.5 Comparison of Network Security Techniques.

Technique	Bloom Filter	FP	FN	Attack
Wu et al. [18]	CBF and PFilter	✓	×	SIP flooding
Halagan et al. [16]	MCBF [16]	✓	✓	DoS
Tseung and Chow [9]	CBF	✓	×	DDoS
SkyShield [7]	Conventional	✓	×	Application layer DDoS
BLAM [14]	Conventional	✓	×	Interest flooding
Jin and Papadimitratos [20]	Conventional, CBF and Compressed Bloom Filter [22]	✓	✓	Security
Li et al. [29]	Conventional	✓	✓	Privacy
Bosch [27]	Conventional	✓	✓	Privacy
Xue et al. [32]	Conventional	✓	✓	Differential Privacy
Dodig et al. [8]	Dual [8]	✓	×	DDoS
Saurabh and Sairam [5]	Wrap [5]	✓	×	DDoS
CBID [24]	MCBF	✓	✓	Security
Gupta et al. [23]	Conventional	✓	✓	Intrusion
Xiao et al. [12]	Conventional	✓	✓	Link Flooding
Deng et al. [26]	Conventional, CBF	✓	✓	Port scanning and TCP flooding
Han et al. [30]	Attribute [30]	✓	×	Privacy
Goss and Jiang [25]	Conventional	✓	✓	Firewall data tampering
LogDos [15]	Conventional	✓	✓	Data Flooding

14.8.1 False negative

The deletion operation is not executed in the network security techniques. One of the main reasons is that the failure nodes in the network do not notify other nodes. The network nodes are commodity hardware that is prone to failure. Sometimes the reason is unrelated to hardware, so after a short time period, the nodes become alive. In some scenarios, when the nodes receive the notification about the failure, node deletion operations are usually not preferred due to the periodical flushing of Bloom Filter. Examples of such techniques are SkyShield [7] and that of Gupta et al. [23]. Generally, the security techniques use Bloom Filter to store either legitimate or malicious nodes, or both. Due to false positives, sometimes the legitimate nodes get added to the Bloom Filter storing the malicious nodes, and malicious nodes are added to Bloom Filter storing the legitimate nodes. Due to this reason, Bloom Filter is periodically flushed. Thus, a periodic flush negates the requirement of the deletion operation, thereupon reducing the false negative probability.

14.8.2 False positive

The issue of false positives has been extensively researched, and the state-of-the-art Bloom Filter has developed a false positive free zone [34,35]. The most important point to consider in a conventional Bloom Filter is that the probability of false positives is very low. Besides, other Bloom Filter variants are successful in further reducing this probability. Hence, the false positive responses have less impact on the performance of the application. Let us consider a network security technique addressing the DDoS attack. In most of the cases, CBF counts the number of requests sent from a node (e.g., Tseung and Chow [9] and Wu et al. [18]). False positive responses cause some of the legitimate nodes to be considered malicious, and their requests are dropped. Later after some time, when Bloom Filter is flushed, the request of legitimate nodes will be accepted. As network failure is a common occurrence, delay in acceptance of request does not affect the users. Hence, false positive responses have a lesser negative impact on security and privacy techniques.

14.9 Conclusion

The main problem with network security and data protection techniques is scanning millions or billions of incoming packets per second. Delivering such a large number is an arduous task. Therefore, most security and privacy techniques use Bloom Filters to solve the problem. The two main problems, i.e., false negatives and false positives, make Bloom Filters unsuitable for some security and privacy techniques. However, these issues have a negligible impact on the performance in a practical scenario. Additionally, other mechanisms are used to address these issues, such as flushing the Bloom Filter on a regular basis to allow access by legitimate users who are deemed malicious based on the false positive response. In addition, Bloom Filter architecture and operation can shield the applications against various attacks in the network.

References

[1] S. Sharwood, Github wobbles under DDoS attack, http://www.theregister.co.uk/2015/08/26/github_wobbles_under_ddos_attack/, Aug 2015. (Accessed 20 June 2018).

[2] B. Schneier, Lessons from the DYN DDoS attack, https://www.schneier.com/blog/archives/2016/11/lessons_from_th_5.html, Nov 2016. (Accessed 20 June 2018).

[3] M. Lad, X. Zhao, B. Zhang, D. Massey, L. Zhang, Analysis of BGP update surge during Slammer worm attack, in: International Workshop on Distributed Computing, Springer, 2003, pp. 66–79.

[4] J. Czyz, M. Kallitsis, M. Gharaibeh, C. Papadopoulos, M. Bailey, M. Karir, Taming the 800 pound gorilla: the rise and decline of NTP DDoS attacks, in: Proceedings of the 2014 Conference on Internet Measurement Conference, ACM, 2014, pp. 435–448.

[5] S. Saurabh, A.S. Sairam, Increasing the effectiveness of packet marking schemes using wrap-around counting Bloom filter, Secur. Commun. Netw. 9 (16) (2016) 3467–3482.

[6] L. Kavisankar, C. Chellappan, S. Venkatesan, P. Sivasankar, Efficient SYN spoofing detection and mitigation scheme for DDoS attack, in: Recent Trends and Challenges in Computational Models (ICRTCCM), 2017 Second International Conference on, IEEE, 2017, pp. 269–274.

[7] C. Wang, T.T. Miu, X. Luo, J. Wang, Skyshield: a sketch-based defense system against application layer DDoS attacks, IEEE Trans. Inf. Forensics Secur. 13 (3) (2018) 559–573.

[8] I. Dodig, V. Sruk, D. Cafuta, Reducing false rate packet recognition using dual counting Bloom filter, Telecommun. Syst. 68 (1) (2018) 67–78.

[9] C.Y. Tseung, K.P. Chow, Forensic-aware anti-DDoS device, in: 2018 IEEE Security and Privacy Workshops (SPW), 2018, pp. 148–159.

[10] R.U. Rasool, U. Ashraf, K. Ahmed, H. Wang, W. Rafique, Z. Anwar, Cyberpulse: a machine learning based link flooding attack mitigation system for software defined networks, IEEE Access 7 (2019) 34885–34899.

[11] S.B. Lee, M.S. Kang, V.D. Gligor, Codef: collaborative defense against large-scale link-flooding attacks, in: Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies, CoNEXT'13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 417–428.

[12] P. Xiao, Z. Li, H. Qi, W. Qu, H. Yu, An efficient DDoS detection with bloom filter in SDN, in: 2016 IEEE Trustcom/BigDataSE/ISPA, 2016, pp. 1–6.

[13] S. Choi, K. Kim, S. Kim, B. Roh, Threat of DoS by interest flooding attack in content-centric networking, in: The International Conference on Information Networking 2013 (ICOIN), 2013, pp. 315–319.

[14] G. Liu, W. Quan, N. Cheng, B. Feng, H. Zhang, X.S. Shen, Blam: lightweight Bloom-filter based DDoS mitigation for information-centric IoT, in: 2018 IEEE Global Communications Conference (GLOBECOM), 2018, pp. 1–7.

[15] B. Al-Duwairi, O. Ozkasap, A. Uysal, C. Kocaogullar, K. Yildirim, LogDoS: a novel logging-based DDoS prevention mechanism in path identifier-based information centric networks, arXiv:2006.01540, 2020.

[16] T. Halagan, T. Kováčik, P. Trúchly, A. Binder, SYN flood attack detection and type distinguishing mechanism based on counting Bloom filter, in: Information and Communication Technology, Springer, 2015, pp. 30–39.

[17] I. Hussain, S. Djahel, Z. Zhang, F. Naït-Abdesselam, A comprehensive study of flooding attack consequences and countermeasures in session initiation protocol (SIP), Secur. Commun. Netw. 8 (18) (2015) 4436–4451, https://doi.org/10.1002/sec.1328.

[18] M. Wu, N. Ruan, S. Ma, H. Zhu, W. Jia, Q. Xue, S. Wu, Detect SIP flooding attacks in VoLTE by utilizing and compressing counting Bloom filter, in: International Conference on Wireless Algorithms, Systems, and Applications, Springer, 2017, pp. 124–135.

[19] T. Gerbet, A. Kumar, C. Lauradoux, The power of evil choices in Bloom filters, in: 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, 2015, pp. 101–112.

[20] H. Jin, P. Papadimitratos, Proactive certificate validation for VANETs, in: 2016 IEEE Vehicular Networking Conference (VNC), 2016, pp. 1–4.

[21] W. Whyte, A. Weimerskirch, V. Kumar, T. Hehn, A security credential management system for V2V communications, in: 2013 IEEE Vehicular Networking Conference, 2013, pp. 1–8.

[22] M. Mitzenmacher, Compressed Bloom filters, IEEE/ACM Trans. Netw. 10 (5) (2002) 604–612.

[23] D. Gupta, S. Garg, A. Singh, S. Batra, N. Kumar, M.S. Obaidat, ProIDS: probabilistic data structures based intrusion detection system for network traffic monitoring, in: GLOBECOM 2017–2017 IEEE Global Communications Conference, 2017, pp. 1–6.

[24] S.M. Hosseini, A.H. Jahangir, An effective payload attribution scheme for cybercriminal detection using compressed bitmap index tables and traffic downsampling, IEEE Trans. Inf. Forensics Secur. 13 (4) (2018) 850–860.

[25] K. Goss, W. Jiang, Distributing and obfuscating firewalls via oblivious Bloom filter evaluation, arXiv:1810.01571, 2018.

[26] F. Deng, Y. Song, A. Hu, M. Fan, Y. Jiang, Abnormal traffic detection of IoT terminals based on Bloom filter, in: Proceedings of the ACM Turing Celebration Conference – China, ACM TURC'19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1–7.

[27] C. Bösch, An efficient privacy-preserving outsourced geofencing service using Bloom filter, in: 2018 IEEE Vehicular Networking Conference (VNC), 2018, pp. 1–8.

[28] L. Calderoni, P. Palmieri, D. Maio, Location privacy without mutual trust: The spatial Bloom filter, in: Security and Privacy in Unified Communications: Challenges and Solutions, Comput. Commun. 68 (2015) 4–16, https://doi.org/10.1016/j.comcom.2015.06.011.

[29] R. Li, A.X. Liu, S. Xiao, H. Xu, B. Bruhadeshwar, A.L. Wang, Privacy and integrity preserving top- k query processing for two-tiered sensor networks, IEEE/ACM Trans. Netw. 25 (4) (2017) 2334–2346.

[30] Q. Han, Y. Zhang, H. Li, Efficient and robust attribute-based encryption supporting access policy hiding in Internet-of-Things, Future Gener. Comput. Syst. 83 (2018) 269–277, https://doi.org/10.1016/j.future.2018.01.019.

[31] C. Dong, L. Chen, Z. Wen, When private set intersection meets Big Data: an efficient and scalable protocol, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS'13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 789–800.

[32] W. Xue, D. Vatsalan, W. Hu, A. Seneviratne, Sequence data matching and beyond: new privacy-preserving primitives based on Bloom filters, IEEE Trans. Inf. Forensics Secur. 15 (2020) 2973–2987.

[33] R. Patgiri, HFil: a high accuracy Bloom filter, in: 2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2019, pp. 2169–2174.

[34] O. Rottenstreich, I. Keslassy, The Bloom paradox: when not to use a Bloom filter, IEEE/ACM Trans. Netw. 23 (3) (2015) 703–716, https://doi.org/10.1109/TNET.2014.2306060.

[35] S.Z. Kiss, E. Hosszu, J. Tapolcai, L. Ronyai, O. Rottenstreich, Bloom filter with a false positive free zone, in: IEEE INFOCOM 2018 – IEEE Conference on Computer Communications, 2018, pp. 1412–1420.

PART III

Applications of Bloom Filter in other domains

This page intentionally left blank

CHAPTER

15

Applications of Bloom Filter in Big data

15.1 Introduction

With the advancement of computational technology, all information is saved digitally in computer systems. This transition generated a huge volume of data. However, different fields have different data formats, which leads to the generation of complex data that are not structured. Thus, Big data has structured, unstructured, or semi-structured data. However, one big issue with Big data is that traditional data processing and analysis techniques are inefficient in handling Big data. Therefore, Big data has become a new field that demands new technology for highly efficient data generation, collection, visualization, processing, analysis, and storage techniques.

Big data is defined using three dimensions, i.e., 3Vs, namely volume, velocity, and variety [1]. Volume is the quantity of data, velocity is the speed of data generation, and variety is the type of data. However, Patgiri and Ahmed [2] added new dimensions to the concept of Big data, namely $V_3^{11} + C$ where there are 11Vs, 3 volume characteristics of volume, and complexity. The 11Vs are volume, velocity, variety, veracity, validity, value, visualization, variability, vendee, vase, and virtual. Moreover, there are three volume characteristics: voluminous, vacuum, and vitality. Section 15.2 provides further elaboration on Big data. Different technologies are required to handle different characteristics of Big data. However, volume plays a vital role in handling Big data. Data reduction requires lower memory for storage, lower resources and time for data processing and analysis, and lower bandwidth for data transmission in the network. Hence, there is a requirement of data filtering to remove meaningless and redundant data during data generation. Unfortunately, another characteristic of Big data adds complexity to this filtering process. Currently, the data generation speed is overwhelming. The total data volume generated, collected, copied, and consumed globally is predicted to be 64.2 zettabytes in 2020 [3]. Furthermore, within 2025 the global data generation will be around 180 zettabytes [3]. The sudden acceleration is due to the COVID-19 pandemic, where everyone worked from home. Under these circumstances, the filtering technique requires a data structure capable of efficiently handling the incoming data while maintaining high accuracy.

Bloom Filter is the simple answer for the requirement of data structure. Bloom Filter requires a low memory footprint, which lowers overhead on the application. Lower memory requirement also helps in deploying Bloom Filter in multiple units for high performance. Bloom Filter takes constant time for its operations; hence, it can easily handle the high data incoming speed. Furthermore, the error in Bloom Filter is low, so Bloom Filter also maintains high accuracy. This chapter sheds light on the role of Bloom Filter in Big data. Furthermore, the chapter includes the role of Bloom Filter in the database and MapReduce. Section 15.2 is about data management which provides a brief definition of Big data. Section 15.3 provides a review of Bloom Filter-based techniques in the database. This section also includes privacy-preserving record linkage, brief explanation and a review. Section 15.4 presents a short explanation of MapReduce and a review of Bloom Filter-based techniques in MapReduce. Section 15.5 presents some discussion on the role of Bloom Filter in data management, databases, and MapReduce. Finally, Section 15.6 concludes the chapter.

15.2 Data management

Initially, Big Data was defined by 3Vs [1]: volume, velocity, and variety. However, the Big data concept is bigger, which cannot be explained using just these 3Vs. Patgiri and Ahmed [2] extended the 3Vs to $V_3^{11} + C$ where there are 11Vs, 3 volume characteristics, and complexity. The 11Vs are volume, velocity, variety, veracity, validity, value, visualization, variability, vendee, vase, and virtual. Volume refers to the huge set of data that grows exponentially without any bound. The three characteristics of volume are voluminous, vacuum, and vitality. Voluminous refers to

Bloom Filter https://doi.org/10.1016/B978-0-12-823520-1.00022-5
Copyright © 2023 Elsevier Inc. All rights reserved.

data quantity. Vacuum is the requirement of free memory space to store the incoming data in huge volume. Vitality indicates the huge active data; however, there is a huge volume of data that is inactive. Velocity refers to the speed at which the data is generated. Variety indicates the structure of data. Big data has three structures: structured, semi-structured, and unstructured. Examples of structured data are tables in databases, semi-structured data comprises logs of documents that are partly structured and partly unstructured, and unstructured data are images, videos, etc. Veracity refers to the meaningfulness and accuracy of data. Validity is the data correctness. The value indicates the information stored in the data. Visualization refers to the visualization techniques used to derive various information from the data. Variability is the dynamic data whose value keeps on changing. Vendee indicates the owner of the data. The vase is the container of data. Big data is stored in huge farmhouses, which require huge land and electric power resources to keep the thousands of systems running, thousands of workers, etc. Virtual refers to effective and efficient data management as per the user demand. Complexity refers to the complexity in processing Big data. This section presents a review of various schemes and techniques based on Bloom Filter for data management. Table 15.2 highlights various features and limitations of the Bloom Filter-based data management techniques. See Table 15.1.

MultiDimensional Segment Bloom Filter (MDSBF) [4] is a technique to perform range queries for data management. Every item maintains heterogeneous resources and its related information such as storage, computation, available time, network bandwidth capacity, and energy. MDSBF maintains a hash table and multiple Bloom Filters. The hash table stores the itemID. Each slot of hash table points to multiple Bloom Filter segments. A Bloom Filter segment represents an attribute. Each Bloom Filter segment consists of multiple Bloom Filters. Each Bloom Filter stores data within a certain range. Hence, different Bloom Filters store data of different ranges. MDSBF has a load balancing issue. To solve the issue, each Bloom Filter is associated with a counter. The counter is incremented when an item is inserted into the corresponding Bloom Filter. The load balancing module consists of three stages: Initialization, Record, and Adjustment. Bloom Filter segments and Bloom Filters are constructed during initialization. The items are inserted into the Bloom Filters, and the counter is updated during the record stage. In the adjustment stage, a new Bloom Filter is assigned to the heavily loaded Bloom Filter to increase its capacity. A Bloom Filter becomes heavily loaded if the FPP crosses a threshold value. The new Bloom Filter is not constructed; rather, a light-loaded Bloom Filter is selected. Two adjacent Bloom Filters having the least counter value are selected. The range of the two Bloom Filters is combined, and all the values are inserted into a single Bloom Filter by union operation of the two Bloom Filters. Another Bloom Filter is flushed and assigned to the heavily loaded Bloom Filter. The technique implements CBF to handle dynamic data. During the record stage, first, it is checked if the itemID is present in the hash table. If not present, then the itemID is saved in the hash table. Then the Bloom Filter segments and Bloom Filters are constructed. In the case of insertion in Bloom Filter, first, the segment for the item is determined, then the hash value to insert into the corresponding Bloom Filter ir obtained, and the counter of the Bloom Filter is incremented. During query operation, multiple items are queried. Hence, all itemIDs are stored in a linked list. The minimum and maximum ranges of the itemIDs are determined. Also, the algorithm calculates the number of segments with the ranges. The items from the linked list are queried to the respective segment and Bloom Filter. If the Bloom Filter returns $True$, then the itemID is stored in another linked list, which is the result list.

Jang et al. [5] presents a Bloom Filter-based deduplication algorithm. The algorithm consists of two algorithms, Bloom Filter and source-based deduplication. First, the data passes through the Bloom Filter algorithm, which performs data division, hash table generation, determination of duplicated data, and data transmission functions. Next, the data passes through a source-based deduplication algorithm which performs a comparison of the hash value and duplicated data, data deduplication, and transfer of non-duplicated data. Initially, the data are classified based on their size. Then the data are hashed by a hash function. The hash values are inserted into a Bloom Filter. Then the new Bloom Filter is compared with old Bloom Filters. If the new Bloom Filter has the same bits as old Bloom Filters, then it confirms the presence of duplicated data. The data then is forwarded to a source-based deduplication algorithm which removes the duplicate data and transmits non-duplicated data to the server. The comparison of Bloom Filters based on bits is inefficient. Because the Bloom Filter may have a k value of more than 1, the bits obtained after mapping data to the Bloom Filter vector are not adjacent. Also, due to the mapping of many data to Bloom Filter, the majority of bits of the vector may be set to 1. In such cases, determining the presence of duplicate data is difficult.

BloomTree [6] is a tree index for solid-state drives (SSDs) to reduce the read and write operations. BloomTree has three types of leaf nodes: Normal Leaf, Overflow Leaf (OF-leaf), and Bloom Filter Leaf (BF-leaf). A normal leaf node occupies one memory page. OF-leaf occupies three pages. When an OF-leaf requires more than three pages, it is converted into a BF-leaf. Reading an OF-leaf requires $(No+1)/2$ where the No is the number of nodes in the tree. In contrast, the BF-leaf requires two-page reading, leaf-head page and data page. The BF-leaf node has two components, leaf-head node and data nodes. The leaf-head node has the Bloom Filters of the data nodes and pointers to the data

TABLE 15.1 Features and Limitations of Data Management Techniques.

Technique	Features	Limitations
Lv et al. [9]	• Identifies the persistent item in the Big data stream • Data are stored in secondary memory in a very compact form • Persistent item is determined only by using the Bloom Filters	• For every data stream IBLT and a Bloom Filter is stored in the secondary storage • To determine the persistent data, all data of any data stream are queried to the Bloom Filter • A false positive result eliminates data from being persistent • Implements standard Bloom Filter
MDSBF [4]	• Maintains information of many attributes • A range query searching technique • Implements CBF to handle dynamic data • New segments are assigned to loaded segment • Old two segments are merged to relieved one segment during assignment of a new segment to loaded segment • Does not construct new Bloom Filters to resolve load balancing issue	• Maintains many Bloom Filters • The number of Bloom Filters depends on the number of attributes and the range • The number of Bloom Filters also depends on partition of data range • Difficult in load balancing
Jang et al. [5]	• Deduplication algorithm • Removes duplicate data to reduce the amount of data send to a server from the client • Bloom Filter helps to reduce the number of comparisons to determine the duplicate data	• Comparison of two Bloom Filters based on bits to determine the duplicate data is inefficient • Implements standard Bloom Filter
BloomTree [6]	• BloomTree is a tree index for solid-state drives (SSDs) • Reduces the number of read and write operations • Bloom Filter enhances the read performance of the overflow page • Leaf-head node stores the Bloom Filter and metadata of the data nodes • Splitting of BF-leaf node leads to construction of normal nodes	• False positives increase the number of pages read in the BF-leaf • No insertion is permitted in solid node • Deletion of data from solid node is costly • Deletion of data from solid node requires additional write • Deletion of data from solid node reconstruction of Bloom Filter is needed • Small query range is more costly
Park et al. [7]	• Hot data identification • Different Bloom Filter has weight and recency coverage • Bloom Filter captures finer-grained recency information • Scheme takes block-level decision • Existence of an address in a Bloom Filter indicates the frequency of the Bloom Filter • Hot data are present in all Bloom Filters • Determination of hot data depend on the frequency and weight of the Bloom Filter • Solves the decay issue by flushing the Bloom Filter	• Maintains multiple Bloom Filters • Maximum frequency depends on the highest number of Bloom Filters maintained by the scheme • Implements standard Bloom Filter
FASBF [8]	• Deduplication in secondary storage • Reduces storage space • Partial FASBF is kept in RAM and whole FASBF is stored in SSD • Ratio to partition the Bloom Filter should be less to occupy less memory in RAM • PSBFA helps in occupying less RAM while keeping a reference of the fingerprint stored in SSD	• If the ratio to partition the Bloom Filter is small then the FPP of the Bloom Filter in RAM is more • Implements standard Bloom Filter
Ali and Saleh [11]	• Scheme for synonym multi-keyword search in encrypted data • Uses Bloofi to reduce the searching time	• Maintains multiple Bloom Filters • The number of Bloom Filters depends on the number of words in the document • Storing the whole Bloofi in RAM is difficult if a document is very long
Sun et al. [14]	• Deduplication of Web URL text data • Technique implements MapReduce • MapReduce can also be expected in a distributed system	• MBF cannot be stored in the secondary storage due to occurrence of data segmentation of MapReduce • Keeping the BF in RAM leads to loss of data in case of system shut down • Technique cannot be executed in Redis because Redis does not support two-dimensional array • MBF implements standard Bloom Filter

nodes. The data node is classified into an active node or solid node based on the presence of the Bloom Filter. Each BF-leaf node maintains at most one active node. An active node has free space and does not have a Bloom Filter. Data is inserted into the active node. When the active node is full, a Bloom Filter is constructed for the active node and converted into a solid node. Then the data is inserted into a new active node. Insertion is not permitted in solid nodes. However, when lots of data are deleted from the node, then the Bloom Filter is discarded, and a new Bloom Filter is constructed. When all solid nodes become full, then the BF-leaf node is split. During searching of the BloomTree, the searching procedure changes based on the type of node. First, the root node is searched if the node is BF-leaf or OF-leaf node, then the BF-leaf search algorithm or OF-leaf search algorithm follows, respectively. In the BF-leaf search algorithm, first, the solid nodes are searched by querying their respective Bloom Filter. If Bloom Filter returns $True$, then the algorithm returns the solid node. Otherwise, it searches for another solid node. If such a node is not found after searching all solid nodes, then the algorithm searches for the active node. If found, it returns the active node. In the OF-leaf search, all nodes are searched for the data. During insertion, first, the root is searched, then the nodes are traversed to find the correct leaf node. In the insertion of a leaf node, the insertion algorithm changes based on the node type. If the node is a normal node, the data is inserted into the leaf. If the leaf is full, then a new node is constructed, and data is inserted. If the node is an OF-leaf node, then the number of nodes is determined. If the number of nodes is less than 3, then the nodes are searched to determine whether the node is full or not. If no empty node is found, then a new node is constructed to insert the data. If the number of nodes exceeds 3, then the node is converted into a BF-leaf node. In case of insertion in a BF-leaf node, the data is inserted into the active node. If there is no active node, then a new active node is selected. When an active node becomes full, then it is converted into a solid node. When the leaf-head node in a BF-leaf node becomes full, then the node is split. The data in data nodes are sorted and inserted into normal nodes. Each data node is converted into a normal node. All normal nodes point to the parent of the BF-leaf node. If the parent node is also full, it will be split following the procedure similar to the B+ tree. During a delete operation, the BloomTree is traversed to find the correct leaf node. If the leaf node is a normal or OF-leaf node, then the data is deleted from the record. In case the node is a BF-leaf node, and the data is in an active node, then the record is deleted. When the required node is a solid node, the record is deleted from the data node. But a delete operation is not permitted in the Bloom Filter. Hence, when the number of deletions in a solid node crosses a threshold, the Bloom Filter is reconstructed without inserting deleted data.

Park et al. [7] proposed a Bloom Filter-based hot data identification scheme. The scheme maintains V Bloom Filters. When a written request is issued, the logical block address (LBA) is hashed by k hash functions. These hash functions provide the bit locations in the Bloom Filter, which are set to 1. When the next write request is issued, the next Bloom Filter is selected for insertion of the requested LBA. The insertion of LBA is performed on Bloom Filter in a round-robin fashion. In case the current Bloom Filter returns $True$ for the requested LBA, the next Bloom Filter is checked. The Bloom Filters are checked in round-robin fashion till a Bloom Filter is found, which has not inserted the LBA. Otherwise, if all Bloom Filter return $True$, then that LBA is concluded as hot data. The frequency of an address is determined by querying all Bloom Filters. The number of Bloom Filters having the address indicates the frequency of the address. The scheme solves the decay issue by flushing the Bloom Filter. After a certain number of write operations, a Bloom Filter which is not accessed for a long time is selected and flushed, i.e., all cells are reset to 0. The scheme uses the number of write operations as the time interval to flush a Bloom Filter. In the next time interval, the next Bloom Filter is selected in a right cyclic shift manner and is flushed. The hot data is determined by using both the frequency and weight. A weight is assigned to the Bloom Filter based on the recency. A flushed Bloom Filter saves the recent data; hence, it is assigned the highest weight. Then the next Bloom Filter in the right cyclic order has less weight. Subsequently, the weight to the Bloom Filters is assigned. For an LBA, combining both the frequency and weight is called a hot data index. If the hot data index is more than a threshold value, then the data is hot data.

Flash Assisted Segmented Bloom Filter for Deduplication (FASBF) [8] is a technique to store the fingerprint of the file in the Bloom Filter for faster access in RAM. The technique is designed for network backup systems. After receiving backup jobs from the clients, the backup server separates the metadata of the files, which are forwarded to the metadata server. The Storage Proxies executes the Rabin fingerprinting algorithm to separate the files received from the backup server. The fingerprint is generated for both file and chunk. The fingerprint is stored in the Hash Bucket Matrix (HBM). HBM is large; hence, the whole HBM is stored in SSD, whereas a small part is kept in RAM. FASBF implements Bloom Filter in two stages, RAM and SSD. The first stage is implemented in RAM, whereas the second stage is implemented in SSD. Initially, a Bloom Filter is constructed in RAM. When a fingerprint is received, it is first queried in the Bloom Filter to check for its presence. If the Bloom Filter returns $True$, then the fingerprint is ignored. Otherwise, the fingerprint is inserted into the Bloom Filter. When the capacity of the Bloom Filter crosses a

threshold, the Bloom Filter is saved in the SSD. Then a new Bloom Filter is constructed in RAM. In the second phase, a ratio is calculated. A part equal to the ratio of the Bloom Filter is kept in RAM, which is called Partial Segmented Bloom Filter Array (PSBFA), and the whole Bloom Filter is written to the SSD. During the fingerprint query, first, it is checked in the Bloom Filter. If absent, then the algorithm checks in PSBFA. If PSBFA returns $True$, then the whole FASBF is copied to RAM. If Bloom Filter in RAM or PSBFA returns $True$, then the fingerprint is duplicate; hence the fingerprint is ignored. Different FASBFs are maintained for files and chunks.

Lv et al. [9] presented an algorithm to identify the persistent item in the Big data stream using Invertible Bloom Lookup Table (IBLT) [10] and Bloom Filter. The algorithm has two stages, record and identify. The record stage uses IBLT to collect and store the incoming data. There is a normal Bloom Filter ($norBF$) and a Bloom Filter to check the attendance ($attBF$). When data is received, it is first checked in the $norBF$. If $norBF$ returns $False$, then the data is inserted into $attBF$, $norBF$, and IBLT. When a data stream ends, IBLT and $attBF$ are stored in secondary storage. In the identity phase, the data that appears in every data stream is identified. All the Bloom Filter vectors stored in secondary storage are compared. If all vectors have 1 at the same bit locations, then another Bloom Filter is constructed by setting that location to 1. All data streams are queried to the constructed Bloom Filter to identify the persistent data. When a data Bloom Filter returns a false positive response, then the data is not stored in IBLT; hence, it cannot be persistent data.

Ali and Saleh [11] proposed a synonym multi-keyword search in encrypted data using a secure index. The secure indexes are stored in hierarchical Bloom Filters. The scheme implements Bloofi [12] and the multi-keyword search algorithm follows the Wang et al. [13] technique. In the scheme, the word is scanned word by word, then the synonyms of every word are determined. Each word is converted to its bigram vector. Each element of the bigram vector is inserted into Bloofi. The LSH Euclidean hash function hashes each element. The hashed value gives the location of the Bloom Filter vector, which is set to 1. A Bloofi is constructed for a document. A Bloom Filter is constructed for each word where the elements of the bigram vector are inserted into the Bloom Filter. Following the procedure, leaf Bloom Filters are constructed. Then internal nodes are constructed by performing bitwise OR operation on the child nodes. This procedure is followed to construct the Bloofi. Bloofi is encrypted by separating every Bloom Filter into two vectors. The vectors are encrypted by transposing the two vectors. During searching, the query has two encrypted vectors. These vectors are searched in the root; if the root returns $True$, then the algorithm searches Bloom Filter children. This procedure is followed by searching the encrypted Bloofi. When the leaf node is reached, the two query vectors are multiplied by the leaf node. The resultant vector is returned as output.

Sun et al. [14] presents a Bloom Filter-based technique for deduplication of Web URL text data. The technique implements a multidimensional Bloom Filter (MBF) consisting of i standard Bloom Filters. The k hash functions are classified into i groups. Each group is assigned to a Bloom Filter. The URL has two parts, text and fingerprint. The fingerprint is inserted into the Bloom Filter. The URL list is partitioned into i groups. Each sub-URL list is assigned to a datanode in the Hadoop Distributed File System. The technique follows MapReduce to deduplicate the URLs. In the Map phase, a sub-URL list is hashed by a group of hash functions, and the hashed value is used to set a bit to 1 in its corresponding Bloom Filter. An MBF is constructed for a sub-URL list. In the Reduce phase, all MBFs are merged into a single MBF.

15.3 Database

Big data generated by various sources such as banking, businesses, e-science, etc., cannot be handled by the traditional relational database management systems (RDBMS). Hence, researchers moved towards a new development of database architectures and technologies [15] such as NoSQL. NoSQL database is distributed, open-source, non-relational, and horizontally scalable [16]. There are three types of NoSQL databases: key-value, column-oriented, and document stores. Key-value databases use keys for data storage and retrieval, where the data are stored in a schema-less way. The key is the identifier of the data. A key-value store is a distributed persistent associative array. It helps in quick access of unstructured data but is slow in query and update operation. Some examples are Redis [17] and SimpleDB [18]. In the column-oriented database, the column is a key–value pair. The key is a qualifier where the value is related to the qualifier. A row has an arbitrary number of columns with a sortable row key. Columns are clustered into column families. This database provides quick access to semistructured and structured data. Some examples are Cassandra [19] and HBase [20]. In document store databases, the data are stored in the form of documents. Documents contain various key-value pairs, key-array pairs, or even hierarchically nested document parts (commonly in JSON style). Some examples are CouchDB [21] and MongoDB [22]. This section provides a review of

some techniques and schemes to handle Big data in the database using Bloom Filter. Table 15.2 highlights various features and limitations of the Bloom Filter-based database techniques.

TABLE 15.2 Features and Limitations of Database Techniques.

Technique	Features	Limitations
Becher et al. [23]	• Bloom Filter speeds up the join operations • Bloom Filter module is the data reduction module • Modules are reconfigured to change the type of query • Different Bloom Filters work on different attributes to speed up the join operation • Bloom Filter filters the non-matching keys • Grouping of multiple Bloom Filters reduces FPP	• False positive responses are costly • False positive response requires checking with all possible keys • Implements standard Bloom Filter • The number of Bloom Filters depends on the number of tables involved in join operation of the table
Doniparthi et al. [24]	• Vertically-scalable framework • Bloom Filter reduces the interactive response time of Boolean containment queries • Capable of handling Big data • Bloom Filter remembers the query result to reuse the result later • Large segment size is considered to reduce the number of Bloom Filters	• Maintains multiple Bloom Filters • A large-sized segment requires a big-sized Bloom Filter, but a big-sized Bloom Filter has high FPP • Delete operation on the database results in reconstruction of the Bloom Filter
FBF [25]	• Bloom Filter keeps record of past elements • Dynamically adjusts the number of Bloom Filters and refreshes time period • Tries to maintain low overhead • FBF avoids duplicate counter updates • Increasing the number of Bloom Filters in FBF decreases the FPP of FBF	• Maintains many Bloom Filters • Maintains a large number of Bloom Filters during pick insertion time • If the database has many counter columns, the system has to maintain many Bloom Filters • Implements standard Bloom Filter
Goyal et al. [26]	• Validates the data transfer between relational and NoSQL databases • Does not depend on the schema mapping technique of the database • Uses relational database to map the data of NoSQL database	• False positive response leads to sequential searching of the database to determine the corrupted cell • Data of the large database cannot be completely inserted into a single Bloom Filter
EAODBT [27]	• Framework verifies the entirety and exactness of the database result • Reduces the communication cost • Decryption is performed by the client rather than the information proprietor • Avoids transmission of huge volume of data	• Transmits CBF, which requires huge memory compared to standard Bloom Filter • CBF has high false positive probability • Least counter value among various counters of CBF is considered, but it may provide incorrect value
Amirishetty et al. [28]	• Bloom Filter reduces the database recovery time • Bloom Filter eliminates the requirement of serialization and deserialization process on the sender and receiver side, respectively • Bloom Filter helps in representing the information in compact form • Bloom Filter is broadcasted to all existing instances	• A Bloom Filter is constructed for each recovery set • Implements standard Bloom Filter

Becher et al. [23] presents a data preprocessing approach in Field Programmable Gate Arrays (FPGA) hardware to minimize unwanted data from external memory. It implements Bloom Filter to speed up the join operations. Data received from the external memory is forwarded to the FPGA-based filter chain. The filter chain has three types of modules: a restriction module (RES) and ALU, Bloom Filter, and alignment. The RES and ALU evaluate the comparison and arithmetic expressions in the *where* clauses. Bloom Filter module is responsible for pre-filtering and calculating the hash value of the data for join operation. The alignment module modifies the filtered data to represent in the proper format for processor access. During join operation, the keys of a table are inserted into a Bloom Filter. Then the keys of the other table are queried to the Bloom Filter. If Bloom Filter returns *True*, then the algorithm performs the join operation. Otherwise, it queries the next key. Multiple Bloom Filters are grouped to reduce the FPP. This increases the overall size of the Bloom Filter. The last few bits of the hash value indicate the Bloom Filter among the group which will handle the data. FPGA stores the Bloom Filter vector in Block Random Access Memory (BRAM).

Doniparthi et al. [24] presented a vertically-scalable framework. Bloom Filter indices reduce the interactive response times during the evaluation of Boolean containment queries. Using the segmentation process, the data tuples in relation to a database are separated into non-overlapping subsets using a monotonically increasing primary tuple identifier. During a query operation, the results are stored in the Bloom Filters. The same query is performed instead of performing the operation on the database. The key identifier is checked in the Bloom Filter. Different Bloom Filters are constructed for different query results. In case a query is a combination of two or more query operations. Then the Bloom Filters of the results are merged, and the key identifier is queried to Bloom Filter to determine the final result. When the result is substantial, then the resultant relation is segmented. A Bloom Filter is constructed for every segment. The tuple identifier is inserted into the Bloom Filter. The Bloom Filter is stored in secondary storage. The technique should consider the status (keep or delete) of the Bloom Filter later. The Bloom Filters which become obsolete need to be removed. Otherwise, they will keep the memory occupied.

Forgetful Bloom Filter (FBF) [25] is a Bloom Filter to store the old data in key-value stores. FBF consists of three types of Bloom Filters: future Bloom Filter, present Bloom Filter, and past Bloom Filter. N-FBF means FBF has a single present and future Bloom Filter, whereas N is the number of past Bloom Filters. All the Bloom Filters have the same size and hash functions. During insertion, the element is checked in any Bloom Filter. Among the Bloom Filters, if no Bloom Filter returns $True$, then the element is inserted into the present and future Bloom Filters. After a fixed time interval, a refresh operation is performed on the FBF. In the refresh operation, the future Bloom Filter is considered as the present Bloom Filter. The current Bloom Filter is included in the past Bloom Filters, deletes the oldest Bloom Filter, and constructs a new Bloom Filter, which is considered as future Bloom Filter. During searching for an element in FBF, first, the algorithm checks the future Bloom Filter. If it returns $False$, then the algorithm checks the present and newest past Bloom Filter simultaneously. If both Bloom Filters return $False$, the algorithm searches the next two past Bloom Filters. When all Bloom Filters return $False$, the element is absent from FBF. A thread is implemented in the background, which continuously calculates the FPP of FBF. When the insertion rate increases and FPP crosses the permitted limit, the number of Bloom Filters in FBF increases from $(N + 2)$ to $2(N + 2)$ and the refresh time period decreases. Similarly, when the number of insertions decreases, the number of Bloom Filters decreases and the refresh time period increases. In the database, an FBF is attached to each counter column. When a counter update is received, first, it is queried to FBF. If FBF returns $True$, then it means that the request is performed earlier, the request is appended to the commitlog, and the Memtable is updated. If FBF returns $False$, then the algorithm performs the counter operation. The counters are transmitted to other replicate nodes to keep their FBFs synchronized.

Goyal et al. [26] presented a technique for data validation between cross-platform databases based on Bloom Filter and denormalized schema structures. The technique first retrieves the metadata from the relational databases to construct an identical mapping for the NoSQL database. The required tables of the relational database are joined. The metadata obtained from the joined tables is used to retrieve data from the NoSQL database. Both databases are transformed into a similar format and given as input to the validation engine. The validation engine has two parts, Bloom Filter and cell validation engine. The primary key of the source database is inserted into the Bloom Filter. The primary key of the target data is queried to Bloom Filter. If Bloom Filter returns $False$, then the data is forwarded to the cell validation engine to determine the corrupted cell. If Bloom Filter returns $True$, then the data is considered extra-record, or the data is checked in the target database sequentially to determine the corrupted cell.

Efficient Auditing for Outsourced Database with Token Enforced Cloud Storage (EAODBT) [27] scheme is based on Bloom Filter to verify the completeness and exactness of the search result. The scheme has four entities: a Group of customers, Information proprietor (IP), Cloud Service Provider (CSP), and Arbitration Center (AC). Initially, IP deploys the database to CSP. IP constructs related to verification composition to verify the reliability of the deployed database. When a client wants to access the database, it sends a token to IP, which consists of a client ID and a query request. IP determines whether to forward the token to the CSP or discard it. The scheme discards the request from an unreliable client. The IP encodes tuples individually and also constructs CBF by inserting the tuples. IP forwards the query request and CBF to CSP. IP also constructs IBF by inserting the tuples and forwards to AC. After receiving the query from IP, CSR checks the query tuples with the database. If the tuples are equal and present in CBF, it returns the result in encrypted form to the client. In case no matching tuple is found, the algorithm forwards only the CBF to the client as confirmation. After successful authentication, the client is able to decrypt the database result using the key forwarded by the IP. An empty CBF verifies a null tuple. Otherwise, conflict tuples are forwarded to AC, which determines whether to accept the result or not. AC uses the IBF to verify. The client determines the entirety of the result by hashing the tuple. The hash value provides the bit location of IBF. If the least counter among the bit locations is equal to the number of accepted tuples, the client accepts the result. Otherwise, the conflicted index is

forwarded to AC. The client checks the counter value of IBF. If the counter value is equal to the number of tuples, then it confirms the exactness of the result, and the client accepts the result.

Amirishetty et al. [28] proposed the use of a buddy instance to reduce the instance recovery time. The scheme uses Bloom Filter in buddy instances to enhance availability. Instance recovery has two phases, cache recovery and transaction recovery. The cache recovery is reapplying all required changes to files' data blocks as per the redo log's changes. Transaction recovery performs rollback of uncommitted changes in data blocks as per the changes recorded in the undo segment. During the beginning of the claim stage, the recovery sets are forwarded to all existing instances in the cluster. However, the recovery sets are large data structures that cannot be transmitted using internode messaging. Instead of this set, the scheme implements Bloom Filter to save the block information. During the claim stage, it is compulsory to acquire locks on the data block that require recovery to prohibit any changes to these data blocks. In case the latest version of a data block is present in the cache rather than the disk, the recovery has to wait for the data block transfer to the disk from the node. Hence, at the beginning of the claim stage, the information of all the data blocks that do not require recovery are transmitted to the existing nodes. This helps the nodes to determine the data block that needs to be avoided till the data block is recovered. This information is inserted into the Bloom Filter, which is constructed during the construction of the recovery set.

15.3.1 Privacy-preserving record linkage

Currently, there is a requirement for linking the databases belonging to different organizations, companies, or institutions. The purpose is to enhance the data quality and integration of more data while linking more data relations. Some application areas are government services, crime and fraud detection, healthcare, and business applications. One example of the importance of data linkage in healthcare is epidemiological studies [29]. The linkage of various healthcare organizations helps in the detection of infectious diseases. The information also helps in the development of health policies in a more effective and efficient way.

Privacy is the main issue that needs to be addressed in such database linkage. To address this issue, privacy-preserving record linkage (PPRL) techniques are developed [30]. Their goal is to determine the matching records having the same entities across various databases while maintaining the privacy and confidentiality of these entities. Before linkage, the database owners agree on disclosing some attributes. These attributes are known as quasi-identifiers (QIDs). QIDs are common attributes in all databases that are used for linkage. Some examples are name, date of birth, address, and phone number. However, QIDs are usually the privacy information of an individual. Hence, PPRL techniques have to resolve the privacy issue. This section presents a review of PPRL techniques based on Bloom Filter. Table 15.2 highlights various features and limitations of the Bloom Filter-based PPRL techniques. See Table 15.3.

TABLE 15.3 Features and Limitations of Privacy-Preserving Record Linkage Techniques.

Technique	Features	Limitations
Vaiwsri et al. [31]	• Implements Bloom Filter encoding to resolve the missing values in privacy-preserving record linkage • Bloom Filter helps in hiding sensitive data • SSI prevents reidentification of sensitive attribute values • Applies different permutations on the bit positions of the Bloom Filters to enhance privacy • In batch linkage method, all partitions are combined into a single encoded database by each database owner and forwarded to linkage unit • Batch linkage method requires single communication between database owners and linkage unit	• Bloom Filter is constructed for every record having no missing value for the agreed parameter • Iterative method requires multiple rounds of communication between the database owners and linkage unit • Implements standard Bloom Filter
Vatsalan et al. [30]	• Encoding method of sensitive data to enhance privacy • CBF-based communication pattern reduces the number of record comparisons • Enhances scalability by distributing comparison operations among parties • CBF is generated in the linkage unit • CBF prevents inference attack • CBF prevents storage of many Bloom Filters in linkage unit	• A Bloom Filter is constructed for each record • Implements standard Bloom Filter

Vaiwsri et al. [31] presented a scheme based on Bloom Filter encoding to resolve the missing values in PPRL. The database owners want to avoid sharing sensitive data with other databases. First, the database owners agree on the parameter which will be used for linkage. Both database owners construct attribute-level Bloom Filters (ABFs). An ABF is constructed for every record having no missing value for the agreed parameter. An ABF is generated for an item by inserting the q-gram of the item into the Bloom Filter. The database owners construct a table consisting of only missing value records. Using the table, both databases generate a local lattice structure, which helps in determining the missingness patterns in the database. The database owners follow the secure set intersection (SSI) [32] protocol to determine the common missingness patterns between the two databases. SSI generates a common lattice. Based on the common lattice, the database owners identify the records with missing patterns. The records are grouped into partitions. The bits are selected from the ABFs of the record that belongs to that partition using the missingness pattern of a partition. A Bloom Filter is constructed for each record. Then the algorithm applies different permutations on the bit positions of the Bloom Filters to enhance privacy. It then groups the Bloom Filters of neighboring partitions based on the local and common lattice structures. The grouping is performed either in the lattice's upward, downward, or sidewise partitions. The scheme has a linkage unit that performs the link between databases. The database owners forward the Bloom Filters of a partition to the linkage unit. The linkage unit calculates the Dice coefficient similarity between the Bloom Filters. A blocking technique [33] is followed to enhance the comparison. The linkage unit returns the comparison result, which is determined by the Bloom Filter similarities and a similarity threshold. The result is forwarded to both the database owners. The similar records are removed from the next partition to reduce redundancy. The comparison continues till processing of all partitions.

Vatsalan et al. [30] proposeed a CBF-based encoding method of sensitive data to enhance privacy for multi-party PPRL (MPPPRL). First, all parties agree on the value of various parameters: Bloom Filter size, k, gram length, minimum similarity threshold, private blocking function, blocking keys, and set of attributes. The gram length is used for Bloom Filter encoding. The minimum similarity threshold helps in determining similar records. The blocking keys are used for blocking. The set of attributes is sensitive attributes used for database linkage. Every party then applies the private blocking function to decrease the number of records. The q-gram of the attributes of these filtered records is inserted into a Bloom Filter. A Bloom Filter is constructed for each record. The linkage unit constructs a random Bloom Filter (R) and forwards it to a party. The party adds its Bloom Filter value to R and forwards it to another party. This process is continued till all parties receive R. Using the R vector, the linkage unit generates a CBF, which is constructed for each set of candidate attributes. The Dice coefficient similarity between the CBFs is calculated. The records are clustered based on the minimum similarity threshold. The security is further enhanced by applying the homomorphic-based secure summation (HSS) scheme. HSS has a set of public and private keys for encryption and decryption of Bloom Filter. The public key is available to all parties, whereas the private key is with the linkage unit. The parties receive the encrypted R vector to which they add their encrypted Bloom Filter. Finally, the linkage unit decrypts the R vector.

15.4 MapReduce

MapReduce [34] is a data processing framework for processing Big data across thousands of servers in a Hadoop cluster. It has two main components, map and reduce. The map is responsible for receiving input data and applying the map function to convert the data into a key–value pair. Shuffle redistributes the data and forwards it to various reducer nodes. It ensures that the same key data are forwarded to the same reducer node. Reduce nodes are responsible for data processing based on the application. The reducer nodes work in parallel. This section includes a review of Bloom Filter-based techniques to improve the performance of the MapReduce framework. Table 15.2 presents various features and limitations of the Bloom Filter-based MapReduce techniques. See Table 15.4.

Anandkrishna and Kumar [35] proposed a technique of implementing Bloom Filter to speed up the MapReduce process. During the initial run, the dataset is inserted into the Mapper, which generates key–value pairs for each data item. The key–value pairs are forwarded to different Reducers. Each Reducer has a Bloom Filter. The data are inserted into the Bloom Filter. During an iterative run, the data received by the Reducer is checked in the Bloom Filter. If Bloom Filter returns $True$, then the data is modified. The Reducer process is executed for every modified data.

Tan et al. [36] presented a Bloom Filter-based approach to filter the data segment at the map phase in MapReduce. The input records are partitioned into buckets containing the same number of records. A Bloom Filter is constructed for each bucket. If the records are absent from the Bloom Filter, then they are inserted into the Bloom Filter. Otherwise,

TABLE 15.4 Features and Limitations of MapReduce Techniques.

Technique	Features	Limitations
Anandkrishna and Kumar [35]	• Bloom Filter speeds up the MapReduce process • Bloom Filter is constructed by every Reducer • Bloom Filter is constructed in the initial run and used in an iterative run	• Bloom Filter can be implemented in Mapper to reduce the forwarding of the data to Reducer • Implements standard Bloom Filter
Tan et al. [36]	• Filters the data segment at the map phase • Maps a multidimensional data into one-dimension index • Lightweight	• More mappers increase the running time of the input record • Implements standard Bloom Filter
Yue et al. [37]	• Reduces the overhead of join operation • Reduces the data transfer traffic • Reduces processing overhead	• FPP of the standard Bloom Filter is reduced by introducing another Bloom Filter • Efficiency is low in small-sized database due to overhead increment by Bloom Filter construction

the duplicate records are ignored. Bloom Filter helps in removing duplicate records. The bucket consisting of unique records is forwarded to the mapper. In the case of multidimensional data, all the attributes are taken as a single string and hashed by a hash function. The hashed value is used to set a bit to 1 in the Bloom Filter.

Yue et al. [37] presented a technique to reduce the overhead of join operation using Bloom Filter. The technique constructs an extended Bloom Filter, which consists of two Bloom Filters. The first is the standard Bloom Filter, and the second is constructed by performing an XOR operation among the hashed values. The hashed values are the values obtained by the k hash functions. The result of the XOR operation provides the bit location of the second Bloom Filter, which is set to 1. Each mapper constructs an extended Bloom Filter for its data. Using the extended Bloom Filter, the mapper reduces the data. The reduced data is forwarded to the reducer. The reducer performs OR operation on the extended Bloom Filters to merge the results. An OR operation is performed separately on all standard Bloom Filters and XOR Bloom Filters. One database constructs the extended Bloom Filter in the join operation, which is forwarded to another database that uses the extended Bloom Filter to filter the record that does not match the relation.

15.5 Discussion

Big data is not only problematic just due to its colossal volume but also due to the complexity involved in its generation, collection, processing, and storage. Lots of resources are wasted for handling meaningless or duplicate data. Handling duplicate data is simple to some extent compared to determining the meaningfulness of the data. The duplicate data is handled by implementing an efficient filtering data structure or technique. For this purpose, Bloom Filter is the best choice for deduplication. Bloom Filter is a simple data structure and requires low memory. Hence, deploying Bloom Filter in different data flow units is not much of an overhead. During data streaming, the speed of data is high; therefore, an efficient filtering data structure or technique is required, which is able to keep the data processing pace with the incoming data speed. Bloom Filter takes constant time for each operation. Thus, Bloom Filter can keep pace with the Big data generation or collection units.

Patgiri and Ahmed [2] included vacuum as one of the characteristics of volume in Big data. Vacuum refers to the requirement of maintaining huge free memory for the incoming future data. The incoming data speed is difficult to predict. Data becomes hot data for a short interval such as 1 or 2 hours in a day, likewise it can remain hot for years. For example, COVID-19 data will become hot data after 2020 and remain so for a few more years. Therefore, the farmhouse needs to maintain huge free memory space to avoid data loss. Because high data incoming speed may not provide time for saving the data in a buffer, in such a scenario, filtering data is more critical because memory is not infinite. Infinite memory is a virtual concept that is made possible by procuring new hardware to fill up the memory deficiency. Thus, implementing Bloom Filter will help in saving lots of memory space.

A database maintains many tables based on various relationships. The tables are further subdivided to normalize the tables. These tables are joined based on the requirement. But these join operations are costly because every row is matched with every row of another table. In this case, Bloom Filter can be used for quick matching of rows. The main issue in PPRL is privacy. For database linkage, the database owners have to share the attribute values with other databases belonging to another organization. These attributes are sensitive information, but these need to be

disclosed for multi-party join operations. Bloom Filter maps the items into a vector. Bloom Filter is incapable of providing original data. So, it cannot be used for data retrieval. However, this feature makes it the best choice for PPRL. The QIDs are inserted into the Bloom Filter and shared with other parties. Bloom Filter is used by other parties for attribute matching. The matching operation is faster in Bloom Filter as the query operation takes constant time. Thus, the trade-off between the sharing of QIDs and execution of query operations incurs very low overhead.

MapReduce framework was developed by Hadoop to handle Big data processing in a distributed system. The input data are collected and distributed to various nodes for processing. MapReduce has three main steps: map, shuffle, and reduce. All these steps have to deal with a huge volume of data. Moreover, these steps generate a huge volume of intermediate data. Therefore, Bloom Filter can be implemented in these steps to remove duplicate data and enhance the performance.

Table 15.5 highlights a comparison of the techniques mentioned in the chapter based on various parameters. Some observations referring to Table 15.5 are that, except for a few, every technique has implemented a standard Bloom Filter. Implementation of a standard Bloom Filter is less efficient compared to some other Bloom Filter variants. Furthermore, the techniques have not implemented any additional procedure to reduce the FPP. Deploying multiple Bloom Filters is costly; however, in Big data, Bloom Filters should be deployed at appropriate points for removal of duplicate or irrelevant data to reduce the burden on data processing units, for instance, MapReduce.

TABLE 15.5 Comparison of Wireless Communication Techniques: Variant, Bloom Filter variant; Reduce FP, technique used to reduce the number of false positives; Purpose, purpose of the technique.

Technique	Variant	Reduce FP	Purpose	Application
Becher et al. [23]	Standard	Grouping of multiple Bloom Filters	Key matching	Database
Doniparthi et al. [24]	Standard	No	Relational	Query Result storage
FBF [25]	Standard	No	NoSQL	Counter update Filtering
Goyal et al. [26]	Standard	Sequentially search the database	Relational, NoSQL	Data validation between cross platform databases
EAODBT [27]	CBF, Inverted	No	Relational	Verification of completeness and exactness
Amirishetty et al. [28]	Standard	No	Relational	Database Recovery
Vaiwsri et al. [31]	Attribute-level	No	Missing values in the database	Privacy-preserving record linkage
Vatsalan et al. [30]	Standard, CBF	No	Privacy	Privacy-preserving record linkage
Lv et al. [9]	Standard	No	Identifies persistent data	Data Stream
MDSBF [4]	Standard, CBF	No	Range query searching	Networking
Jang et al. [5]	Standard	No	Deduplication	Networking
BloomTree [6]	Standard	No	Reducing read write operation	Solid-state drives
Park et al. [7]	Standard	No	Determination of hot data	Hard drive
FASBF [8]	Standard	No	Deduplication	Hard drive
Ali and Saleh [11]	Bloofi	No	Synonym multi-keyword search	Cloud Computing
Sun et al. [14]	Multidimensional	No	Deduplication	Web URL text data
Anandkrishna and Kumar [35]	Standard	No	Filtering in Reducer	MapReduce
Tan et al. [36]	Standard	No	Mapper Filtering	MapReduce
Yueet al. [37]	Extended	Yes	Join Operation	MapReduce

15.6 Conclusion

This chapter discussed the solutions Bloom Filter provides for enhancing Big data performance. The chapter on Big data included presentation of data management, database, privacy-preserving record linkage, and MapReduce. The chapter also reviewed many techniques based on the Bloom Filter proposed for Big data. Filtering the redundant data provides big support in data processing, analysis, and storage of Big data. These tasks have to invest lots of resources to handle Big data. Furthermore, these tasks also generate intermediate data, which adds to the volume

of Big data. Therefore, Bloom Filter provides a great solution to easily handle these Big data issues while incurring lower overhead.

References

[1] D. Laney, et al., 3D data management: controlling data volume, velocity and variety, META Group Res. Note 6 (70) (2001) 1.
[2] R. Patgiri, A. Ahmed, Big data: the V's of the game changer paradigm, in: 2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS), 2016, pp. 17–24.
[3] Total data volume worldwide 2010–2025 – Statista, https://www.statista.com/statistics/871513/worldwide-data-created/#statisticContainer. (Accessed 2 March 2022).
[4] Y. Hua, D. Feng, T. Xie, Multi-dimensional range query for data management using Bloom filters, in: 2007 IEEE International Conference on Cluster Computing, 2007, pp. 428–433.
[5] Y.-H. Jang, N.-U. Lee, H.-J. Kim, S.-C. Park, Design and implementation of a Bloom filter-based data deduplication algorithm for efficient data management, J. Ambient Intell. Humaniz. Comput. (2018) 1–7.
[6] P. Jin, C. Yang, C.S. Jensen, P. Yang, L. Yue, Read/write-optimized tree indexing for solid-state drives, VLDB J. 25 (5) (2016) 695–717.
[7] D. Park, W. He, D.H. Du, Hot data identification with multiple Bloom filters: block-level decision vs I/O request-level decision, J. Comput. Sci. Technol. 33 (1) (2018) 79–97.
[8] G. Dagnaw, A. Teferi, E. Berhan, Flash assisted segmented Bloom filter for deduplication, in: Afro-European Conference for Industrial Advancement, Springer, 2015, pp. 87–98.
[9] Z. Lv, F. He, L. Chen, Finding persistent items using invertible Bloom lookup table, in: 2019 International Conference on Computer, Information and Telecommunication Systems (CITS), 2019, pp. 1–5.
[10] M.T. Goodrich, M. Mitzenmacher, Invertible Bloom lookup tables, in: 2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton), 2011, pp. 792–799.
[11] A.A. Ali, S. Saleh, Synonym multi-keyword search over encrypted data using hierarchical Bloom filters index, in: Machine Learning and Big Data Analytics Paradigms: Analysis, Applications and Challenges, Springer, 2021, pp. 521–545.
[12] A. Crainiceanu, Bloofi: a hierarchical Bloom filter index with applications to distributed data provenance, in: Proceedings of the 2nd International Workshop on Cloud Intelligence, 2013, pp. 1–8.
[13] B. Wang, S. Yu, W. Lou, Y.T. Hou, Privacy-preserving multi-keyword fuzzy search over encrypted data in the cloud, in: IEEE INFOCOM 2014-IEEE Conference on Computer Communications, IEEE, 2014, pp. 2112–2120.
[14] S. Sun, J. Gong, A.Y. Zomaya, A. Wu, A distributed incremental information acquisition model for large-scale text data, Clust. Comput. 22 (1) (2019) 2383–2394.
[15] J. Pokorný, Database technologies in the world of Big data, in: Proceedings of the 16th International Conference on Computer Systems and Technologies, CompSysTech'15, Association for Computing Machinery, New York, NY, USA, 2015, pp. 1–12.
[16] Nosql databases list by hosting data – updated 2022, https://hostingdata.co.uk/nosql-database/. (Accessed 27 February 2022).
[17] Redis, https://redis.io/. (Accessed 27 January 2022).
[18] AWS – Amazon SimpleDB – Aimple database service, https://aws.amazon.com/simpledb/. (Accessed 27 January 2022).
[19] Apache cassandra — apache cassandra documentation, https://cassandra.apache.org/_/index.html. (Accessed 27 January 2022).
[20] Apache HBase – Apache HBase™ home, https://hbase.apache.org/. (Accessed 27 January 2022).
[21] Apache CouchDB, https://couchdb.apache.org/. (Accessed 27 January 2022).
[22] MongoDB atlas: Cloud document database – MongoDB, https://www.mongodb.com/cloud/atlas. (Accessed 27 January 2022).
[23] A. Becher, D. Ziener, K. Meyer-Wegener, J. Teich, A co-design approach for accelerated SQL query processing via FPGA-based data filtering, in: 2015 International Conference on Field Programmable Technology (FPT), 2015, pp. 192–195.
[24] G. Doniparthi, T. Mühlhaus, S. Deßloch, A Bloom filter-based framework for interactive exploration of large scale research data, in: J. Darmont, B. Novikov, R. Wrembel (Eds.), New Trends in Databases and Information Systems, Springer International Publishing, Cham, 2020, pp. 166–176.
[25] R. Subramanyam, I. Gupta, L.M. Leslie, W. Wang, Idempotent distributed counters using a forgetful Bloom filter, in: 2015 International Conference on Cloud and Autonomic Computing, IEEE, 2015, pp. 113–124.
[26] A. Goyal, A. Swaminathan, R. Pande, V. Attar, Cross platform (RDBMS to NoSQL) database validation tool using Bloom filter, in: 2016 International Conference on Recent Trends in Information Technology (ICRTIT), 2016, pp. 1–5.
[27] C.M. Geeta, B.N. Rashmi, R.G. Shreyas Raju, S. Raghavendra, R. Buyya, K.R. Venugopal, S.S. Iyengar, L.M. Patnaik, EAODBT: efficient auditing for outsourced database with token enforced cloud storage, in: 2019 IEEE International WIE Conference on Electrical and Computer Engineering (WIECON-ECE), 2019, pp. 1–4.
[28] A.K. Amirishetty, Y. Li, T. Yurek, M. Girkar, W. Chan, G. Ivey, V. Panteleenko, K. Wong, Improving predictable shared-disk clusters performance for database clouds, in: 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017, pp. 237–242.
[29] L.G. Álvarez, P. Aylin, J. Tian, C. King, M. Catchpole, S. Hassall, K. Whittaker-Axon, A. Holmes, Data linkage between existing healthcare databases to support hospital epidemiology, J. Hosp. Infect. 79 (3) (2011) 231–235.
[30] D. Vatsalan, P. Christen, E. Rahm, Scalable privacy-preserving linking of multiple databases using counting Bloom filters, in: 2016 IEEE 16th International Conference on Data Mining Workshops (ICDMW), 2016, pp. 882–889.
[31] S. Vaiwsri, T. Ranbaduge, P. Christen, R. Schnell, Accurate privacy-preserving record linkage for databases with missing values, Inf. Syst. 106 (2022) 101959, https://doi.org/10.1016/j.is.2021.101959.
[32] C. Dong, L. Chen, Z. Wen, When private set intersection meets big data: an efficient and scalable protocol, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS'13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 789–800.

[33] D. Karapiperis, V.S. Verykios, An LSH-based blocking approach with a homomorphic matching technique for privacy-preserving record linkage, IEEE Trans. Knowl. Data Eng. 27 (4) (2015) 909–921, https://doi.org/10.1109/TKDE.2014.2349916.
[34] What is apache MapReduce? – IBM, https://www.ibm.com/topics/mapreduce. (Accessed 1 January 2022).
[35] R. Anandkrishna, D. Kumar, Improving MapReduce for incremental processing using map data storage, in: Fourth International Conference on Recent Trends in Computer Science & Engineering (ICRTCSE 2016), Proc. Comput. Sci. 87 (2016) 288–293, https://doi.org/10.1016/j.procs.2016.05.163.
[36] Z.-l. Tan, K.-r. Zhou, H. Zhang, W. Zhou, BF-MapReduce: a Bloom filter based efficient lightweight search, in: 2015 IEEE Conference on Collaboration and Internet Computing (CIC), 2015, pp. 125–129.
[37] M. Yue, H. Gao, S. Shi, H. Wang, Join query processing in data quality management, in: H. Gao, J. Kim, Y. Sakurai (Eds.), Database Systems for Advanced Applications, Springer International Publishing, Cham, 2016, pp. 329–342.

This page intentionally left blank

CHAPTER

16

Bloom Filter in cloud computing

16.1 Introduction

Cloud computing gained popularity due to its provision of limitless services such as software, infrastructure, platform, etc. The cloud service providers provide high-quality services based on user demand. The users pay based on the consumed services. Many individuals, big companies, and institutions are availing the cloud services. Some examples of cloud service providers are Amazon Web Service (AWS) [1], Microsoft Azure [2], Google Cloud Platform [3], IBM Cloud Services [4], and Adobe Creative Cloud [5]. The cloud eliminates the maintenance cost. Buying and acquiring the latest and sophisticated technologies for a small company becomes a big affair with respect to the economy. Hence, expanding infrastructure is easy by accessing the cloud. It is just extending the amount of services and cost. Cloud is an excellent support to small companies that are incapable in all aspects to acquire, change, or maintain the latest technologies. Thus, the cloud helps in the smooth functioning of the business.

Cloud computing has many issues and challenges. Multiple users access data all around the world at the same time. Cloud needs to provide data security to protect its resources from misuse and attacks [6]. Security is also linked to the trust of data users in the cloud service provider. Efficient and optimized task scheduling is also important [7]. The cloud promises to provide a continuous high quality of services to the users. This promise is attached to the optimized utilization of resources. Optimized task scheduling is also linked to load balancing. Load balancing [8] is engaging every node with tasks equally for faster and timely completion. Another issue is data storage. Cloud appears as providing infinite memory. However, practically it is impossible. Big data also further contributes to the problem of data accessing, computing, analysis, and management.

Many cloud computing techniques are exploring Bloom Filter for easy and faster execution of operations. Bloom Filter, with its operation having constant time complexity, is of great support for cloud computing. When data owners outsource their data to the cloud to maintain privacy and security, instead of plaintext documents, encrypted documents are stored in the cloud. However, searching for documents among the encrypted documents based on the keywords is a difficult task. But storing the keywords in the cloud removes privacy. Bloom Filter is the solution for this issue. The keywords are inserted into the Bloom Filter, and along with the encrypted documents, the Bloom Filter is also stored in the cloud server. Another implementation of Bloom Filter is deduplication. Cloud servers store a huge volume of data. Redundant data further increases the complexity. In this case, Bloom Filter, with its simple architecture and fastest operation, helps in the fastest data filtering.

This chapter focuses on the role of Bloom Filter in enhancing the performance of cloud services. Bloom Filter is implemented in cloud computing for various purposes. But this chapter focuses on data indexing and searching of encrypted data and data storage and management. In this chapter, Section 16.2 highlights the role of Bloom Filter in the data indexing and searching of encrypted data. The section also includes a review of Bloom Filter-based techniques. Section 16.3 discusses the role of Bloom Filter in data storage and management. The Bloom Filter is used for deduplication and deletion of data. The section includes a review of Bloom Filter-based techniques. Section 16.4 presents a brief discussion of the role of Bloom Filter in cloud computing. We also provide some solutions for better implementation of Bloom Filter in the cloud.

This chapter uses three entities frequently: data users, data owners, and the server. The data owners are the original owners of a set of documents which are willing to share with other authorized data users. The data owners send the documents to the cloud server for easy access. To protect the data, the owners encrypt the documents and the keywords of the document. The encrypted documents and keywords are transferred to the cloud rather than the original document. The data users are the authorized users who access the outsourced data on the server. Cloud servers store the encrypted data and provide services for easy access of the data to data users.

 Copyright © 2023 Elsevier Inc. All rights reserved.

16.2 Indexing and searching of encrypted data

Usually, the cloud servers and cloud users are located in different domains. The cloud service provider has to ensure protected communication with the cloud users. It has to maintain storage, data privacy [9], and transmission security. Data protection requires data encryption. But it affixes additional complexity in data access. Thus, cloud computing has to provide efficient indexing and searching techniques for accessing encrypted data. For searching the encrypted data, Bloom Filter is constructed for the keywords associated with the document. The cloud server also has a complex problem, i.e., multi-keyword similarity searching [10]. The data user requests documents by forwarding keywords. The cloud server has to search for documents matching all or maximum keywords. Based on the matching keywords, the documents are ranked, and some top-ranked documents are forwarded to the data user. Thus, multi-keyword similarity searching is complex. This section discusses the role of the Bloom Filter in solving this problem. This section also includes a review of the techniques proposed for indexing and searching of encrypted data. Table 16.1 highlights the features and limitations of the Bloom Filter-based encrypted data indexing and searching techniques in cloud servers.

Poon and Miri [11] proposed Bloom Filter-based phrase search technique. First, the technique determines keywords associated with every document. A Bloom Filter is constructed for each document, and the keywords are inserted. Every keyword is first hashed by a private key. Then the hashed value is hashed by k hash functions. The k hashed values give the bit locations of the Bloom Filter, which are set to 1. The data owner then transmits the encrypted documents and their corresponding Bloom Filter to the cloud server. During searching, the user sends a set of keywords to the data owner. The data owner encrypts the keywords, changes the order, and sends it to the cloud server. The cloud server constructs a query Bloom Filter. The query Bloom Filter is compared with every Bloom Filter. If a Bloom Filter matches, its corresponding document is forwarded to the user.

Zhu et al. [12] presented a Bloom Filter and locality sensitive hashing (LSH) based verifiable dynamic fuzzy search of encrypted data. An index 26^2-bit long vector is constructed to store the 2-gram of keywords. Each slot of 256-bit index vector is a possible bigram. The index vector bit is set to 1 with respect to the bigrams of the keyword. First, the keywords are extracted from the document. The index vector is constructed for the keywords. The bigrams of the keywords are reflected in the index vector. The bigrams are also inserted into the Bloom Filter using LSH. Both documents and keyword indexes are encrypted and sent to the Cloud server. During searching, the keywords are sent by the user. A Bloom Filter is constructed, and the bigram of the keywords is inserted into the Bloom Filter. The Bloom Filter is hashed by the LSH functions. Then Bloom Filter is encrypted using two pseudorandom functions. The trapdoor contains both encrypted Bloom Filters, which are sent to the server. The server uses the index and the trapdoor to determine the documents. Then the documents are sent to the user.

Yu et al. [10] presented a Bloom Filter-based data searching in cloud computing. The data owner constructs a table storing the information about the documents and keywords. The first column is the file identification number, and the second column is the keywords of the file. The data owner encrypts the file IDs. A Bloom Filter is constructed for each row, i.e., one Bloom Filter for each file. Another table is constructed called the Bloom table. In the Bloom table, the first column is encrypted file IDs, and the second column is Bloom Filter. Then the data owner sends the Bloom table to the cloud server. When a data user wants to access the data, it has to confirm the authentication by the data owner. After confirmation, the data user hashes the keywords using hash functions. The hashed values are forwarded to the cloud server. The cloud server uses the hashed values and checks all the Bloom Filters. If the number of keyword matches crosses a threshold value, then that file ID is selected.

Secure Index based on Counting Bloom Filter (SICBF) [13] is a CBF-based secure indexing scheme for ranked multiple keywords search. SICBF maintains a CBF and a hash table. Each counter of CBF is associated with a bucket of the hash table. CBF is used to keep the count of duplicate data. However, a pruning algorithm is executed to delete duplicate data. SICBF deletes duplicate data but keeps the correct count of the data in CBF. If a new data is inserted, the data is hashed by k hash functions. The CBF counters are incremented. The new data is inserted in the bucket corresponding to the CBF counter having the lowest value. In the scheme, first, the data owner generates the secret key of the Advanced Encryption Standard (AES). The next step is building the Index. The keywords are extracted from the plaintext document and the relevance score is calculated. The score is encrypted by the Paillier cryptosystem. Then the inverted Index is constructed, which is also encrypted. The encrypted Index is inserted into the CBF. The encrypted documents, encrypted indexes, and CBF are stored in the cloud server. During a user request, the user sends a request using keywords. A trapdoor is constructed for the keywords and transmitted to the cloud server. After receiving the trapdoor, the encrypted Index is searched in the CBF. The identifiers and relevance scores

TABLE 16.1 Features and Limitations of Bloom Filter based Encrypted Data Indexing and Searching Techniques in Cloud Server.

Technique	Features	Limitations
Poon and Miri [11]	• Bloom Filter-based phrase search technique • Easy searching for keywords • Comparison is performed only between the Bloom Filters • Small Bloom Filter enhances security and lower storage cost	• A large Bloom Filter reduces security and increases storage cost • Small Bloom Filter has high FPP and high computational cost • Small Bloom Filter leads to searching of irrelevant documents • A Bloom Filter is constructed for every document • Cloud server maintains both encrypted documents and their corresponding Bloom Filter • Searching time is directly proportional to the total number of documents
Zhu et al. [12]	• Bloom Filter and locality sensitive hashing based verifiable dynamic fuzzy search of encrypted data • Dynamically updates the documents • Verifies the search results • Efficient fuzzy search • Does not require predefined dictionary • Due to LSH, misspelled words are hashed to same bit location in Bloom Filter	• Maintains a 256-long vector for each document along with the Bloom Filter • Bloom Filter is constructed for each document • Implements standard Bloom Filter • Searching time is directly proportional to the total number of documents
Yu et al. [10]	• Bloom Filter-based data searching in cloud computing • In Bloom table, the first column contains encrypted file IDs and the second column is Bloom Filter • Bloom table compress the information stored in cloud server • Bloom table reduces communication cost • Data user sends hash value of requested keywords • Sending hash values increases security against cloud service providers	• One Bloom Filter is constructed for each file • Maintains many Bloom Filters • Bloom Filter is a bit array, so any bit change during transmission enhances FPP of the Bloom Filter • If there are more documents then more bandwidth is required to transmit the Bloom table
SICBF [13]	• CBF-based secure indexing scheme for ranked multiple keywords search • Encrypted index is inserted into the CBF • Pruning algorithm deletes duplicate data • CBF has the correct count of the documents • Some random keywords are send to the user with the trapdoor keywords	• Paillier cryptosystem has more computational and storage overhead compared to traditional symmetric algorithm • Technique has many encryption overheads: document, index, and relevance score • Addition of random keywords increases the computational time of trapdoor algorithm • Searching time is directly proportional to the total number of documents
Song et al. [14]	• Full-text retrieval from encrypted document • Implements hierarchical Bloom Filter tree index • Bloom Filters of the index tree are linked to the encrypted document • Index tree stores three types of information: the Bloom Filters, the status (active or inactive) of the node, and links • Data owner calculates the membership value of each Bloom Filter to maintain privacy • Cloud server uses the membership values to rank the result documents	• Scheme extracts all words, but all are not relevant for document searching • Many Bloom Filters are generated for a single document • The number of Bloom Filters for a document directly depends on the number of word clusters • The number of index nodes in a tree is many times more than the number of documents • Insertion of a new document requires searching for an inactive node
Umer et al. [15]	• Encryption data searching technique where the encrypted index is compressed to reduce communication cost • Implements a sliding window Bloom Filter • Bloom Filter is encrypted by the Paillier algorithm • A polynomial is sent to the data user to reduce communication cost • Value returned to the data user is independent of the Bloom Filter	• A Bloom Filter is constructed for each document • Maintains many Bloom Filters • Implements standard Bloom Filter

continued on next page

TABLE 16.1 (*continued*)

Technique	Features	Limitations
OS2 [17]	• OS2 technique is for searching encrypted data in cloud servers • Bloom Filter reduces the number of comparison between the outsourced data and the search request • Bloom Filter reduces the memory space requirements • Bloom Filter size and the hash functions are decided by the data owner and authorized data users • Each bit of Bloom Filter is encrypted to increase privacy • Encrypting each bit of Bloom Filter makes it difficult to determine the bit as 1 or 0 • Sliding window reduces the number of data inserted into the Bloom Filter	• One Bloom Filter is constructed for each file • Encrypting each bit location of Bloom Filter increases computational cost • Searching operation requires many addition operations
Zhong et al. [18]	• Dynamic multi-keyword fuzzy searching technique for encrypted data in cloud computing • Constructs a binary tree of Bloom Filter indexes • Euclidean distance is used to calculate the closeness of a misspelled word to a correct word • Client maintains an unencrypted binary tree for easy update in the binary tree	• A Bloom Filter is constructed for each document • Client maintains an unencrypted binary tree
Su et al. [19]	• Verifiable multi-key searchable encryption based on Garbled Bloom Filter • Cloud server sends proof of the output • Performs verifiable searching using only symmetric operation, i.e., XOR and hash operations • Document key is directly sent from data owner to data user	• Verification of the cloud server result set leads to increase in computation of the technique • Verification of the cloud server result set is not optional • Maintains many keys because one key is used to encrypt one document • Size of GBF depends on the number of shared documents • Recording of the identifiers in the result set increases the search time
Guo et al. [21]	• Bloom Filter-based multi-keyword ranked search scheme • Data owner maintains an unencrypted index tree to reduce communication cost • Bloom Filter is constructed by inserting the keywords for each document • Index tree reduces the document searching	• Insertion of null bloom Filter increases the time of the scheme • A Bloom Filter is constructed for each document • In case the number of documents is huge, the index tree cannot be maintained in RAM
EliMFS [22]	• Framework for efficient searching of encrypted data in cloud data storage • Inverted index maintains a list of documents against each keyword • Search time complexity independent of size of entire file sets	• Causes more leakage • Search time complexity depends on the number of documents associated with a keyword • Maintains many Bloom Filters • Searching time is directly proportional to the total number of documents
PASStree [23]	• PASStree is a secure searchable index structure • Bloom Filter is used to check substrings of the keywords • PASStree is constructed using the keywords where the nodes are Bloom Filters • Each leaf Bloom Filter has the substrings of a unique keyword • Each leaf node is associated with a list of documents	• Size of the parent Bloom Filter is twice that of the child Bloom Filter • Size of Bloom Filter increases with the decrease in level • Number of nodes is twice the number of unique keywords • Multiple searching path in PASStree is possible

of the matched documents are transmitted to the user. The user decrypts the identifiers and relevance score. Using this information, some documents are encrypted locally.

Song et al. [14] presented an efficient hierarchical Bloom Filter tree-index-based scheme for full-text retrieval from encrypted documents. Initially, the data owner extracts all the composite words longer than a fixed length from the document. Then the composite words are clustered based on similarity, which determines the rank of the result. Each cluster of words is inserted into a Bloom Filter. Thus, many Bloom Filters are generated for a single document. During searching, the similarity score is used to determine the similarity between the Bloom Filter and the query

Bloom Filter. The data owner calculates the semantic similarity of the words and the document. The membership value of each Bloom Filter is calculated by the data owner to maintain privacy. Along with encrypted documents and Bloom Filter, the membership value is also sent to the cloud server. After receiving the encrypted documents and Bloom Filters, the cloud server constructs the hierarchical Bloom Filter tree. The nodes in the tree are Bloom Filters. The Bloom Filters are arranged in the tree such that a Bloom Filter in the ith layer has i 1s. The tree has two types of links, inner and external. An inner link is between the Bloom Filters in adjacent layers where the Bloom Filter in the ith layer has 1 in the same bit location in the Bloom Filter in the $(i+1)$th layer. The encrypted document and Bloom Filter are connected by the external link. The nodes of the tree have two types of status, active and inactive. A node is active if at least one descendant node has an external link to an encrypted document. A node is inactive if all descendant nodes do not have any link to an encrypted document. After receiving the documents and Bloom Filters from the data owner, the cloud server searches for index nodes equal to the received Bloom Filter. If found, then the index node is linked to the encrypted document. Then the algorithm updates the status (active or inactive) of the nodes. During the deletion of a document, first, the cloud server deletes the encrypted document. Then it searches the index nodes linked to the document. The external links connecting the deleted document and index nodes are deleted. Then, it updates the status of the nodes. An authorized user sends some query keywords to the cloud server to request documents. A similar procedure is followed to construct Bloom Filters from the keywords. The data owner sends the query Bloom Filters to the cloud server. The cloud server searches for the index tree to find an index node equal to the query Bloom Filter. The search starts from the first level. The searching is terminated if the document is found or an inactive node is reached. The algorithm selects the index nodes matching at least one keyword. The cloud server uses the membership values to rank the result documents. Based on the priority, the encrypted documents are sent to the data user. The user decrypts the encrypted documents using the key provided by the cloud.

Umer et al. [15] proposed an encryption data searching technique where the encrypted index is compressed to reduce communication costs. The data owner implements a sliding window Bloom Filter where the window size is defined and keywords are sliced based on the length. The slices of the keyword are inserted into a Bloom Filter. An index file is generated for each document. After insertion of keywords in Bloom Filter, the number of 1s is noted. Then every bit of the Bloom Filter is encrypted by the Paillier algorithm [16]. In the encrypted Bloom Filter, the frequency and number of 1 bits are written in the index file. The data owner sends the index file and the encrypted documents to the cloud server. The data user creates an index using the keywords. The query Bloom Filter is sent to the cloud server. The ciphertexts are multiplied, and when the result is decrypted, it provides the sum of the plaintexts. The sum of the bits is 0, 1, or 2. The 2s represent the matches, whereas the 1s represent the unmatched bits. Then the similarity score is calculated as the ratio of the number of 2s to 1s. A polynomial is defined using plaintext to reduce the size of the result, which is sent to the data user to reduce communication costs.

Oblivious Similarity based Search (OS2) [17] is a technique for searching encrypted data in cloud servers. Bloom Filter size and the hash functions are decided by the data owner and authorized data users. The data owner creates a shared repository in the cloud server to share the outsourced data with the authorized data users. The data owner generates a homomorphic public key and shares it with the cloud server and the authorized data users to enable them access the shared repository. The data owner retrieves keywords from the documents and inserts them into a Bloom Filter. Instead of inserting a single keyword, a sliding window is used to insert the keywords. Based on the sliding window, the number of keywords are selected and hashed by the hash functions. A hash value is the bit location in the Bloom Filter. These bit locations are set to 1. After the construction of the Bloom Filter, the number of bits set to 1 is counted. Each bit of Bloom Filter is encrypted to increase privacy. When the data user wants some documents, it constructs a Bloom Filter and inserts the required keywords using the sliding window. Moreover, the documents are encrypted to protect the request from the cloud server from exploitation. The encrypted Bloom Filter is sent to the cloud server. After receiving the encrypted Bloom Filter, it is matched with the stored encrypted Bloom Filters. The frequency count of 1s helps in determining the similarity between the Bloom Filters. Along with this, Jaccard similarity coefficient is used to obtain correct similarity results. Similar Bloom Filters are selected and perform bitwise oblivious addition between the queried and selected encrypted Bloom Filters. The resultant vector is forwarded to the data user.

Zhong et al. [18] presented a dynamic multi-keyword fuzzy searching technique for encrypted data in cloud computing. First, keywords are extracted from the documents. The technique constructs a $26 \cdot 5 + 30$-bit long vector where $26 \cdot 5$ represents the letters, where a letter frequency can appear 5 times, and 30 represents common symbols and numbers. The unigram of the keyword is obtained, and the respective bit is set to 1. The vector is converted into a Bloom Filter using LSH. A Bloom Filter is constructed for each document. The index vector is constructed by

using the keyword weight. A binary tree of Bloom Filter indexes is constructed to speed up the searching process. The Bloom Filter indexes are leaf nodes. The parent node is constructed from the leaf by considering the leaf node sequentially. A bit of the parent node is the max of the corresponding bit value of the considered leaf nodes. This process continues till the construction of the root node. When a user requests documents they send keywords. The server constructs the query Bloom Filter. Then the Bloom Filter is encrypted to generate the trapdoor. The trapdoor Bloom Filter indexes are used to traverse the binary tree and search for the documents. Then the documents are transferred to the user. Any update in the document, the corresponding changes are also made in the binary tree. To reduce the communication cost, the client maintains an unencrypted binary tree.

Su et al. [19] presented a verifiable multi-key searchable encryption scheme based on Garbled Bloom Filter (GBF) [20]. The data owner uses different keys on each document for encryption. Moreover, a different key is used to encrypt the keyword of the document. After encryption, the documents and keywords are outsourced to the cloud server. In case an authorized user wants to access any document, the data owner sends the document key to the user. After receiving the key, the user downloads the corresponding encrypted keywords. The user decrypts the keywords using the key. Then the user constructs GBF using the plaintext keywords. The data user generates a token and sends it to the cloud server to request the documents using a keyword. After receiving the request, the cloud server searches the hash table for the documents. In case the document is found, the document identifier is inserted into the result set. The cloud server returns the proof, i.e., GBF and the result set to the data user. After receiving the response from the cloud server, the data user searches the GBF to verify the result. If the cloud server sends an empty result set, then k locations in GBF are checked for the special string. If found, then the result is correct; otherwise, the cloud server is malicious. On the other hand, if the result set is not empty, then the XOR operation is performed on the k locations of GBF. XOR operation is also performed on the elements of the result set. If both results of the XOR operation are equal, then the result set is correct. Verification of the cloud server result set should be made optional to the data user.

Guo et al. [21] proposed a Bloom Filter-based multi-keyword ranked search scheme. The data owner constructs an unencrypted search index tree of Bloom Filters. An identifier is generated for each document. Keywords are retrieved from the document, which are inserted into a Bloom Filter. Then the Bloom Filter is encrypted and called an index vector. Then Bloom Filter is used to construct a search index tree where the Bloom Filters are the leaf node. Initially, the internal nodes are null Bloom Filters. The internal nodes are updated from the bottom up. An internal node is updated by performing OR operation between the leaf and right Bloom Filters. After the construction of the unencrypted index tree, the Bloom Filter is encrypted to construct an index tree. The data owner sends the encrypted document and the index tree to the cloud server. The data user generates a query vector by inserting query keywords. Then the vector is encrypted by using a secret key to generate a trapdoor. The cloud server receives the trapdoor, which is the query Bloom Filter. The query Bloom Filter is compared with the nodes of the index tree. After searching, some relevant leaf Bloom Filters are selected. A similarity score is calculated between the query Bloom Filter and leaf Bloom Filters. These scores are sorted using an appropriate sorting algorithm. Then some top similar documents are sent to the data user. During the deletion of a document, its corresponding leaf Bloom Filter is deleted from the index tree. In case deletion of the Bloom Filter leads to an imbalance of the index tree, it is replaced by a null Bloom Filter whose ID is also null. The scheme maintains a list of null Bloom Filters. The updated index tree is sent to the cloud server by the data owner. In case a new document is inserted, then a new Bloom Filter is needed to be inserted into the index tree. The first null Bloom Filter list is searched. If a null Bloom Filter is found, then it is replaced by the new Bloom Filter. The updated index tree is sent to the cloud server by the data owner. In case a document is modified, then some keyword changes. Instead of deletion of the Bloom Filter, new keywords are inserted into the Bloom Filter.

Efficient Leakage-resilient Multi-keyword Fuzzy Search (EliMFS) [22] is a framework for efficient searching of encrypted data in cloud data storage. The framework has a two-stage index structure, forward index and inverted index. EliMFS framework has three stages: initialization, token generation, and searching. Bloom Filter is implemented in the token generation stage and in the inverted index. The initialization stage performs two tasks. One is the generation of a secret key, and the other is the construction of the forward and inverted index. The forward index has two parts, tag and Bloom Filter. The Bloom Filter contains the file identifiers. The inverted index maintains file identifiers and Bloom Filters. Each file identifier is associated with a Bloom Filter containing the associated keywords. Then the data owner transfers the encrypted documents and index tables to the cloud server. The token generation phase is performed for data searching on the data owner side. The phase generates a token for searching keywords. First, a single keyword is selected for searching; then, other keywords are searched. All the keywords are converted into a 2-gram and inserted into a Bloom Filter, which is encrypted. Then the data owner constructs a token

and sends it to the cloud server. During the searching phase, the cloud server uses the tag to search the inverted index to retrieve the list of encrypted file identifiers matching the keywords. The secret key is used to decrypt the file identifiers. Then the file identifiers are used to check the associated Bloom Filter to determine whether the keyword is associated with another keyword.

Pattern Aware Secure Search tree (PASStree) [23] is a string searching technique that uses Bloom Filter to check substrings of the keywords. All the possible substrings are listed from the keywords extracted from each document. PASStree is constructed using the keywords where the nodes are Bloom Filters. Each Bloom Filter in the leaf nodes stores substrings of a unique keyword. The parent node is the union of the children Bloom Filters. The size of the parent Bloom Filter is twice that of the child Bloom Filter to limit the FPP within the permissible limit. PASStree is constructed from top to bottom. First, the root Bloom Filter is constructed where all the substrings are inserted. Then two-child Bloom Filters are constructed. All the substrings are partitioned into two disjoint sets and inserted into the child Bloom Filters. This procedure is followed to construct the whole PASStree. The construction of PASStree is terminated when each leaf Bloom Filter has the substrings of a unique keyword, and the height is $\log kw$ where kw is the number of unique keywords. Each leaf node is associated with a list of documents. When a Bloom Filter is selected, then the document list is retrieved. During query operation, the searching starts from the root Bloom Filter. If the root Bloom Filter contains the queried substring (some or all), then it is searched in the children's Bloom Filters. If the children Bloom Filters contain the queried substring (some or all), then the search continues in its descendant Bloom Filters; otherwise, the search stops for that subtree. When the search reaches a leaf node, then the associated list of documents is retrieved and forwarded to the data user.

16.3 Cloud data storage management

One of the most attractive features of cloud computing is cloud storage. The cloud is able to provide infinite memory. The user only pays for the services, and the cloud takes all other responsibilities such as execution of effective data storage, data access, data retrieval, data analysis, and security. Using a distributed system continuously provides memory for storage to the user [24]. Cloud protects the data, manages it, and maintains integrity and privacy. Cloud storage services help to outsource the data, which can be accessed by data users conveniently from any part of the world. However, the cloud has to address many issues and challenges such as data integrity, availability, confidentiality, and deletion [25]. This section presents the role of Bloom Filter in the data storage and management of cloud servers. This section also includes the review of Bloom Filter-based data storage and management techniques. Table 16.2 highlights the features and limitations of the Bloom Filter-based data storage and management techniques.

kBF (Key-Value Bloom Filter) [26] is a Bloom Filter implemented for ease and efficient key-value operations for cloud computing services. The data is key–$value$ pair. First a binary string is generated for the $value$, i.e., $value$ is encoded. After generating an encoded string, the key and encoded string pair is queried into a secondary kBF (s-kBF). s-kBF helps to check the uniqueness of the encoding. The key–encoded string pair is queried to s-kBF. If found, then another encoded string is generated: otherwise, it is assigned to the key. The new key–encoded string pair is inserted into s-kBF. The encoded string to key mapping is stored in a lookup table for retrieval of key from the encoded string. Then key–encoded string pair is inserted into the main kBF, which has a cell consisting of two parts, a counter and a possibly superimposed encoding result. The counter keeps the count of the number of encoded strings mapped to that cell. The second part stores the original encoded string if it is the first encoded string mapped to that cell. Otherwise, the algorithm performs XOR operations on the mapped encoded string and the stored string. kBF performs seven operations: create, insert, query, update, delete, join, and compress. During a create operation, a new empty kBF is constructed. During an insert operation, the key–encoded string pair is hashed by k hash functions. The hashed value provides the cell locations in kBF. If the counter value is 0, then the counter is incremented, and the encoded string is stored in the other part of the cell. In case the counter is more than zero, then the counter is incremented, and an XOR operation is performed between the stored string and the mapped encoded string. During a delete operation, if the counter is non-zero, then it is decremented, and XOR operation is performed on the superimposed value and the mapped encoded string. During an update operation, the pair is deleted, and the new pair is inserted. When the number of pairs exceeds a threshold, a new kBF of the same size is constructed to insert new pairs. On the other hand, if a kBF has few pairs after many delete operations, then the compress operation is executed. The join operation is performed between two kBFs. In the operation, the counters are added, whereas the XOR operation is performed on the same corresponding cells of the superimposed part.

TABLE 16.2 Features and Limitations of Bloom Filter-based Data Storage and Management Techniques.

Technique	Features	Limitations
kBF [26]	• kBF is a Bloom Filter implemented for ease and efficient key–value operations for cloud computing services • *value* is encoded to increase security • Encoding string to *key* mapping is stored in a lookup table • Query is the most complex operation • kBF stores the encoding string	• Some information is lost due to XOR operation during join operation • Stores many types of data structures: s-kBF, kBF, and lookup table
FFBF [27]	• Bloom Filter-based space-effective scheme for massive data storage in cloud computing • Fuzzy operations reduce storage requirements • Fuzzy fold operation compresses two Bloom Filters into one • Data kept for longer time due to slow data decay • Storage space is efficiently used without compromising the accuracy • Slow data decay in FFBF is due to merging and using the same bit array as Bloom Filters	• Small-sized bit array results in inefficient fuzzy folded operation • Small-sized bit array requires frequent flushing of Bloom Filters • Implements standard Bloom Filter
CAONT [28], 2019	• Bloom Filter is used for selection of data location • Bloom Filter helps in data privacy • Bloom Filter hinders in locating the sensitive information • For each file, the Bloom Filter-based location selection is executed once	• False positive response leads to generating same location for two different data • Implements standard Bloom Filter
Yang et al. [29,30], 2020	• Fine-grained outsourced data deletion scheme • Implements invertible Bloom Filter • Verifies the storage and deletion results • Verification does not require private information of data owner and cloud server • Provides data deletion proof • Cloud server sends only the IBF to reduce the communication cost • Data owner deletes the local backup after storage verification • During delete verification, data owner involves the TPA to determine the trustworthiness of cloud server	• Cloud server sends back some information for verification of storage or deletion, which increases the communication cost • Delete verification may fail due to false positives in IBF

Fuzzy folded Bloom Filter (FFBF) [27] is a Bloom Filter-based space-effective scheme for massive data storage in cloud computing. A m-bit array is partitioned into two Bloom Filters. Data are inserted into the two Bloom Filters. When the FPP of both Bloom Filters crosses a threshold, both Bloom Filters are merged into the first half of the bit array. This merging is called the first compressed form, or the first fold. Next, the other half of the bit array is partitioned into two halves and used as two Bloom Filters. Similarly, when both Bloom Filters cross a threshold, they are merged, and the remaining bit array is partitioned to be used as two Bloom Filters. Merging and using the same bit array as Bloom Filters provides slow data decay to FFBF. During a query operation, the data is searched in the active Bloom Filter. If present, then FFBF returns $True$. Otherwise, the data is searched in the compressed Bloom Filters.

Convergent All-Or-Nothing Transform (CAONT) [28] is a secure cloud data deduplication scheme where Bloom Filter is used for the selection of data location. To achieve data privacy, the data owner partitions the data package into two parts. The last 256 bits are trimmed from the data package. The trimmed data package is the first part, whereas the trimmed part is the second part. The trimmed data package is stored in a location selected randomly among 256 locations. The location is selected using Bloom Filter. The packages are re-encrypted using a file key. The data owner sends the re-encrypted packages to the cloud server. For location selection, initially, all cells of Bloom Filter are set to 0. For generating a 256-bit location, a random key is obtained, which is hashed by 256 hash functions. These hash functions provide 256 cell locations in Bloom Filter. All these cells are set to 1. In case any hash function generates the same cell location, then again, the hash function recomputes the value.

Yang et al. [29,30] presented an efficient invertible Bloom Filter (IBF) based fine-grained outsourced data deletion scheme. Also, the technique verifies the storage and deletion results. During data outsourcing, the information is encrypted and partitioned into different blocks. A unique random integer is chosen as the index of a block. The data owner sends the encrypted data blocks and their indexes to the cloud server. The cloud server constructs an IBF and inserts the data block indexes. Then the cloud server generates a signature for IBF. As evidence of data storage, the cloud server sends IBF and its signature to the data owner. The data owner verifies whether the cloud server has correctly stored the data by first checking the validity of the signature. In case the signature is valid, the data owner uses its data block indexes to construct its own IBF. Then the algorithm compares its own IBF and the IBF returned by the cloud server. If both IBFs are the same, then the data owner deletes the local backup. In case the signature is invalid, or the IBFs are different, then the data owner concludes the cloud server is untrustworthy. When the data owner wants to delete a data block, then it generates a signature and sends a delete command to the cloud server. The delete command contains the signature, data block indexes, and time-stamp. After receiving the delete command, the cloud server verifies the validity of the command, i.e., the validity of the signature. If the signature is valid, the data block is deleted, and indexes are removed from IBF. The cloud server generates a signature and sends the IBF with the signature to the data owner. The data owner also verifies whether the cloud server has correctly deleted the data blocks. First, the data owner checks the validity of the signature. If valid, then the deleted indexes are checked in the IBF. If some index is present in the IBF, then the data owner forwards the disputed indexes to the third-party auditor (TPA). TPA lists all the indexes inserted into the IBF and checks the presence of the disputed indexes. In case the indexes are present, the TPA concludes the cloud server is untrustworthy.

16.4 Discussion

Bloom Filter is implemented for fast searching of encrypted documents. The data owner stores data in the cloud server. The cloud server is responsible for protecting the data and make the data available to authorized data users. Instead of plaintext, the data owner stores encrypted data in the cloud server to prevent the exploitation of data by cloud service providers. The data user requests document(s) to the cloud server by forwarding the required keywords. However, searching for the required document(s) for data user(s) from a huge number of encrypted documents requires lots of time. The delay in searching adversely affects the quality of the service of the cloud service provider. Hence, Bloom Filter is used to insert the keywords because checking the queried keywords in Bloom Filter is faster. Bloom Filter is constructed by the cloud server or data owners. A Bloom Filter is constructed for every document. The data owners send the Bloom Filter along with the encrypted documents to further enhance the privacy. Majority of Bloom Filter-based indexing and searching techniques follow this procedure. Bloom Filter is also implemented for data deduplication to reduce the memory requirement in cloud servers.

In Bloom Filter-based indexing and searching techniques, one Bloom Filter is constructed for each document. Cloud is preferred for acquiring resources by paying less money. Hence, many data users are availing the cloud services, and the number of documents is increasing drastically. Bloom Filter is the fastest when it is present in RAM. However, maintaining a huge number of Bloom Filters for a huge number of documents is difficult. One solution is implementing multi-set Bloom Filters. One Bloom Filter can be used to store keywords of multiple documents. Another solution is constructing a single Bloom Filter where many keywords match multiple documents. However, it requires more preprocessing before storage. Another issue is the introduction of error in Bloom Filter during transmission. The data owner constructs the Bloom Filters and transmits the Bloom Filters to the cloud server. Bloom Filter is a bit array, so a single change of bit may introduce many false positives in case many data are hashed to that bit location.

Table 16.3 highlights a comparison of the techniques mentioned in the chapter based on various parameters. Except for a few, other techniques implement a standard Bloom Filter. Performing operations on many Bloom Filters is costly. Along with that, the standard Bloom Filter requires more hash functions to maintain a permissible FPP, which further increases the computational cost. Implementing Bloom Filter variants reduces the FPP and the number of hash functions. It will also reduce the computational cost and improve the performance. Thus, these techniques should explore Bloom Filter variants for better efficiency and high performance.

TABLE 16.3 Comparison of Bloom Filter based Cloud Computing Techniques: Variant, Bloom Filter variant; Encrypted BF, whether Bloom Filter is encrypted or not; BF Purpose, purpose of the Bloom Filter; Purpose, purpose of the technique.

Technique	Variant	Encrypted BF	BF Purpose	Purpose
Poon and Miri [11], 2015	Standard	No	Membership checking of keywords	Encrypted data Searching
Zhu et al. [12], 2015	Standard	No	Membership checking of keywords	Encrypted data Searching
Yu et al. [10], 2015	Standard	No	Membership checking of keywords	Data Searching
SICBF [13], 2016	Counting	No	Document count	Encrypted data Indexing
Song et al. [14], 2016	Standard, Hierarchical	No	Membership checking of words	Encrypted data Searching
Umer et al. [15], 2017	Standard	Yes	Membership checking of keywords	Encrypted data Searching
kBF [26], 2017	kBF	No	Stores key–encoded string pair	Data Storage
OS2 [17], 2017	Standard	Yes	Membership checking of keywords	Data Searching
FFBF [27], 2019	Standard	No	Membership checking	Data storage
Su et al. [19], 2019	Garbled Bloom Filter	No	Verification of result set	Encrypted data Searching
Guo et al. [21], 2019	Standard	Yes	Membership checking of keywords	Encrypted data Searching
CAONT [28], 2019	Standard	No	Location selection	Data Storage
EliMFS [22], 2020	Standard	Yes	Membership checking of file identifiers and keywords	Encrypted data Searching
Zhong et al. [18], 2020	Standard	No	Membership checking of keywords	Encrypted data Searching
Yang et al. [29], 2020	Inverted	No	Checks the data block indexes	Data Storage and Delete verification
PASStree [23], 2021	Standard	No	Membership checking of substring of the keywords	Data Searching

16.5 Conclusion

This chapter discussed the role of the Bloom Filter in cloud computing. The cloud computing topics covered in the chapter are encrypted data indexing and searching, data storage, and management. Due to security and privacy reasons, the data owner stores encrypted data instead of plaintext. However, searching within encrypted data is difficult and time-consuming. Hence, Bloom Filter is implemented to address both issues. An issue associated with Bloom Filter from this aspect is that one Bloom Filter is constructed for each document. So, cloud servers have to maintain many Bloom Filters along with encrypted data. This can be solved by implementing the multi-set Bloom Filter. Data storage and management in the cloud require efficient data deletion and deduplication techniques. In this case, Bloom Filter also provides excellent assistance for faster execution of techniques to solve the issues.

References

[1] Amazon, Free cloud computing services – AWS free tier, https://aws.amazon.com. (Accessed 11 February 2022).
[2] Microsoft Azure, Cloud computing services – Microsoft Azure, https://azure.microsoft.com/en-us/. (Accessed 11 February 2022).
[3] Cloud computing services – Google Cloud, https://cloud.google.com/. (Accessed 11 February 2022).
[4] IBM Cloud – India – IBM, https://www.ibm.com/in-en/cloud. (Accessed 11 February 2022).
[5] Adobe Creative Cloud – Details and Products – Adobe, https://www.adobe.com/in/creativecloud.html. (Accessed 11 February 2022).
[6] H. Tabrizchi, M. Kuchaki Rafsanjani, A survey on security challenges in cloud computing: issues, threats, and solutions, J. Supercomput. 76 (12) (2020) 9493–9532.
[7] M. Kumar, S. Sharma, A. Goel, S. Singh, A comprehensive survey for scheduling techniques in cloud computing, J. Netw. Comput. Appl. 143 (2019) 1–33, https://doi.org/10.1016/j.jnca.2019.06.006.
[8] P. Kumar, R. Kumar, Issues and challenges of load balancing techniques in cloud computing: a survey, ACM Comput. Surv. 51 (6) (2019), https://doi.org/10.1145/3281010.
[9] Z. Xiao, Y. Xiao, Security and privacy in cloud computing, IEEE Commun. Surv. Tutor. 15 (2) (2013) 843–859, https://doi.org/10.1109/SURV.2012.060912.00182.
[10] C.-M. Yu, C.-Y. Chen, H.-C. Chao, Privacy-preserving multikeyword similarity search over outsourced cloud data, IEEE Syst. J. 11 (2) (2017) 385–394, https://doi.org/10.1109/JSYST.2015.2402437.
[11] H.T. Poon, A. Miri, A low storage phase search scheme based on Bloom filters for encrypted cloud services, in: 2015 IEEE 2nd International Conference on Cyber Security and Cloud Computing, 2015, pp. 253–259.
[12] X. Zhu, Q. Liu, G. Wang, Verifiable dynamic fuzzy search over encrypted data in cloud computing, in: G. Wang, A. Zomaya, G. Martinez, K. Li (Eds.), Algorithms and Architectures for Parallel Processing, Springer International Publishing, Cham, 2015, pp. 655–666.

[13] H. Yao, N. Xing, J. Zhou, Z. Xia, Secure index for resource-constraint mobile devices in cloud computing, IEEE Access 4 (2016) 9119–9128, https://doi.org/10.1109/ACCESS.2016.2622299.

[14] W. Song, B. Wang, Q. Wang, Z. Peng, W. Lou, Y. Cui, A privacy-preserved full-text retrieval algorithm over encrypted data for cloud storage applications, J. Parallel Distrib. Comput. 99 (2017) 14–27, https://doi.org/10.1016/j.jpdc.2016.05.017.

[15] M. Umer, T. Azim, Z. Pervez, Reducing communication cost of encrypted data search with compressed Bloom filters, in: 2017 IEEE 16th International Symposium on Network Computing and Applications (NCA), 2017, pp. 1–4.

[16] P. Paillier, Public-key cryptosystems based on composite degree residuosity classes, in: International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 1999, pp. 223–238.

[17] Z. Pervez, M. Ahmad, A.M. Khattak, N. Ramzan, W.A. Khan, OS2: oblivious similarity based searching for encrypted data outsourced to an untrusted domain, PLoS ONE 12 (7) (2017) 1–22, https://doi.org/10.1371/journal.pone.0179720.

[18] H. Zhong, Z. Li, J. Cui, Y. Sun, L. Liu, Efficient dynamic multi-keyword fuzzy search over encrypted cloud data, J. Netw. Comput. Appl. 149 (2020) 102469, https://doi.org/10.1016/j.jnca.2019.102469.

[19] Y. Su, J. Wang, Y. Wang, M. Miao, Efficient verifiable multi-key searchable encryption in cloud computing, IEEE Access 7 (2019) 141352–141362, https://doi.org/10.1109/ACCESS.2019.2943971.

[20] C. Dong, L. Chen, Z. Wen, When private set intersection meets big data: an efficient and scalable protocol, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS'13, Association for Computing Machinery, New York, NY, USA, 2013, pp. 789–800.

[21] C. Guo, R. Zhuang, C.-C. Chang, Q. Yuan, Dynamic multi-keyword ranked search based on Bloom filter over encrypted cloud data, IEEE Access 7 (2019) 35826–35837, https://doi.org/10.1109/ACCESS.2019.2904763.

[22] J. Chen, K. He, L. Deng, Q. Yuan, R. Du, Y. Xiang, J. Wu, Elimfs: achieving efficient, leakage-resilient, and multi-keyword fuzzy search on encrypted cloud data, IEEE Trans. Serv. Comput. 13 (6) (2020) 1072–1085, https://doi.org/10.1109/TSC.2017.2765323.

[23] A.X. Liu, R. Li, Privacy Preserving String Matching for Cloud Computing, Springer International Publishing, Cham, 2021, pp. 231–251.

[24] Y.S. Jghef, S. Zeebaree, et al., State of art survey for significant relations between cloud computing and distributed computing, Int. J. Sci. Bus. 4 (12) (2020) 53–61.

[25] N. Vurukonda, B.T. Rao, A study on data storage security issues in cloud computing, in: 2nd International Conference on Intelligent Computing, Communication & Convergence, ICCC 2016, Bhubaneswar, Odisha, India, 24–25 January 2016, Proc. Comput. Sci. 92 (2016) 128–135, https://doi.org/10.1016/j.procs.2016.07.335.

[26] S. Xiong, Y. Yao, S. Li, Q. Cao, T. He, H. Qi, L. Tolbert, Y. Liu, kbf: towards approximate and Bloom filter based key–value storage for cloud computing systems, IEEE Trans. Cloud Comput. 5 (1) (2017) 85–98, https://doi.org/10.1109/TCC.2014.2385063.

[27] A. Singh, S. Garg, K. Kaur, S. Batra, N. Kumar, K.-K.R. Choo, Fuzzy-folded Bloom filter-as-a-service for big data storage in the cloud, IEEE Trans. Ind. Inform. 15 (4) (2019) 2338–2348, https://doi.org/10.1109/TII.2018.2850053.

[28] H. Yuan, X. Chen, J. Li, T. Jiang, J. Wang, R.H. Deng, Secure cloud data deduplication with efficient re-encryption, IEEE Trans. Serv. Comput. 15 (1) (2022) 442–456, https://doi.org/10.1109/TSC.2019.2948007.

[29] C. Yang, Y. Liu, X. Tao, F. Zhao, Publicly verifiable and efficient fine-grained data deletion scheme in cloud computing, IEEE Access 8 (2020) 99393–99403, https://doi.org/10.1109/ACCESS.2020.2997351.

[30] C. Yang, X. Tao, F. Zhao, Y. Wang, A new outsourced data deletion scheme with public verifiability, in: International Conference on Wireless Algorithms, Systems, and Applications, Springer, 2019, pp. 631–638.

This page intentionally left blank

CHAPTER

17

Applications of Bloom Filter in biometrics

17.1 Introduction

The conventional method of authentication is providing ID and password. However, it has led to many issues. An easy password is easy to guess, whereas the user can easily forget a complex password. Nowadays, every website is requesting the user to create his/her account. So, there are many passwords to remember. Moreover, websites discourage the use of the same password. But remembering many passwords is a difficult task. Furthermore, the websites request a periodic change of passwords to maintain security. Many attacks can easily retrieve the password of a user. Some examples are dictionary attacks, brute force attacks, and eavesdropping attacks. In dictionary attacks, different combinations of passwords are repeatedly applied to know the actual password. A brute force attack tries to guess the password. In an eavesdropping attack, the attacker retrieves the password during the transmission of the password through the Internet. Thus, biometric features are used for authentication and strong security.

Biometric features such as the pattern in our iris, fingerprint, etc., are unique for every human being, including the twins. Thus, currently, researchers are exploring these biometric features as passwords for the authentication of an individual. Examples of some applications of biometrics are forensic sectors, border immigration control, surveillance, and human–computer interaction. The user does not have trouble remembering the password. Moreover, the attacker cannot forge these biometric features. However, the application requires some extra technology to record the biometric features. Furthermore, the computation of biometric features is compute-intensive. The biometric data is mostly in images that need ample memory space for storage. The images are converted into a binary vector to reduce computation complexity. The primary purpose of biometric indexing is quick verification of biometric query templates. Binary search trees take less time for searching. However, constructing a binary tree using biometric data, i.e., images or binary vectors, is costly because of the high memory overhead. Cancelable biometrics aims to transform the biometric data into an irreversible form that cannot be reverted to its original data. Bloom Filter is the perfect solution for biometric indexing and cancelable biometrics.

Bloom Filter has a simple architecture that occupies low memory but inserts many strings. Biometric indexing uses Bloom Filter for the construction of binary trees. Bloom Filter-based binary trees have low memory overhead with easy query comparison. The original data are mapped to the Bloom Filter in cancelable biometric, which is a binary vector. This binary vector has small size but performs fast operations. This chapter explores the implementation of Bloom Filter in various biometric applications to enhance performance. Section 17.2 explores the role of Bloom Filter in making the indexing and searching procedure easy. Section 17.3 highlights the role of Bloom Filter in cancelable biometrics. Section 17.4 provides a discussion of the advantages and limitations of Bloom Filter in biometrics.

17.2 Biometrics

Biometrics is a word derived from Greek words: "bio" is life and "metric" is a measure. Biometrics is a body measurement and calculation using human characteristics. Examples of human characteristics that are mostly used in biometrics are iris, fingerprint, face, palm, etc. "Biometrics is the science and technology used to uniquely identify individuals based on their physical, chemical or behavioral traits" [1]. Biometrics is used in many fields such as human–computer interaction, access control, forensic sectors, surveillance, border immigration control, etc. Biometrics is classified mainly into three categories [2]: physical, behavioral, and chemical. Physical biometrics is the measurement of the physical characteristics of a human, such as a face, iris, fingerprint, etc. Behavioral biometrics is the measurement of behavioral traits of a human such as speech, hand gestures, etc. Chemical biometrics is the measurement of human characteristics due to chemical reactions in the human body, such as odor. Biometric recognition

Bloom Filter https://doi.org/10.1016/B978-0-12-823520-1.00024-9

Copyright © 2023 Elsevier Inc. All rights reserved.

is the verification or identification of a person. Verification is determining whether the user's biometrics feature is already present in the stored database. Identification is determining the identity of the user using the stored database.

A biometric template is the digital representation of a human characteristic, i.e., fingerprint, face, iris, etc. It is constructed by extracting the unique features of the human characteristic. The features are transferred into a form from which it is difficult to retrieve the original data. This transformed form is called a biometric template. It is stored in the database for biometric authentication and identification.

The aim of biometric indexing is to reduce the number of template comparisons during biometric verification performed in a large biometric database. One of the main issues is an increase in the response time due to the fuzziness of biometric data indexing in biometric databases [3]. Fuzziness in the biometric data is caused due to high dimensionality of the biometric data. Many biometric data are not exact, but very similar [4]. However, a technique has to compromise between accuracy and faster response time. Moreover, the indexing technique utilizes complex data structures that perform reconstruction of the data structure during insertion and deletion operations. Many biometric indexing techniques have implemented Bloom Filter to reduce the response time. This section presents a review of these techniques. Table 17.1 highlights various features and limitations of the Bloom Filter-based biometric indexing and searching techniques.

TABLE 17.1 Features and Limitations of Biometric Indexing and Searching Techniques.

Technique	Features	Limitations
Rathgeb et al. [3]	• Implements Bloom Filter for iris biometric database indexing • Binary tree of Bloom Filters reduces the search space by $O(\log N)$ • XOR operation determines the dissimilarity between two Bloom Filters	• A Bloom Filter is constructed for each block of feature • Maintains many Bloom Filters • Implements standard Bloom Filter • Column storing the block boundaries information causes miss-alignment and gets inserted into neighboring Bloom filters • FPP of the tree increases with decrease in the level • Deletion of a node in binary Bloom Filter tree requires reconstruction of the whole tree
Lai et al. [5]	• Bloom Filter-based alignment-free Indexing-First-One hashing technique • Does not involve pre-alignment	• Requires more computational time compared to conventional IFO hashing technique • Maintains multiple Bloom Filters • Implements standard Bloom Filter
Drozdowski et al. [6]	• Bloom filter and binary search tree based multi-iris indexing system • Bloom Filter is rotation invariant • During comparison, template alignment is not required	• Some information is lost in template merging due to collision • Bloom Filter size becomes double in template concatenation method • Maintains multiple trees • Maintains multiple Bloom Filters • Implements standard Bloom Filter
Drozdowski et al. [4]	• Bloom Filter-based indexing technique for protected iris templates • Partitioning of iris-code into real and imaginary parts enhances performance • Partitioning of iris-code minimizes the negative effects of template protection on the technique • Bloom filter-based templates are rotation-invariant to a certain degree • During query operation, the searching of the trees are performed in parallel	• Maintains many trees • Each tree has many Bloom Filters • Query operation requires searching in many trees • Implements standard Bloom Filter
Shomaji et al. [7]	• Large-scale biometric search system based on hierarchical Bloom Filter • Bloom Filter reduces storage space • Bloom Filter helps in determining the membership of noisy data • Quantization reduces size and noise of a template • Address the false positive issue by inserting concatenations of the substrings	• Each level of HBF maintains many Bloom Filters • High number of levels in HBF increases the overhead due to Bloom Filter compared to gained advantages • Implements standard Bloom Filter

Rathgeb et al. [3] presented a technique that implements Bloom Filter for iris biometric database indexing. The iris features are represented in binary form using a two-dimensional vector. The vector is partitioned into equal-sized

blocks. A Bloom Filter is constructed for each block. In each block, each column is hashed by a hash function which provides a decimal value. The decimal value is the bit location in Bloom Filter, which is set to 1. Bloom Filters are used to construct a binary tree of Bloom Filters. All Bloom Filters are merged into a single Bloom Filter, which is the root. The children Bloom Filters of the root is constructed by merging half of the Bloom Filters as the left child and half of the Bloom Filters as the right child. A similar procedure is followed to construct the binary tree. All the Bloom Filters are present as the leaf nodes at the lowest level. The dissimilarity between two Bloom Filters is calculated by performing an XOR operation, which helps when calculating the number of 1s not matching in the Bloom Filters. The count is normalized by the Hamming weight of both Bloom Filters. FPP of the tree decreases with an increase in the level. The dissimilarity between two children nodes is calculated to determine the FPP of each level under a threshold value. It is normal if the dissimilarity decreases with increasing level and the dissimilarity is within a threshold value.

Lai et al. [5] presented a Bloom Filter-based alignment-free Indexing-First-One (IFO) hashing technique. The iris images are converted to a binary string inserted into the Bloom Filters. Some Bloom Filters are concatenated and represented by a bigger Bloom Filter. A permutation token is generated, containing the P permutations. The Bloom Filter is permuted column-wise P times. All the permuted Bloom Filters are multiplied element-wise to obtain a product code. Next, in each row of the product code, first, a few elements are selected while others are discarded. The indexing corresponds to the first occurrence bit 1 among the selected elements. Finally, the algorithm executes the modulo threshold function to obtain the IFO hashing. It uses the similarity score for matching.

Drozdowski et al. [6] proposed a Bloom Filter and binary search tree-based multi-iris indexing system. The iris images are converted to binary iris-code represented as a two-dimensional matrix. The matrix is partitioned into equal-sized blocks. The columns of each block are inserted into a Bloom Filter. Each column is transformed into a decimal value. The decimal value is the bit location in a Bloom Filter, which is set to 1. The dissimilarity score between two Bloom Filters is the normalized Hamming distance among them. Bloom Filters are constructed for multi-iris biometric templates, i.e., left and right eye iris. The biometric templates are merged using template concatenation and template merging. In template concatenation, both the left- and right-eye Bloom Filter vectors are concatenated, i.e., the binary strings of the Bloom Filters are concatenated. In this case, the Bloom Filter size becomes double. Both Bloom Filters are merged in template merging by executing an OR operation. In this case, the Bloom Filter size remains the same. But some information is lost due to the collision. The binary tree is constructed using the Bloom Filters. Performing OR operations among them merges all Bloom Filters. The resultant Bloom Filter is the root node in the tree. The Bloom Filters are partitioned into two halves. The first-half Bloom Filters are merged using OR operation to construct the left child node of the root. Similarly, the other-half Bloom Filters are merged to construct the right child node of the root. Using the Bloom Filters of the left child node, its two children nodes are constructed. A similar procedure is followed to construct the whole binary tree. All Bloom Filters are present individually as leaf nodes. Searching is performed by traversing the nodes. During searching, the query iris image is transformed into the biometric template, i.e., construction of Bloom Filters. The dissimilarity score is calculated between the query Bloom Filters and the two-child node Bloom Filters. The node having the lowest dissimilarity score is selected and traversed in that direction. The searching continues till the leaf node is reached. If the dissimilarity score is below a threshold value in the leaf node, then the query is considered a member of the database. In the case of a large dataset, multiple trees are constructed.

Drozdowski et al. [4] presented a Bloom Filter-based indexing technique for protected iris templates. The iris-code is partitioned into real and imaginary parts to minimize the negative effects of template protection on the technique. A row-based permutation is applied to the iris-code parts to reduce the possibility of inversion attacks. The iris-codes are partitioned into blocks. Each block is hashed to a Bloom Filter. The columns in each block are converted into a decimal value. The decimal value is the bit location of Bloom Filter, which is set to 1. The Bloom Filters are partitioned into several sets. Each set of Bloom Filters is used to construct a tree. All Bloom Filters in a set are merged by performing OR operation to construct the tree's root node. The children of the root are constructed by splitting the Bloom Filters of the set into two halves. One-half of Bloom Filters are merged by OR operation to construct the left child. Following the same procedure, the other half of Bloom Filters are used to construct the right child. All the Bloom Filters are present as leaf nodes in the tree. The dissimilarity score between the root node and the query Bloom Filters is calculated during query operation. Based on the score, some roots are selected. In these trees, the nodes are searched till the query Bloom Filters matches. Traversing is performed by calculating the dissimilarity score between the node Bloom Filter and query Bloom Filters.

Shomaji et al. [7] proposed a large-scale biometric search system based on hierarchical Bloom Filter (HBF). The process essentially has two steps, database construction and membership checking. Initially, many pre-processing

steps are executed on face images. Then the quantization is performed on the processed templates to reduce the size and noise of the template. The quantized templates are inserted into the HBF. During the query, the images are converted to quantized templates. Then it is searched in HBF. If HBF has the query or a noisy template with a minimum number of Bloom Filters matching, it is considered a member of the database. The quantized templates are partitioned into fixed-sized blocks. Each block is inserted into a single Bloom Filter. The column is a binary string. HBF consists of multiple standard Bloom Filters of the same size. In HBF, the substring and its concatenated strings are inserted. The whole binary string is inserted in the Bloom Filter located at the highest level, i.e., level 0 of the HBF. In level 1, the binary string is partitioned into fixed-size blocks, and each block is inserted into a Bloom Filter. In level 2, the binary string is partitioned into fixed-size blocks where the size is smaller than the block size of level 1. A similar procedure is followed to insert binary strings into the Bloom Filter and construct HBF to a certain level. This procedure helps in inserting concatenations of the substrings.

Overall, many techniques and schemes follow the same steps for generating the Bloom Filter-based biometric template. Gomez et al. [8] presented a protected biometric template generation based on Bloom Filter. The technique has introduced a new step called structure-preserving feature rearrangement. Moreover, the biometric templates are cross-matching attack resistant. Sadhya and Singh [9] proposed a biometric template generation based on Bloom Filter. The technique generates a key matrix. The biometric template is generated by performing an XOR operation between the Bloom Filters and key matrix.

17.3 Cancelable biometrics

The main aim of cancelable biometrics is to provide security and privacy. In cancelable biometrics, the data is distorted to disable the intruder from recognizing the original data; however, identification can be performed on the distorted data. The cancelable biometrics techniques should have four characteristics: renewability, unlinkability, irreversibility, and performance [10]. Renewability is the technique's ability to generate and issue new biometric templates and revoke old biometric templates in case of any attack. Unlinkability indicates the generation of different biometric templates from the same biometric data. Irreversibility is the inability of the cancelable template to obtain the original image of the biometric from the template. Performance is not compromising the technique's performance while maintaining the other characteristics. The cancelable biometrics techniques have to maintain another characteristic called diversity. Diversity refers to the restriction of cross-matching in biometric templates. Unlinkability is important because it addresses the record multiplicity attacks [11], coalition, or correlation attack [12]. Furthermore, biometric templates obtained from the same biometric data should also have a dissimilarity score higher than a threshold value to prevent cross-matching attack [8]. See Table 17.2.

Rathgeb et al. [13] proposed an adaptive Bloom Filter-based framework to generate an irreversible representation of multiple biometric templates. The technique has considered face and iris features. Bloom Filter is used for retrieving an irreversible representation of binary face and iris features. Both face and iris binary features are represented in a two-dimensional matrix. The matrix is partitioned into equal-sized blocks. For each block, a Bloom Filter is constructed. Initially, Bloom Filter is initialized to 0. A hash function hashes each column, and the hashed value is used to set a bit to 1 in Bloom Filter. Thus, two sets of Bloom Filters are constructed for face and iris features, respectively. Then the algorithm performs a fusion of both Bloom Filters. An OR operation is performed between the corresponding face and iris Bloom Filters. The fusion operation enhances the privacy of the features.

Li et al. [14] presented Bloom Filter-based fingerprint template generation. First, the fingerprint image passes through a pre-alignment module. The pre-alignment module considers only the minutiae located in a circle for the binary template. The aligned fingerprint follows Yang et al. [15] scheme for binary string generation. In a binary template, the number of rows is fixed, whereas the number of columns depends on the number of minutiae in each sample. A binary template contains N-dimensional binary matrices generated from each minutiae vicinity. The binary string or vector size is reduced by performing an XOR operation between the two halves of the binary string. The reduced binary vector is partitioned into blocks from vertical and horizontal directions. A Bloom Filter is constructed for each block. The columns in each block are inserted into a Bloom Filter. The fingerprint verification is performed by comparing the dissimilarity score of the Bloom Filters.

Qiu et al. [16] proposeed a Bloom Filter-based generation of cancelable palmprint templates. First, the algorithm computes CompCode map [17] of ROI of palmprints using Gabor filters, which extract the orientation of palmprint edge information. The CompCode is transformed to binary form. Then binary CompCode is partitioned into equal-sized blocks. The binary CompCode blocks are encoded using a coding rule. The encoding is performed on each

TABLE 17.2 Features and Limitations of Cancelable Biometrics Techniques.

Technique	Features	Limitations
Rathgeb et al. [13]	• Framework to generate an irreversible representation of multiple biometric templates • Implements adaptive Bloom Filter • Feature level fusion of different biometrics (face and iris) to a single protected template • Improves privacy compared to the systems based on a single biometric trait • Bloom Filter is used for retrieving irreversible representation of binary face and iris features • Fusion process highly improves the privacy	• A Bloom Filter is constructed for each block of feature • Maintains many Bloom Filters • Implements standard Bloom Filter • Distribution of bits of the final protected template is unbalanced
Li et al. [14]	• Bloom Filter-based fingerprint template generation • Pre-alignment module enhances robustness • XOR operation is performed on biometric template to reduce the size by half • An optimal word size is essential for high performance • Faster match response	• Computational complexity increases with increase in the word size • Maintains many Bloom Filters • Implements standard Bloom Filter
Qiu et al. [16]	• Generation of cancelable palmprint templates • CompCode maps the image from the grayscale space to the direction information space • CompCode occupies less memory • CompCode has high recognition accuracy • CompCode has excellent recognition effect for lower resolution palmprint • Coding rule does an irreversible transformation of the feature blocks	• Constructs one Bloom Filter for each block • Maintains many Bloom Filters • Implements standard Bloom Filter
You and Li [18]	• Bloom Filter-based cancelable biometrics technique • XOR operation is used for fusion of fingerprint binary features and the binary face features into a single template • Fusion features are transformed into an irreversible form using Bloom Filter • Technique maintains diversity by modifying the random matrix • When the application detects any compromise in the security, it can change the random matrix • Irreversibility is made possible by using Bloom Filter	• Unfixed number of extracted fingerprint minutiae requires matching the registered fusion features one-by-one • Some information is lost due to Bloom Filter • Maintains many Bloom Filters • Implements standard Bloom Filter
Gomez-Barrero et al. [21]	• Bloom Filter-based multi-biometric template protection scheme • Bloom Filter size depends on the feature • OR operation makes it infeasible to determine which bit is set to 1 by a feature in the final Bloom Filter • OR operation reduces information loss and enhances accuracy • Many sparse Bloom Filters can be fused without degrading the performance • Weighted score level fusion enhances verification accuracy	• Fusion of several Bloom Filters depends on the sparsity of the Bloom Filters • Less dense Bloom Filter are fused • Maintains many Bloom Filters • Implements standard Bloom Filter
Ajish and Anilkumar [22]	• Iris template generation using double Bloom Filter • Double Bloom Filter-based feature transformation enhances accuracy • Enhances data compression ratio • Enhances irreversibility • Enhances unlinkability • Bloom Filter reduces match response time • Reduces the false acceptance rate • Quick match response • Uses two security keys to increase security	• Double Bloom Filter increases single Bloom Filter size • Maintains many Bloom Filters • Implements standard Bloom Filter

column of a block. Each encoded block is inserted into a Bloom Filter. The technique constructs one Bloom Filter for each block. An XOR operation is performed between each column of block and a secret key. The value is converted to a decimal value. This value provides the bit location in the Bloom Filter, which is set to 1. The above procedure is followed during authentication to obtain the cancelable palmprint features. Then the features are classified using Support Vector Machine (SVM) classifiers for identification.

You and Li [18] presented a Bloom Filter-based cancelable biometrics that performs XOR operation for the fusion of fingerprint and face binary features into a single template. An array mapping technique [19] is implemented to map the relative distance and direction difference of the minutiae on the two-dimensional array to generate the binary fingerprint features. Biohashing [20] helps to generate the user's biometric vector and a random matrix. These are used to construct the cancelable face binary features from the face images. The fingerprint binary features are partitioned into groups. Then XOR operation is performed between each group of fingerprint binary features and cancelable face binary features. The result is the fusion features. The fusion features are transformed into an irreversible form using Bloom Filter. Each column of the fusion feature is converted into a decimal value, which is the bit location in Bloom Filter set to 1. When a fingerprint and face feature is queried, the query fusion features in the irreversible form are first generated, i.e., Bloom Filters are constructed. The distance score and dissimilarity are calculated between the Bloom Filters of query fusion features and the registered fusion features. These values are used to authenticate the query fusion features. The technique maintains diversity by modifying the random matrix. Different systems can use different random matrices to unlink the templates from each other. Cancelability is maintained by changing the random matrix. When the application detects any compromise in the security, it can change the random matrix to generate fusion features that follow different rules. Earlier fusion features are considered invalid. Bloom Filter helps in achieving irreversibility.

Gomez-Barrero et al. [21] proposed a multi-biometric template protection scheme based on Bloom Filter. The Bloom Filter size is varying, which depends on the type of feature. An image is converted into a binary vector. The vector is partitioned into blocks. A Bloom Filter is constructed for each block. The columns in each block are inserted into that block's respective Bloom Filter. After construction of the Bloom Filters, an OR operation is performed on all Bloom Filters to generate a single biometric template. A vector is generated containing pre-defined positions. The position specifies the location to perform the OR operation. The position vector helps in performing OR operations among different size Bloom Filters. Hence, an OR operation is not performed on the whole Bloom Filter. Rather, OR operation is performed on some bits, and the rest is copied to the final template. Furthermore, a weight is assigned to every feature. Based on the weight, the feature template contributes to the final biometric template.

Ajish and Anilkumar [22] presented an iris template generation using a double Bloom Filter. A single Bloom Filter is partitioned into two parts. The double Bloom Filter refers to the two halves of a Bloom Filter. The iris images are transformed into a binary feature represented by two-dimensional vectors. Then the vector is partitioned into blocks. The column in each block is partitioned into two parts. The first is inserted into the first Bloom Filter, whereas the second is inserted into the second Bloom Filter.

Many other techniques are also proposed for cancelable biometrics, which implement Bloom Filter. Kauba et al. [23] proposed a cancelable template for finger vein features. Bansal and Garg [24] proposed format-preserving encryption and Bloom Filters based on a cancelable biometric template protection scheme. First, a random symmetric key is generated by a pseudorandom number generator. Each column is encrypted by the generated key, which is the format-preserving encryption. Then a similar procedure is followed to insert the encrypted template into the Bloom Filter.

17.4 Discussion

The main purpose of indexing is fast searching. The time complexity of a binary search tree is $O(\log n)$, where n is the total number of search samples. However, constructing a binary search tree using binary vectors does not provide substantial advantages. Such trees require huge memory to store all the binary vectors of all samples, and comparison between the $\log n$ binary vector nodes with the query binary vector is costly. Nonetheless, comparison between Bloom Filters is easy. The Bloom Filter comparison is performed by using dissimilarity scores. The dissimilarity scores between the left and right nodes are compared. Suppose the search space moves towards low dissimilarity scores. Thus, construction and traversing the Bloom Filter binary tree is easy and requires less computational time.

Cancelable biometrics is transforming biometric data into an irreversible form. In this application, Bloom Filter provides a cost-effective and easy solution. The overall procedure of Bloom Filter-based cancelable biometric is as

follows: a biometric image is converted into a binary vector. The binary vector is a two-dimensional array having a huge size. Hence, the vector is partitioned into blocks. A Bloom Filter is constructed for each block. Each column in each block is considered a string inserted into the corresponding block's Bloom Filter. The query biometric image is converted into Bloom Filters. The comparison of the query image is performed with the sample biometric by calculating the dissimilarity score between the Bloom Filters,

$$Dissimilarity\ Score = \frac{1}{|blocks|} \sum_{BF=1}^{b} \frac{HD(BF_Q, BF_S)}{|BF_Q^1| + |BF_S^1|},$$

where $|blocks|$ is the total number of blocks, b is the total number of Bloom Filters constructed for the corresponding block, HD is Hamming distance, BF_Q is queried Bloom Filter, BF_S is sample Bloom Filter, and $|BF_Q^1|$ and $|BF_S^1|$ are the total number of bits set to 1 in BF_Q and BF_S, respectively. Implementing Bloom Filter in cancelable biometrics performs data compression. The comparison between the biometric data is performed between the Bloom Filter, which is a compression version of the biometric data.

Among the four characteristics fulfilled by the cancelable biometric techniques, Bloom Filter can completely fulfill three characteristics and partially fulfill one characteristic. The mandatory characteristics are irreversibility, unlinkability, and performance, whereas a partial fulfilled characteristic is renewability. The binary vectors are mapped to the Bloom Filters. Bloom Filter is also a binary vector that returns whether a string is possibly present or definitely absent. However, it is incapable of constructing the original binary vector. The main reason is the collision of some strings to the same bit location, which leads to the loss of information about the original data. Thus, Bloom Filter fulfills irreversibility conditions. The time complexity of the Bloom Filter operation is $O(1)$. Therefore, executing operations of Bloom Filter is faster, which enhances the performance of cancelable biometrics. Unlinkability only requires changing the hash function. The same string will be inserted into different bit locations in a Bloom Filter by changing the hash function. Renewability requires deletion and construction of a new Bloom Filter. Bloom Filter has a simple data structure and the same Bloom Filter can be reused. Flushing of a Bloom Filter only requires a reset of bits to 0. Then the same Bloom Filter can be used to insert new strings. Thus, renewability explores the reusability of Bloom Filter.

The cancelable biometric techniques maintain many Bloom Filters. The number of Bloom Filters is equal to the number of blocks. If the block size is large, then more columns are present in each block. Each column is a string that is inserted into a Bloom Filter. Large insertions increase the FPP of Bloom Filter. In other words, more collisions occur in the Bloom Filter, which results in the loss of information about the biometric feature. If the small block's size is considered, the number of blocks increases, increasing the number of Bloom Filters. More Bloom Filters require more memory. Thus, small block sizes result in more memory overhead. Therefore, the technique has to decide on an optimal block size to have a good tradeoff between FPP and memory overhead.

Table 17.3 highlights a comparison of the techniques mentioned in the chapter based on various parameters. Bloom Filter-based biometric techniques maintain many Bloom Filters. These techniques should explore multi-set Bloom Filters. A multi-set Bloom Filter is a single Bloom Filter that stores strings of different sets. Using a multi-set Bloom Filter will reduce the total number of Bloom Filters required to maintain the biometric templates. Moreover, these techniques implement standard Bloom Filter without providing any additional scheme to reduce FPP. The main purpose of implementing Bloom Filter in biometrics is its fast execution of operations. But implementing standard Bloom Filter requires the execution of more hash functions to reduce FPP. However, it increases the execution time. Different Bloom Filter variants should be explored for better performance and efficiency.

17.5 Conclusion

This chapter discussed the role of Bloom Filter in biometrics. The biometric topics covered in the chapter are biometric indexing and cancelable biometrics. The biometric indexing explores Bloom Filter to construct a binary search tree for faster comparison of biometric query templates with the saved biometric templates. Similarly, cancelable biometrics also implements Bloom Filter for easy transformation of original biometric data into an irreversible form. However, both applications have to maintain many Bloom Filters due to the huge size of the biometric data. The chapter includes a review of Bloom Filter-based biometric techniques. Tables are provided that present the features and limitations of these techniques. All the reviewed techniques implement standard Bloom Filter, which has high

TABLE 17.3 Comparison of Bloom Filter-based Biometric Techniques: Variant, Bloom Filter variant; Reduce FP, technique used to reduce the number of false positives; Purpose, purpose of the technique.

Technique	Variant	Reduce FP	Biometric Characteristic	Purpose
Rathgeb et al. [13], 2015	Standard	X	Face, Iris	Privacy, Template protection
Rathgeb et al. [3], 2015	Standard	X	Iris	Indexing
Li et al. [14], 2015	Standard	X	Fingerprint	Template generation
Gomez et al. [8], 2016	Standard	X	Face	Template generation
Drozdowski et al. [6], 2017	Standard	X	Iris	Multi-Iris Indexing
Lai et al. [5], 2017	Standard	X	Iris	Indexing-First-One hashing
Sadhya and Singh [9], 2017	Standard	X	Iris	Template generation
Gomez-Barrero et al. [21], 2018	Standard	X	Multiple	Template generation
Qiu et al. [16], 2018	Standard	X	Palmprint	Cancelable biometrics
Drozdowski et al. [4], 2018	Standard	X	Iris	Template Searching
You and Li [18], 2018	Standard	X	Fingerprint, face	Cancelable biometrics
Shomaji et al. [7], 2019	Standard	Inserts concatenations of substrings	Face	Template Searching
Ajish and Anilkumar [22], 2020	Standard	X	Iris	Template generation
Kauba et al. [23], 2022	Standard	X	Finger Vein	Cancelable biometrics
Bansal and Garg [24], 2022	Standard	X	Multiple	Cancelable biometrics

FPP and requires the execution of more hash functions to reduce FPP. Hence, different Bloom Filter variants should be explored. Biometric techniques can also explore multi-set Bloom Filters to reduce the maintenance cost of many Bloom Filters.

References

[1] A. Pocovnicu, Biometric security for cell phones, Inform. Econ. 13 (1) (2009) 57–63.

[2] I. Verma, S.K. Jain, Biometrics security system: a review of multimodal biometrics based techniques for generating crypto-key, in: 2015 2nd International Conference on Computing for Sustainable Global Development (INDIACom), 2015, pp. 1189–1192.

[3] C. Rathgeb, F. Breitinger, H. Baier, C. Busch, Towards Bloom filter-based indexing of iris biometric data, in: 2015 International Conference on Biometrics (ICB), 2015, pp. 422–429.

[4] P. Drozdowski, S. Garg, C. Rathgeb, M. Gomez-Barrcro, D. Chang, C. Busch, Privacy-preserving indexing of iris-codes with cancelable Bloom filter-based search structures, in: 2018 26th European Signal Processing Conference (EUSIPCO), 2018, pp. 2360–2364.

[5] Y.-L. Lai, B.-M. Goi, T.-Y. Chai, Alignment-free indexing-first-one hashing with Bloom filter integration, in: 2017 IEEE International Conference on Intelligence and Security Informatics (ISI), 2017, pp. 78–82.

[6] P. Drozdowski, C. Rathgeb, C. Busch, Multi-iris indexing and retrieval: fusion strategies for Bloom filter-based search structures, in: 2017 IEEE International Joint Conference on Biometrics (IJCB), 2017, pp. 46–53.

[7] S. Shomaji, F. Ganji, D. Woodard, D. Forte, Hierarchical Bloom filter framework for security, space-efficiency, and rapid query handling in biometric systems, in: 2019 IEEE 10th International Conference on Biometrics Theory, Applications and Systems (BTAS), 2019, pp. 1–8.

[8] M. Gomez-Barrero, C. Rathgeb, J. Galbally, C. Busch, J. Fierrez, Unlinkable and irreversible biometric template protection based on Bloom filters, Inf. Sci. 370–371 (2016) 18–32, https://doi.org/10.1016/j.ins.2016.06.046.

[9] D. Sadhya, S.K. Singh, Providing robust security measures to Bloom filter based biometric template protection schemes, Comput. Secur. 67 (2017) 59–72, https://doi.org/10.1016/j.cose.2017.02.013.

[10] ISO/IEC 24745:2011, Information technology–security techniques–biometric information protection, International Organization for Standardization.

[11] C. Li, J. Hu, Attacks via record multiplicity on cancelable biometrics templates, Concurr. Comput., Pract. Exp. 26 (2014) 1593–1605.

[12] X. Zhou, S.D. Wolthusen, C. Busch, A. Kuijper, Feature correlation attack on biometric privacy protection schemes, in: 2009 Fifth International Conference on Intelligent Information Hiding and Multimedia Signal Processing, 2009, pp. 1061–1065.

[13] C. Rathgeb, M. Gomez-Barrero, C. Busch, J. Galbally, J. Fierrez, Towards cancelable multi-biometrics based on Bloom filters: a case study on feature level fusion of face and iris, in: 3rd International Workshop on Biometrics and Forensics (IWBF 2015), 2015, pp. 1–6.

[14] G. Li, B. Yang, C. Rathgeb, C. Busch, Towards generating protected fingerprint templates based on Bloom filters, in: 3rd International Workshop on Biometrics and Forensics (IWBF 2015), 2015, pp. 1–6.

[15] B. Yang, C. Busch, D. Gafurov, P. Bours, Renewable minutiae templates with tunable size and security, in: 2010 20th International Conference on Pattern Recognition, 2010, pp. 878–881.

[16] J. Qiu, H. Li, J. Dong, Generating cancelable palmprint templates based on Bloom filters, in: Proceedings of the 2018 6th International Conference on Bioinformatics and Computational Biology, ICBCB 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 78–82.

[17] D. Zhang, A. Kong, Competitive coding scheme for palmprint verification, in: Pattern Recognition, International Conference on, vol. 2, IEEE Computer Society, Los Alamitos, CA, USA, 2004, pp. 520–523.

[18] L. You, X. Li, A cancelable multi-biometric template generation algorithm based on Bloom filter, in: J. Vaidya, J. Li (Eds.), Algorithms and Architectures for Parallel Processing, Springer International Publishing, Cham, 2018, pp. 547–559.

[19] M. Sandhya, M.V. Prasad, *k*-nearest neighborhood structure (*k*-NNS) based alignment-free method for fingerprint template protection, in: 2015 International Conference on Biometrics (ICB), 2015, pp. 386–393.

[20] A. Teoh, A. Goh, D. Ngo, Random multispace quantization as an analytic mechanism for biohashing of biometric and random identity inputs, IEEE Trans. Pattern Anal. Mach. Intell. 28 (12) (2006) 1892–1901, https://doi.org/10.1109/TPAMI.2006.250.

[21] M. Gomez-Barrero, C. Rathgeb, G. Li, R. Ramachandra, J. Galbally, C. Busch, Multi-biometric template protection based on Bloom filters, Inf. Fusion 42 (2018) 37–50, https://doi.org/10.1016/j.inffus.2017.10.003.

[22] Ajish S, K.S. Anil Kumar, Iris template protection using double Bloom filter based feature transformation, Comput. Secur. 97 (2020) 101985, https://doi.org/10.1016/j.cose.2020.101985.

[23] C. Kauba, E. Piciucco, E. Maiorana, M. Gomez-Barrero, B. Prommegger, P. Campisi, A. Uhl, Towards practical cancelable biometrics for finger vein recognition, Inf. Sci. 585 (2022) 395–417, https://doi.org/10.1016/j.ins.2021.11.018.

[24] V. Bansal, S. Garg, A cancelable biometric identification scheme based on Bloom filter and format-preserving encryption, J. King Saud Univ, Comput. Inf. Sci. 34 (8) (2022) 5810–5821, https://doi.org/10.1016/j.jksuci.2022.01.014.

This page intentionally left blank

CHAPTER

18

Bloom Filter for bioinformatics

18.1 Introduction

Bioinformatics is the field of science that uses computation for a better understanding of biology. It is defined as

> "Bioinformatics is conceptualising biology in terms of macromolecules (in the sense of physical-chemistry) and then applying 'informatics' techniques (derived from disciplines such as applied mathematics, computer science, and statistics) to understand and organise the information associated with these molecules, on a large-scale" [1].

Many interdisciplinary fields are involved, such as computer science, mathematics, statistics, information engineering and biology. Biology gives the target of the study. For example, the target study is DNA. Computer science provides various algorithms or techniques, or technology for the analysis. For example, machine learning, Big data. In the case of the study of DNA, Big data is used to store the voluminous genomic data. Machine learning and data mining are used for DNA microarray analysis. Mathematics provides various mathematical theorems and formulas. For example, probability theory is used for predicting the probability of mutation in DNA. Information engineering helps in the understanding, usage, and manipulation of biological data.

Bioinformatics mainly deals with genes. Genomic data are voluminous. The size of the genomic database doubles [2] in every few months. Genomic data also contains a higher percentage of repeated regions [3]. In some cases distinguishing between the DNA sequences having very little difference is difficult. Hence, genomic data processing is memory- and task-intensive [4]. Handling such large scale and repetitive data requires an efficient data structure. The data structure should increase the performance and occupy very little memory space. Bloom Filter satisfies those criteria.

The attention of researchers is already towards the Bloom Filter. Many research works are proposed based on Bloom Filter. Many variants of Bloom Filter are proposed to handle genomic data with low false positive probability. Counting Quotient Filter [5] is a space-efficient and scalable Bloom Filter. It gives good efficiency in the case of highly skewed distributed input. Blocked Bloom Filter [6] is a cache-efficient variant of Bloom Filter, which is implemented in many techniques [7–9]. Bloom Filter Tree [10] is an extension of [11]. The extension is based on Bloom Filter, which is also used in many fields of bioinformatics such as DNA sequencing, genome annotation, pan-genomics, genetics of diseases, etc. DNA sequencing is a very long process consisting of many stages. Bloom Filter is used in many stages of DNA assembly. This chapter discusses the role of Bloom Filter in DNA assembly. The chapter also includes pre-processing techniques: k-mer counting, read compression, and error correction. Bloom Filter and variants of CBF are implemented to count the k-mers. Section 18.2.1 mentions the contribution of Bloom Filter in enhancing the performance of k-mer counting. Section 18.2.2 presents the working of Bloom Filter to reduce the big genomic data and makes the compression easier. Section 18.2.3 highlights the role of the Bloom Filter in removing errors in DNA sequences.

The chapter also elaborates on Bloom Filter's help in enhancing the DNA assembly indexing and searching in Section 18.4. Section 18.6 is dedicated to discussing the implementation of Bloom Filter in other fields of bioinformatics. Moreover, this chapter presents a brief review of the Bloom Filter-based techniques in each section. The main focus of this chapter is Bloom Filter; hence, the whole process is not explained.

Bloom Filter https://doi.org/10.1016/B978-0-12-823520-1.00025-0

Copyright © 2023 Elsevier Inc. All rights reserved.

18.2 Preprocessing filtering

The generation of the whole DNA sequence is impossible. Hence, short DNA fragments are generated which are called reads. However, the reads have many errors, namely overlapping DNA subsequences, ambiguous bases, indels, and substitutions [12,13]. An indel [14] is an insertion or deletion of a nucleotide base in a DNA sequence. Substitution is the exchange of a single nucleotide. Ambiguous bases are situations where the nucleotide is unknown. These errors are removed in the preprocessing filtering stage. This section includes three preprocessing filtering schemes: *k*-mer counting, read compression, and error correction. Also, this section presents a review of the Bloom Filter-based techniques to enhance the preprocessing filtering schemes.

18.2.1 *k*-mer counting

The speed of generation of genomic data is enhanced by generating short reads. The length of short reads is roughly 100 base pairs. However, the error is introduced due to underlying chemical and electrical processing [15]. One method of removing the error is repeated reading in the same region. Another method for solving the issue is overlapping the reads using coverage. The coverage is the number of reads within a particular region of the sequenced genome. Minimum 10× average coverage is needed for generating high accuracy sequences in a de novo assembly. Moreover, in the case of biased coverage, higher average coverage is required to obtain a representation of all regions [16]. However, both solutions have the same disadvantage of higher processing time. Thus, many assembling techniques implement the *k*-mer counting stage. The *k*-mer counting performs data reduction and error removal. Basically, in *k*-mer counting, the short reads are sliced into 20–70 base long *k*-mers. Then the algorithm counts the frequency of the *k*-mers. Usually, the *k*-mers that have frequency lower than a threshold value are removed. DNA sequences are repetitive, so less-occurring *k*-mers are considered erroneous *k*-mers. The *k*-mer counting step is very expensive because this step requires half of the total computation time in some assembling techniques. Table 18.1 highlights various features and limitations of the Bloom Filter-based *k*-mer counting techniques.

Melsted and Pritchard [17] presented a *k*-mer counting technique based on Bloom Filter and hash table. Bloom Filter is responsible for determining whether a *k*-mer is unique or not, and the hash table is responsible for storing *k*-mers. This *k*-mer counting technique stores the *k*-mer, which has occurred more than once. Initially, all slots of Bloom Filter are set to 0, and the hash table is empty. A *k*-mer of a fixed length is read and checked in the Bloom Filter. If the *k*-mer is absent from Bloom Filter, then the *k*-mer is inserted into Bloom Filter. In case the *k*-mer is present, then it is checked in the hash table. If it is present in the hash table, then the hash table increments the counter. If the *k*-mer is absent, then the *k*-mer is inserted into the hash table, and the counter is set to 2. This procedure helps in storing *k*-mers having more than one occurrence. However, a false positive *k*-mer gets inserted into the hash table.

Turtle [7] is a Bloom Filter-based cache-efficient *k*-mer counting tool to minimize the cache misses. It implements blocked Bloom Filter [6] and a sort-and-compact scheme to speed up the counting process. The blocked Bloom Filter localizes the bits set for a *k*-mer to a few consecutive bytes to decrease the cache misses. Turtle considers the *k*-mer and its reverse complement as two representations of the *k*-mer. Turtle has two tools, scTurtle and cTurtle. scTurtle is proposed to handle small-sized *k*-mer datasets, whereas cTurtle handles huge-sized *k*-mer datasets. scTurtle maintains a Bloom Filter and an array to store the *k*-mer and its frequency. It follows a single producer and multiple consumers model with a pool of buffers. The producer is responsible for reading *k*-mers. Each consumer has a Bloom Filter and an array. When a *k*-mer is read, first, it is checked in Bloom Filter. If absent, it is stored in the Bloom Filter. If present, it is stored in the array, and the count is set to 1. When the number of *k*-mers in the array exceeds a threshold value, the sort-and-compact technique is executed to sort the array. It places the same *k*-mers in consecutive slots. After sorting, a linear traversal is performed on the array to remove multiple similar *k*-mers and add up all the frequencies under the single *k*-mer instance. cTurtle implements CBF. After reading the *k*-mer, it is checked in the CBF. If absent, the algorithm inserts it into CBF. If present, then the *k*-mer is written to the disk.

Mcvicar et al. [15] proposed a *k*-mer counting technique based on the FPGA-attached Hybrid Memory Cube (HMC) [18]. It implements CBF [19]. It uses four FPGAs, and each FPGA maintains a small Bloom Filter. The huge number of small atomic operations generated by the Bloom Filter is efficiently executed by HMC. The independent operations are performed in parallel. The *k*-mer is hashed by shift-add-XOR (SAX) hash functions and kept in a bypass FIFO to output. The hashed value is used to check the Bloom Filter. If found, the *k*-mer is inserted into the CBF. The technique maintains a buffer of addresses to stall any process attempting to access a Bloom Filter already accessed by another process. After the completion of Bloom Filter construction, Bloom Filter determines the minimum frequency of the *k*-mer and updates the unsaturated counters. These addresses are also removed from the

TABLE 18.1 Features and Limitations of Bloom Filter-based k-mer Counting Techniques.

Technique	Features	Limitations
Melsted and Pritchard [17]	• k-mer counting technique based on Bloom Filter and hash table • Reduces memory consumption • Bloom Filter determines whether a k-mer is unique or not • Hash table stores k-mers • Hash table keeps the count of the k-mer frequency • Once occurring k-mer is stored only in Bloom Filter • Count of the k-mer in hash table starts from 2	• Due to false positive response of Bloom Filter, the k-mers get stored in hash table • Implementing multiple threads on single Bloom Filter increases errors • Implements standard Bloom Filter
Turtle [7]	• Bloom Filter-based cache-efficient k-mer counting tool to minimize the cache misses • Balances time, space, and accuracy requirements to increase efficiency • Localizes the bit set of an item to a few consecutive bytes to reduce cache misses • k-mers are stored in extra buffer in case the processing of k-mers takes long time • More lazy compactions within a phase reduces the number of phases • Bloom Filter of scTurtle does not give a false negative response • cTurtle is more memory efficient	• Unable to provide exact k-mer counts in a large dataset • Multiple Bloom Filters are maintained, which increases memory consumption • Producer of k-mers should be faster than all processors (consumers) of k-mers • Sorting of array increases computational cost • Bloom Filter of scTurtle gives a false positive response • CBF of cTurtle gives both false positive and false negative responses • cTurtle does not provide k-mer frequency count
Mcvicar et al. [15]	• HMC increases parallel computation • HMC provides high bandwidth • HMC improves random-access performance • Bloom Filters generate large volumes of small, random accesses • Bloom Filter generates independent memory operations • Gives double performance compared to parallel CPU implementation	• Random-access bandwidth must be more than the host access bandwidth • Representation must be compact • HMC gives bad performance in case of many operations on a small dataset • HMC maintains four separate address spaces, which reduces accessed memory space • Maintains many Bloom Filters • One large Bloom Filter gives bad performance compared to several small Bloom Filters • Collision causes stalling of addresses present in the buffer • Overflow of counter occurs in CBF • Large k-mer length with low bandwidth gives bad performance
Squeakr [20]	• Uses a single off-the-shelf data structure (i.e., CQF) with a straightforward system design • Processes a lock-free queue • Global CQF executes a single operation on many k-mers • Each thread maintains a local CQF • Saturated local CQF is written to the global CQF • Uses thread-safe CQF • Thread-safe reading and parsing improves scalability of threads	• During de Bruijn graph traversal, Squeakr performs four queries for each k-mer • Squeakr spends three-fourths of time in processing false query operations • Thread-safe CQF degrades the thread scalability in a larger highly-skewed dataset • Large dataset with high skewness contains k-mer with high multiplicity and causes hotspot issue • Hotspot causes high lock contention among threads • Squeakr-exact supports k-mer having maximum length of 32 bases

continued on next page

buffer. Finally, Bloom Filter evaluated the actual frequency of the k-mers by calculating the total frequency stored in all FIFO. This technique provides double performance compared to parallel CPUs.

Squeakr (Simple Quotient filter-based Exact and Approximate k-mer Representation) [20] is an in-memory k-mer counter based on the counting quotient filter (CQF) [5]. Thread is used in reading DNA reads from the disk, parsing them to extract the k-mers and inserting them into CQF. Squeakr has two types of CQF, local and global. Global CQF executes a single operation on many k-mers. Each thread maintains a local CQF. When the local CQF becomes saturated, it is written to the global CQF. In the case of a smaller low-skewed dataset, single thread-safe CQF helps in

TABLE 18.1 (*continued*)

Technique	Features	Limitations
kmcEx [21]	• Bloom Filter variant proposed for efficient and faster *k*-mer counting • Each slot is of two bits • The value of the bit location in the frequency array is based on the binary form of the frequency • When a collision occurs in the first array, a new set of pair arrays are constructed to insert the new *k*-mer • Checks neighbors of the *k*-mer to resolve FPP issues • In case of presence of a *k*-mer in multiple kmcEx, the frequency close to the neighboring *k*-mers is selected • A standard Bloom Filter stores the *k*-mers having frequency 1 • Large frequency is represented by clustering the frequencies to use fewer bits	• Collision leads to construction of new pair arrays • Using neighboring *k*-mers to reduce FPP increases the number of operations • The number of hash functions is same as the number of binary representations of the frequency • Large frequency cannot be considered to maintain good trade-off between the number of hash functions and FPP
CQF-deNoise [22]	• A *k*-mer counting method that dynamically filters the incorrect *k*-mers • Maintains the low frequency of the *k*-mers • Improves the CQF • Fewer bits are required per *k*-mer compared to CQF	• Incorrect *k*-mer count is saved • Performance depends on the number of false *k*-mer removal rounds
SWAPCounter [23]	• Highly scalable distributed *k*-mer counting scheme • Uses MPI streaming I/O module for faster loading of dataset • CBF improves memory and communication efficiency	• Dataset reading, *k*-mer extraction, distribution, and filtering are the most time consuming modules • Maintains multiple CBFs • Does not handle the false positive responses generated by CBF
kmtricks [24]	• Technique that performs joint *k*-mer counting across many samples • Takes multiple samples as input • Joint *k*-mer counting retrieves more *k*-mers, which could be ignored in case they are processed independently • Counting of hash values is faster compared to counting of *k*-mers • Good performance in case of big sample size	• Maintains multiple Bloom Filters • Constructs many intermediate Bloom Filters • Implements standard Bloom Filter • Bad performance in case of small sample size • FPP is partition dependent
KCOSS [25]	• A *k*-mer counting technique • Follows a single producer and multiple consumer strategy • Producer reads the DNA data and consumer are nodes responsible for processing of data • Nucleotides converted into binary format reduce memory requirements • Hash table is big compared to elastic hash table	• Maintains many data structures: segmented Bloom Filter, hash table, and elastic hash table • Access of elastic hash table is low • Implements standard Bloom Filter

scaling the incrementing number of threads. On the other hand, it cannot scale in case of highly-skewed datasets. A higher number of threads decreases the performance. The highly repetitive *k*-mers require acquiring the same locks repetitively and reduce the performance.

kmcEx [21] is a Bloom Filter variant proposed for efficient and faster *k*-mer counting. It consists of two arrays. The first array stores the *k*-mers, and the second stores the frequency of the *k*-mers. The *k*-mer is hashed by hash functions, and the hashed values are the bit locations of the first array, which are set to 1. For the second array, the same hashed values provide the bit locations. The value of the bit location is based on the binary form of the frequency. The frequency of the *k*-mer is represented in a binary form. The bit value of the binary form is assigned to the corresponding hash location. Hence, the value of the bit location can be 0. During the query operation, the first array is used to check the presence of the *k*-mer, and the second is used to retrieve its frequency. The FPP is handled by constructing multiple kmcEx. When a collision occurs in the first array, then a new set of pair arrays are constructed to insert the new *k*-mer. kmcEx also uses checking of neighbors of the *k*-mer to resolve FPP issues. In case when a *k*-mer is not inserted but present in kmcEx, the neighboring *k*-mers are queried. If absent, then the *k*-mer is non-existent, and the *k*-mer is ignored. Another issue is determining the frequency of a *k*-mer present in multiple kmcEx. The frequency of the neighboring *k*-mers is checked. The frequency which is close to the neighboring *k*-mers

is selected. Large frequency cannot be considered to maintain a good trade-off between the number of hash functions and FPP. This issue is solved by grouping the frequency into various clusters. This helps in representing the cluster using fewer bits. Moreover, a standard Bloom Filter is implemented to store the k-mers having frequency 1.

Counting Quotient Filter deNoise (CQF-deNoise) [22] is a k-mer counting method that dynamically filters the incorrect k-mers. It maintains the low frequency of the k-mers. CQF is a multiset Bloom Filter variant. CQF has two Bloom Filters. One stores the q bits and the other bits are stored in another Bloom Filter. When the frequency of a k-mer is 1, a single slot is used to store the remaining bits of the k-mer. In case the frequency of a k-mer is more than 1, multiple slots are used; the first slot stores the remaining bits and the other slots store the counter value. The user defines a threshold value which is the limit of the frequency count of a k-mer. After a round of noise removal, the k-mer having one count is removed.

SWAPCounter [23] is a highly scalable distributed k-mer counting scheme. It uses MPI streaming I/O module for faster loading of dataset and CBF to improve memory and communication efficiency. It has four modules: I/O, k-mer extraction and distribution, k-mer filtering, and counting and statistics. To increase performance, all four modules are pipelined by multiple instances. The I/O module is responsible for reading the DNA reads. The k-mer extraction and distribution are responsible for extracting k-mers of a particular length from the reads. Then the k-mers are forwarded to the next module, where they are distributed to different nodes. The k-mer filtering module counts the k-mers and stores them in different data structures. The counting and statistics module uses the information stored in the data structures to prepare a histogram of the unique k-mers. The third module, i.e., the k-mer filter, requires CBF. Three data structures are maintained: a hash table, a container, and a CBF. All unique k-mers are stored in the hash table. The trustworthy k-mers are stored in the container. At the same time, CBF keeps the count of the frequency of the k-mers. After receiving the k-mers from the previous module, the unique k-mer is inserted into the hash table. CBF sets the bit value to 1 and counter to 1. When repeated k-mers are received, CBF only increments the counter. When the frequency of a k-mer crosses a threshold, the k-mer is inserted into the container of the trustworthy k-mers.

The k-mer Matrix Tricks (kmtricks) [24] proposed a technique that performs joint k-mer counting across many samples. Multiple samples are forwarded as input to kmtricks. The k-mers of particular length are extracted from the sample DNA reads. Some common k-mers are grouped together based on their similarity. They are called minimizers and are represented by a symbol. The partitions have the same number of k-mers. Then super-k-mers are written to the disk where the DNA reads are replaced by the minimizers. A super-k-mer is hashed. The unique hashes are counted and stored in the disk. Then the algorithm aggregates the counted hashes that belong to equivalent partitions. Merging is performed by executing the n-way merge algorithm, where the count of the hashes is used for sorting. The counts are inserted into a Bloom Filter. A Bloom Filter is constructed for each equivalent partition. Then the Bloom Filters are transposed. The transposed Bloom Filters are concatenated to generate a single Bloom Filter for each sample.

KCOSS [25] is a k-mer counting technique based on segmented Bloom Filter, lock-free thread pool, lock-free queue, and cuckoo hash table. It follows a single producer and multiple consumer strategy. A single producer reads DNA reads from the files and partitions them into blocks. The blocks are distributed among multiple lock-free queues. The consumer fetches the blocks and maps the single nucleotide into a binary format. Letters A, C, G, and T are mapped to 00, 01, 10, and 11, respectively. When the frequency of a k-mer is less than 14, a hash table is used to store the k-mer. However, if the frequency is more than 14, then a lock-free segmented Bloom Filter is used to store the k-mer. KCOSS constructs a set called an overlapping sequence set. In that set, all the single occurrence k-mers are inserted. When a block is full, the block is written to the disk. KCOSS partitions the heap memory into segments of the same length. Each segment is a Bloom Filter, and the whole is called a segmented Bloom Filter. When a k-mer is read, first, it is checked in the segmented Bloom Filter. The k-mer is hashed by multiple hash functions. The hashed value of the first hash function provides the segment where the k-mer needs to be searched. Other hashed values provide the bit locations of the Bloom Filter. If at least one-bit location of the Bloom Filters is not 1, then the response is $False$. If not found, then it is inserted into the sequence set and Bloom Filter. The multiply-occurring k-mer is stored in two hash tables, a hash table and an elastic hash table. When a k-mer is present in Bloom Filter, it is searched in the hash table. If it is not found in the hash table, then it is searched in the elastic hash table.

18.2.2 Read compression

A large-scale DNA assembly project generates tens to several thousand of genomes per species. It leads to the storage and transfer of genomic data. Storage and its maintenance are important because genomic data is required for future research work. The big-sized data is also a hindrance to the collaboration between research teams [26].

National Centre for Biotechnology Information (NCBI) in the US withdrew its support from Sequence Read Archive (SRA) and declared termination of the data submission to SRA in 2011. One of the main reasons was the big genomic data. Its huge size made it difficult to download, access, and transmit [27]. Therefore, read compression is important for smooth data transfer and storage for a long time. Table 18.2 presents various features and limitations of the Bloom Filter-based read compression techniques.

TABLE 18.2 Features and Limitations of Bloom Filter-based Read Compression Techniques.

Technique	Features	Limitations
BARCODE [28]	• Cascading Bloom Filters based DNA sequence compression method • Cascading Bloom Filter can differentiate between true positive and false positive • Cascading Bloom Filter stores the false positive reads in another Bloom Filter • Reads are inserted into Bloom Filters as a means of compression • Compresses more effectively at higher coverage	• Maintains multiple Bloom Filters • Increase in k-mer length increases computation time
LEON [26]	• Bloom Filter-based compression method which performs lossless sequence compression and lossy quality compression • Bloom Filter is used for quicker and memory-efficient graph construction • Reference is represented by de Bruijn graph • Bloom Filter stores the large data structure of de Bruijn graph • Reuses same anchor k-mer for different reads • Maintains good performance in case of multiple datasets	• Optimal Bloom filter size depends on the depth of sequencing • Depends on k-mer counting technique to reduce the number of reads • Compression ratio depends on the quality of the reference • Implements standard Bloom Filter
DARRC [30]	• A Bloom Filter-based read compression method • Guided de Bruijn graph allows a unique traversal of sequences • Partitioned k-mers are encoded to reduce memory • SSR encoding utilizes Bloom Filter for determining an optimal node for gdBG • Partition integers are recycled • gdBG is compressed with Zstd compression method which prefers compression and decompression speed over compression ratio	• Does not perform parallel computation • de Bruijn graph requires long computation • Implements standard Bloom Filter

Bloom filter Alignment-free Reference-based COmpression and DEcompression (BARCODE) [28] is a cascading Bloom Filter-based [29] DNA sequence compression method. It is assumed that all reads are unique during the encoding phase. All reads are inserted into a cascading Bloom Filter, which helps in determining the unique reads. Two sets are maintained. One set stores reads for which the cascading Bloom Filter returned false positives. Another set stores mutated reads, or instances where some error occurred while reading a string. The cascading Bloom Filter also has two Bloom Filters. One of them stores the reads of the false positive read set. Another Bloom Filter stores the reads of the erroneous read set. During read insertion, first, the read is checked in the false positive Bloom Filter. If not present, then the read is inserted into the cascading Bloom Filter. After the completion of the construction of cascading Bloom Filter, all reads are queried to Bloom Filters to determine whether the read is unique. After completion of the encoding phase, all Bloom Filters are compressed using an off-the-shelf compression tool.

LEON [26] is a Bloom Filter-based compression method which performs lossless sequence compression and lossy quality compression. Bloom Filter is used for quicker and memory-efficient graph construction. LEON first constructed a de Bruijn graph. Considering a small Bloom Filter reduces compression memory but increases the storage space of each read. Hence, the Bloom Filter size depends on the depth of sequencing. The solid k-mers are inserted into the Bloom Filter.

Dynamic Alignment-free and RefeRence-free Compression (DARRC) [30] is a Bloom Filter-based read compression method. It is extended from ORCOM read sequencing [31]. The data goes through four steps for compression: preprocessing, Spanning Super Read (SSR) encoding, partition encoding, and metadata and guided de Bruijn graph (gdBG) compression. SSR encoding utilizes Bloom Filter for determining an optimal node for the de Bruijn graph. DARRC uses gdBG which is a variant of de Bruijn graph. The k-mer extraction path requires an optimal node for

starting (say, *start position*). From the *start position*, the path is followed to construct the gdBG. Bloom Filter helps in comparing various *start position* nodes before selection. After selection of the *start position*, it is inserted into the Bloom Filter.

18.2.3 Error correction

Many errors get introduced during genomic data collection or generation. Errors are due to various reasons such as insertion of a new DNA fragment or omission of any DNA fragment [12,13]. The presence of an error reduces the data quality. Hence, error correction is the most important step in all preprocessing processes. This stage removes errors for the implementation of DNA assembly on correct reads. However, it has a long processing time. The error correction methods are executed in parallel to speed up the process because errors are independent of each other. One big issue with error correction methods is that a single method is unable to remove all errors due to the diverse nature of the errors. Table 18.3 illustrates the various features and limitations of the Bloom Filter-based error correction techniques.

TABLE 18.3 Features and Limitations of Bloom Filter-based Error Correction Techniques.

Technique	Features	Limitations
BLESS [32]	• Bloom Filter stores all solid k-mers for faster membership checking • Extends the reads to find multiple k-mers that cover the erroneous bases at the end of the reads to improve error correction at the end of the reads • Converts weak k-mer to solid k-mer • False positive response is identified by checking with a hash table	• Increase in depth of sequencing increases the disk space occupancy • False positive response of Bloom Filter may lead to change an erroneous base to a wrong one • Identifying false positive response increases time complexity • If the threshold of k-mer frequency is high, then the correct k-mer is determined as weak k-mer • Very small k leads to many unnecessary paths in the error correction process • Large k decreases frequency of solid k-mers • In the counting step, k-mers are distributed into N files and occupy more memory
Lighter [9]	• Implements pattern-blocked Bloom Filter • Error correction accuracy is independent of depth of sequencing • Bloom Filter size is independent of depth of sequencing • False positive probability is independent of depth of sequencing • Pattern-blocked Bloom filter decreases the overall number of cache misses • Bloom Filter reduces synchronization process • Requires no temporary disk space	• Technique requires a big-sized Bloom Filter • Avoids error present towards very last k-mer • Correction of solid k-mers having frequency less than the threshold decreases accuracy • Increase in subsampling probability increases threshold value, which decreases accuracy • Lighter cannibalizes k-mers, which loses information regarding whether an error tends to occur on one strand or the other
FADE [8]	• Blocked version of the CBF increases the computational speed • LFSR-based hash function helps to pipeline the hash functions • Multiple hash functions can be generated by simply changing the LFSR taps • Multiple-level arbitration scheme improves parallelism	• Blocked version of the CBF increases false positive probability • Large input to hash function reduces accuracy due to collision • Large number of hash functions on a smaller block increases the false positive probability

Bloom-filter-based Error correction Solution for high-throughput sequencing reads (BLESS) [32] is a standard Bloom Filter-based method having higher tolerance of FPP. The first step of BLESS is counting the frequency of each k-mer. However, the k-mer counting step is time-consuming and takes the most time in the entire processing. Thus, BLESS requires an efficient k-mer counting technique. Among the k-mers, solid k-mers are found and stored in a hash table. All solid k-mers are also inserted into the Bloom Filter. Bloom Filter size is equal to the total number of solid k-mers. Determining all solid k-mers is a difficult task; hence, determining the Bloom Filter size is difficult. Bloom Filter helps in the conversion of solid k-mers into weak k-mers. BLESS also handles the false positive response by removing the correction results performed because of it. In the case of k-mers with modified bases, the k-mers

are checked in the hash table. If not found, it is concluded that it is caused due to a false positive response of the Bloom Filter. BLESS does not perform parallel execution in many processes. BLESS performs k-mer counting, error correction, and k-mer distribution to files in parallel to lower the time complexity of the whole process.

LIGHTweight ERror corrector (Lighter) [9] is a Bloom Filter-based memory-efficient tool for correcting sequence errors. It implements pattern-blocked Bloom Filter [6] for error correction. Bloom Filter lowers the number of cache misses to enhance efficiency. Cannibalized k-mers are inserted into the Bloom Filter. The cannibalized k-mers are the k-mer itself or its reverse complement based on lexicographical order. In Lighter, the input reads are forwarded through three passes. The first pass reads the DNA reads from a sample input and inserts it into a Bloom Filter. In the second pass, the solid k-mers are filtered using the first-pass Bloom Filter. These solid k-mers are inserted into another Bloom Filter. Error correction is based on trusted and untrusted k-mer positions. When a k-mer is present in the first pass Bloom Filter, and the number of k-mers overlapping the position is lower than a fixed threshold value, then the position is called untrusted. If the position does not satisfy any one condition, then it is trusted. All trusted k-mers are inserted into the second pass Bloom Filter. In the final pass, a greedy error correction method is executed, which is similar to BLESS [32]. The error correction algorithm takes the help of both the Bloom Filters.

FPGA Accelerated DNA Error Correction (FADE) [8] is an FPGA-based error correction tool for Illumina reads. Illumina reads are for paired-end sequencing. Paired-end sequencing performs sequencing in both ends of a fragment to generate alignable, high-quality sequence data. FADE implements the BLESS [32] error correction method because it is the most accurate algorithm for Illumina reads. FADE uses CBF [19] which is constructed in DDR3 memory. CBF is implemented as a blocked Bloom Filter [6] to lower the number of DDR3 memory accesses. The generated k-mers are inserted into the CBF.

18.3 de Bruijn graph

The de Bruijn graph [33] is a directed graph that represents the overlaps between the nucleotides in a genomic sequence. The graph is named after Nicolaas Govert de Bruijn. Determining the length of the k-mer is the first and most important step of the construction of the standard de Bruijn graph. The shotgun fragments are partitioned into reads of fixed length. Two $(k-1)$-mers are generated from each read. The first $(k-1)$-mer is generated from the first nucleotide, and the second $(k-1)$-mer from the second nucleotide. Then each unique $(k-1)$-mer is a vertex of the de Bruijn graph. An edge exists from the first $(k-1)$-mer to the second $(k-1)$-mer. Multiple edges exist between the vertices to represent the repetition of the k-mer in the genomic sequence. Considering the case of perfect sequencing, the time complexity of the de Bruijn graph construction is $O(N)$, where N is the length of the sequencing. Perfect sequencing is the scenario where a read occurs once in an error-free sequence. There are two main issues in the de Bruijn graph. First, there is only a single correct sequence among multiple Eulerian walks in a de Bruijn graph. It is impossible to determine the correct sequence in de novo assembly. The second issue is the presence of gaps and coverage in a sequence, which results in a disconnected graph.

Bloom Filter is gaining preference in the construction of de Bruijn graphs. Currently, the de Bruijn graph is used for the representation of genome, and transcriptome assembling [34]. The compacted de Bruijn graph is ideal for the representation and indexing of repetitive sequences. It stores the k-mers once in the graph in the set of non-branching and unique paths [33]. Query operation in the graph requires huge memory. Storing the de Bruijn graph in a hash table requires 8 bytes for a k-mer, 4 to 8 bytes for an edge, and 4 to 8 bytes for an offset. Also, the query operation requires many additional data structures. To address these issues, the focus is shifted to Bloom Filter for implementation in the de Bruijn graph. Bloom Filter easily speeds up the query operation in the de Bruijn graph. The following section presents a review of the Bloom Filter-based de Bruijn graph construction techniques, and Table 18.4 highlights the features and limitations of these techniques.

Salikhov et al. [29] proposed a cascading Bloom Filter-based method to represent the de Bruijn graph. The k-mers, which are the nodes of the graph, are inserted into the Bloom Filter. This method reduces RAM usage by incurring more disk accesses. The genomic reads are read from the disk and inserted into a Bloom Filter. Another Bloom Filter is constructed, which inserts the k-mers from the previous Bloom Filter successively. The extension of each k-mer inserted into the second Bloom Filter is checked in the first Bloom Filter. If found, the k-mer is written to the disk. Otherwise, it is inserted into the first Bloom Filter by performing set difference. The set difference operation is performed partially on the first Bloom Filter by loading the Bloom Filter partially in RAM. After the set difference operation, the k-mers in the second Bloom Filter are the critical false positives. Following a similar procedure, k-mers from the second Bloom Filter are inserted into the third Bloom Filter. This procedure is repeated a number of times

TABLE 18.4 Features and Limitations of Bloom Filter-based de Bruijn Graph Construction Techniques.

Technique	Features	Limitations
Salikhov et al. [29]	• Cascading Bloom Filters-based method to represent the de Bruijn graph • All k-mers are stored in the disk • After set difference, one Bloom Filter contains critical false positive k-mers • k-mers are sorted externally and a binary search is performed to reduce construction time	• Reduces RAM usage but increases the number of disk accesses • Continuous interaction with the disk increases the processing time • Performance in a large dataset is poor due to a large number of disk accesses • Operations on the Bloom Filter are executed on partially loaded Bloom Filter in RAM • Maintains many Bloom Filters • Implements standard Bloom Filter
deBGR [35]	• A counting quotient filter (CQF) based method to exactly represent the compacted weighted de Bruijn Graph • Implements Squeakr [20] for k-mer counting • CQF helps in implementing a simple traversal algorithm for in-memory creation of the weighted de Bruijn graph • Graph is simple, manageable, and completely present in RAM • A CQF stores the edges involved in the cycle	• Higher space complexity • False positive responses of Bloom Filter cause incorrect topology • False positive responses misestimate the value of abundance
Faucet [36]	• Two-pass streaming algorithm for compacted de Bruijn graph construction • Maintains two Bloom Filters, but discards one of them after completion of the scanning of the k-mers • k-mers stored in the second Bloom Filter are junction nodes • Hashmap maps a k-mer to a vector that stores the count of the k-mer	• Counting is performed by a vector • A variant of a CBF is not used to store the count of the k-mer • Implements standard Bloom Filter

which is obtained experimentally. All k-mers are stored in the disk; k-mers are sorted externally and a binary search is performed to reduce construction time.

deBGR [35] is a counting quotient filter (CQF) based method to exactly represent the compacted weighted de Bruijn graph. The weighted de Bruijn graph assigns weight on each edge to reduce the number of repeated edges between two vertices. It implements Squeakr [20] for k-mer counting. CQF helps in implementing a simple traversal algorithm for the in-memory creation of the weighted de Bruijn graph. The graph is simple, manageable, and completely present in RAM. False positive responses of Bloom Filter cause incorrect topology and misestimate abundance value. deBGR executes an algorithm to solve these problems. The de Bruijn graph contains cycles of small length which interfere in the correct estimation of the occurrence of an edge. deBGR constructs a CQF to store the edges involved in the cycle. deBGR is appropriate for the applications that require static and navigational weighted de Bruijn graph. deBGR has high space complexity.

Faucet [36] is a two-pass streaming algorithm for compacted de Bruijn graph construction. Faucet processes each k-mer and constructs the de Bruijn graph incrementally. It maintains two Bloom Filters. When a k-mer is read, it is first checked with the first Bloom Filter. If found, the k-mer is inserted into the second Bloom Filter. Otherwise, the algorithm inserts the k-mer into the first Bloom Filter. The first Bloom Filter is removed after completion of the scanning of the k-mers. The k-mers stored in the second Bloom Filter are junction nodes. A k-mer is called a junction node when it has an in- or out-degree more than one or has at least one extension that differs from the next base on the k-mer. Then all k-mers are stored in a hashmap. The hashmap maps the k-mer to a vector that stores the count of the k-mer. Bloom Filter is also used to store the adjacency between the pairs of k-mers. The compacted de Bruijn graph is constructed by first traversing each forward extension for every special k-mer. Then the algorithm traverses in reverse complement direction. In case a node is not reached, the traversing is again started from a new node. Faucet queries the second Bloom Filter for the k-mers. This procedure is followed till reaching a new special node.

18.4 Searching

Public databases that store sequencing data, such as Sequence Read Archive (SRA) and European Nucleotide Archive (ENA), have become the fundamental shared community resource for sequence analysis. The data in these databases are growing exponentially; for instance, the raw reads of SRA have reached 35 Peta-bases [37]. Various issues associated with exponential data growth are scalability, storage, etc. The public database also has to address efficient processing and sequence analysis pipelines. Moreover, there is a requirement for high performance in data parallelism, compression, and concurrent computing. The main focus of these public databases is quick and accurate searching. However, searching for any specific genomic sequence is a very difficult task due to the overwhelming computational time. Usually, the searching is performed on the metadata or retrieving result data obtained by a recently executed search on an experiment. Thus, the searching techniques are unable to provide searching functionalities. This section presents the role of the Bloom Filter in reducing the search complexity of the searching techniques. This section also includes a review of the Bloom Filter-based searching techniques. Table 18.5 illustrates the features and limitations of Bloom Filter-based searching techniques.

TABLE 18.5 Features and Limitations of Bloom Filter-based Searching Techniques.

Technique	Features	Limitations
SSBT [38]	• An indexing scheme for efficient searching of *k*-mers in the experiments • SSBT is a binary tree consisting of compressed Bloom Filters • SSBT is capable of searching both known and arbitrary *k*-mers • SSBT consists of compressed Bloom Filters • Each node consists of two identical Bloom Filters, similarity and remainder filters • Intersection of similarity and remainder filters is null • Searching a similar leaf Bloom Filter helps to keep a minimum-sized SSBT	• Each node has two Bloom Filters • Each node processing during insertion of a new Bloom Filter involves changes in similarity filter, remainder filter, and new Bloom Filter • SSBT is a tree of Bloom Filters which contains twice as many as nodes Bloom Filters • Construction time of SSBT is three times slower than SBT
MetaProFi [39]	• *k*-mer indexing tool based on Bloom Filter for amino acid sequences • Constructs a 2-dimensional Bloom Filter matrix where a row stores the hash indices and each column is a Bloom Filter of different samples • User defines the memory MetaProFi is permitted to occupy • Bloom Filter matrix is constructed using POSIX shared memory • POSIX shared memory shares the Bloom Filter matrix among the processes	• Periodic synchronization of the Bloom Filter matrix is essential as it is shared between various processes • Implements standard Bloom Filter • New *k*-mers are stored in newly constructed Bloom Filter • Multiple Bloom Filters are maintained in secondary storage
PAC [40]	• A comb tree of Bloom Filters for efficient searching of sequence reads • Consists of two binary comb trees, left and right comb trees • Enhances cache coherence • Insertion of a new Bloom Filter does not change the index structure • Different partition PAC is handled by different threads which enhances parallel computation • Partitions that do not include the query *k*-mers are ignored	• Search of a single *k*-mer requires the construction of an inverted hash value mapping table • Searching of a sequence requires much preprocessing

Split Sequence Bloom Tree (SSBT) [38] is an indexing scheme for efficient searching of *k*-mers present in various experiments. SSBT is the improvement of Sequence Bloom Tree (SBT). SSBT is a binary tree consisting of compressed Bloom Filters. Each node in SSBT consists of two Bloom Filters. The pair of Bloom Filters are called split Bloom Filter. The first Bloom Filter is called the similarity filter, and the other is called the remainder filter. In the root node, the similarity filter contains the union of all *k*-mers present in all Bloom Filters. The remainder filter contains the rest of the *k*-mers absent from the similarity filter. The intersection of similarity and remainder filter is null. Both

similarity and remainder filters have the same size. The root node passes the remainder filter for the construction of its children. During insertion of a node/Bloom Filter in SSBT, the k-mers are inserted into a newly constructed Bloom Filter. The new Bloom Filter is inserted as a leaf in the SSBT by traversing from the root. When reaching a node, the new Bloom Filter is partitioned and inserted into the similarity and remainder filter. Each bit of both filters is updated by following a predefined table that provides changes depending on the value of both filters and the new Bloom Filter. A similar procedure is followed to update all traversed nodes. This update also updates the new Bloom Filter based on the bit values of the similarity and remainder filter. The updated new Bloom Filter is checked against another leaf Bloom Filters to find the most similar Bloom Filter. The similarity is determined by the number and the position of 1s. This similarity helps in choosing the path in SSBT. During a query operation, a Bloom Filter is constructed by inserting the queried k-mers. The Bloom Filter is matched with the similarity filter of the root. If some k-mers are found, then the k-mers are removed from the new Bloom Filter. The rest are searched in the remainder filter. If the number of k-mer matches is more than a threshold, then the SSBT is searched further; otherwise, the searching operation is terminated.

MetaProFi [39] is a k-mer indexing tool based on Bloom Filter for amino acid sequences. It implements a packed Bloom Filter, which combines various Bloom Filter variants. It uses chunked data storage and Zstandard compression. It constructs a 2-dimensional Bloom Filter matrix where a row stores the hash indices, and each column is a Bloom Filter of different samples. The user defines the memory MetaProFi is permitted to occupy. The input samples are partitioned into different subsamples to keep the memory requirement of MetaProFi within limits. The 2-dimensional Bloom Filter matrix is constructed using POSIX shared memory, which is used for efficient interprocess communication. It also helps to share the Bloom Filter matrix among the processes. After insertion of the k-mers into the Bloom Filter matrix, the matrix is compressed using the standard compression algorithm. It further reduces memory occupancy. Then it is written to the disk. For indexing, MetaProFi constructs a 2D POSIX shared memory matrix where each row refers to a column of the Bloom Filter matrix. The number of rows is equal to the number of subsamples. A column is read from the Bloom Filter matrix and compressed to reduce the memory size. Then it is stored in the index matrix. In the case of querying multiple sequences, MetaProFi retrieves k-mers from the file and performs hashing. The hashed value is inserted into a Bloom Filter. Then the hash value is used to retrieve the index from the index matrix.

Partitioned Aggregative Bloom Comb (PAC) trees [40] is a comb tree of Bloom Filters for efficient searching of sequence reads. The k-mers of each experiment are inserted into Bloom Filters. These Bloom Filters are organized to construct the comb trees. The comb tree consists of two binary Bloom comb trees, left and right comb trees. Each node in PAC is a Bloom Filter. In the binary Bloom left comb tree, the leftmost node is the union of the first two Bloom Filters. In PAC, each node is the union of one internal node and a leaf node. The Bloom right comb tree is symmetric to the Bloom left comb tree. The Aggregative Bloom comb tree consists of the Bloom left and right comb trees. The comb tree is represented by aggregative Bloom Filters and BF Bloom Filters. The aggregative Bloom Filter contains the branch passing through internal nodes till the lowest leaf node. The BF Bloom Filters are the leaf nodes of both Bloom comb trees. PAC has a aggregative Bloom Filters constructed for each partition. A query of a single k-mer is performed by using an inverted PAC index. The inverted index is constructed for each partition. For each possible hash value, a bit vector is constructed that stores the leaf node, which has 1 at the position indicated by the hash value. The inverted index is constructed by evaluating each Bloom Filter successively. During the query of a k-mer, first, the k-mer is hashed. The hash value is searched in the inverted table. Then respective Bloom Filters are checked against the hash value to determine the presence of the k-mer. During the query of a sequence, first, the sequence is separated into super-k-mers. The k-mers are divided based on their partitions. Then the k-mers are hashed and searched in the Bloom Filters.

18.5 DNA assembly technique

De novo DNA assembly is a challenging problem, specifically in the case of large and complex genomes. DNA assembly reconstructs the chromosome sequence(s) from the billions of genomic reads. The processing of these huge volumes of reads requires a high-performance computing environment, specialized software, and high expertise [41]. These conditions are difficult to fulfill and lead to compromises in the DNA assembly techniques. The current state-of-the-art assemblers are not reconstructing the chromosome sequences fully. The reads are filtered to remove redundant reads to generate more contiguous sequences (contigs). In case linkage information is available, the contigs are ordered and oriented to obtain scaffolds. However, these scaffolds have some undetermined bases between

contigs. Overall, the whole DNA assembly process is time- and compute-intensive. This section discusses the role of the Bloom Filter in DNA assembly. This section also presents a review of the proposed DNA assembly techniques, and Table 18.6 presents the features and limitations of these techniques.

TABLE 18.6 Features and Limitations of Bloom Filter-based DNA Assembly Techniques.

Technique	Features	Limitations
LightAssembler [42]	• A lightweight DNA assembling algorithm • Pattern-blocked Bloom Filter reduces the cache misses • Maintains two Bloom Filters • First Bloom Filter stores the uniform sample of the sequenced *k*-mers • Second Bloom Filter stores the correct *k*-mers which are determined using a simple statistical test • Does not implement error correction technique instead the sequenced reads are scanned twice to determine the trusted nodes • Implements Minia's [33] traversal algorithm for graph traversal • Identifies trusted *k*-mer without using a counting module while reducing disk-space overhead • Multithreaded program is used to achieve parallelism	• Graph traversal latency is more • Minia requires disk space • Insufficient disk space results in false sequencing results • Minia algorithm performance reduces drastically when the whole *k*-mers are given as input
Kollector [43]	• An alignment-free targeted assembly pipeline • Alignment-free approach • Uses a BioBloom Tool in Bloom Filter, called Progressive Bloom Filter • User defines the length of *k*-mer overlapping • Reads thousands of transcript sequences concurrently • Implements ABySS [45] and GMAP [46] for scaffolding • Selects relatively divergent sequences due to PBF • PBF initially reads from conserved regions while later starts reading from more divergent regions	• Performance depends on the performance of ABySS and GMAP • Bloom Filter is sensitive to the arrangement of reads in the input file • Absence of assembly algorithm • Absence of scaffolds algorithm • Biased to short *k*-mers • Fails to reconstruct the long reads • Cannot identify the reads where exons are separated by long introns • Maintains many parameters to control the number of erroneous reads
ABySS 2.0 [45]	• Multi-stage de novo sequencer pipeline • New version of ABySS 1.0 • Bloom Filter reduces the memory requirements and removal of erroneous *k*-mers • Bloom Filter in unitig stage reduces the memory usage • Implements Minia [33] algorithm for graph traversal • A look-ahead mechanism during the graph traversal to eliminate short branches to reduce the number of false positives and sequencing errors • After completion of *k*-mer reading, except for the last chain Bloom Filter, all other Bloom Filters are discarded	• Erroneous sequences are ignored • Large-sized Bloom Filter used to reduce FPP • Implementation of cascading Bloom Filter requires large memory • Bloom Filter does not store edges • Inability of Bloom Filter to store edges leads to more query operations • Every graph traversal step performs four query operations for neighbor *k*-mers • Maintains many Bloom Filters during *k*-mer reading

LightAssembler [42] proposed a lightweight DNA assembling algorithm. LightAssembler implements Pattern-blocked Bloom Filter [6]. It maintains two Bloom Filters to reduce the cache misses. The first Bloom Filter stores a uniform sample of the sequenced *k*-mers. The second Bloom Filter stores the correct *k*-mers, which are determined using a simple statistical test. LightAssembler has two modules, graph construction and graph traversal. Furthermore, the graph construction module has two stages, uniform *k*-mers sampling and trust/untrust *k*-mer filtering. In the graph construction module, *k*-mers are inserted into Bloom Filter. LightAssembler does not implement an error correction technique, instead the sequenced reads are scanned twice to determine the trusted nodes. Bloom Filter stores the trusted nodes. Graph traversal module takes Bloom Filter and trusted *k*-mers as input and generates a set of sequenced contigs. The graph traversal module also has two steps, computing branching *k*-mers and simplifying the de Bruijn graph, and extending the branching *k*-mers. LightAssembler implements Minia's [33] traversal algorithm for graph traversal. The graph traversal latency is more because Minia requires disk space to handle the memory usage limitation. Another issue is that false sequences are generated in case of insufficient disk space. When the whole k-mers in the simulated datasets are given as input, the performance of the Minia algorithm reduces drastically.

Kollector [43] is an alignment-free targeted assembly pipeline. It uses BioBloom Tools (BBT) [44] in Bloom Filter, called Progressive Bloom Filter (PBF). The first stage is tagging, in which a set of genomic reads is scanned to select the read pairs having a certain length of k-mer overlap. The length of overlapping is defined by the user. The second stage is called a pipeline. PBF helps in the selection of a read having a k-mer overlap based on a fixed read length. PBF is biased because it is sensitive to the read arrangement in the input file. Kollector implements ABySS [45] and GMAP [46] for scaffolding. It maintains many parameters to control the number of erroneous reads. However, it is biased to short k-mers because it fails to reconstruct the long reads, i.e., approximately 20 kbp. The reason is that Kollector cannot identify the reads where exons are separated by long introns. As Kollector is a greedy algorithm, it selects reads of off-target regions. It selects relatively divergent sequences due to PBF, which initially reads from conserved regions while later starts reading from more divergent regions.

ABySS 2.0 [45] is a multi-stage de novo sequencer pipeline which is a new version of ABySS 1.0 [47]. Compared to ABySS 1.0, it implements Bloom Filter to reduce the memory requirements and the removal of erroneous k-mers. The Bloom Filter variant used is called cascading Bloom Filter [29]. It is used to represent the de Bruijn graph. It has three stages: uniting, contig, and scaffold. The k-mers in the unitig stage moves through two passes. The first pass retrieves the k-mers from the read to insert into the Bloom Filter. After completion of the k-mer reading, except for the last chain Bloom Filter, all other Bloom Filters are discarded. The last Bloom Filter contains the error-free k-mers. The second pass determines the error-free k-mers and uses them to construct the unitigs. ABySS 2.0 implements Minia [33] algorithm for graph traversal. ABySS 2.0 does not store edges in Bloom Filter, which leads to many query operations. Every graph traversal step performs four query operations for neighbor k-mers. The contig stage does not use Bloom Filter. In the scaffold stage, the pair reads are aligned to the contigs. Then they are orientated and joined to produce scaffolds. This stage also inserts characters at coverage gaps and resolves repeats. ABySS 2.0 uses a large-sized Bloom Filter to reduce FPP.

18.6 Other bioinformatics areas

Bloom Filter is not limited only to DNA sequencing. It helps in many other fields of bioinformatics. Some examples are genome annotation, mutation, pan-genomics. In this section, a brief introduction on the topic and the contribution of the Bloom Filter in increasing the technique efficiency and performance is explained.

18.6.1 Genome annotation

The DNA has a sequence of nucleotides which have some function and non-functional sequences having no evolutionary constraint. Genome annotation is the process of identifying the functional sequences of nucleotides in a given DNA fragment. The annotation process is performed after DNA Sequencing. Genome annotation gives location and features. The feature is defined as any region having a defined sequence or structure in the DNA fragment. Genome annotation gives information about different features such as the genomic position of intron–exon boundaries, gene names, protein products, regulatory sequences, and repeats. This information is stored along with DNA. It is stored in genomic databases, for example, Mouse Genome Informatics [48], GeneDB [49], Flybase [50]. It makes the DNA more informative and useful. In the annotation process [51], the first step is determining the genetic markers, for example, tRNA and rRNA. It uses different gene prediction or gene similarity checking algorithms, for example, Powermarker [52] and GenePRIMP [53]. After finding the features, biological information is attached to these locations. This information helps the scientists and researchers gain a proper understanding of the DNA.

ChopStitch [54] is a de novo annotation method. It finds reputed exons in the assembled DNA. It also constructs splice graphs using DNA Sequencer. It uses cascading Bloom Filter [29] to represent the k-mers. The Bloom Filter is constructed using the ntHash [55] algorithm, which hashes continuous k-mers in a sequence. ChopStitch uses the ntCard [56] algorithm to calculate the number of unique k-mers with user-specified FPP. ChopStitch ignores all single occurrences of k-mers to eliminate the k-mers having sequential errors. The elimination is done using a two-level cascading Bloom Filter. The first-level Bloom Filter is called the primary Bloom Filter (pBF). It stores the k-mers having one or more occurrences. The second-level Bloom Filter is called the secondary Bloom filter (sBF). Similarly, it stores the k-mers having more than one occurrence. During insertion of a new k-mer, it is first queried to the pBF. If the response is $True$, k-mer is inserted into sBF. Otherwise, k-mer is inserted into pBF. After the completion of the insertion step, pBF is discarded. To detect an exon–exon junction, sBF is checked. ChopStitch uses sBF as input to execute further steps of the method. ChopStitch considers only k-mers having cardinality more than one. It

believes those k-mers are not erroneous, which is not completely correct. During the determination of the exact exon boundaries, the performance of the ChopStitch decreases.

Schilken et al. [57] proposed a color-compression graph based on Bloom Filter. It is capable of handling pan-genome or metagenome data. The colored de Bruijn graph also has an associated annotation. The annotation is represented using a bit matrix. Each edge in the graph is associated with a subset of predefined annotation classes. The Bloom Filter is used to store the columns of the annotation matrix. Moreover, in the error correction step, the Bloom Filter is used to indicate the nodes whose neighboring edges have changed colors. The approach supports dynamic coloring. When more color labels are added to the colored de Bruijn graph, each color bit gets a new Bloom Filter. Similarly, deleting a color label means removing the corresponding Bloom Filter.

18.6.2 Pan-genomics

Pan-genome is defined as the set of genes of all strains of a species. It includes both core and accessory genomes. A core genome is defined as the genes present in all strains of a species. An accessory or variable genome is defined as the genes present in some specific strain. The pan-genome is classified into open or closed pan-genome. In an open pan-genome, the set of genes is open. It means genes get added to the set with an increase in its genes, for example, Escherichia coli. It lives in different environments, having mixed microbial communities. Such environments exchange genetic material in many ways. Hence, in the species, new genes get added. In a closed pan-genome, the set is limited. The pan-genome contains repetitive subsequences. It means hundreds and thousands of DNA sequences contain the same subsequence. Therefore, the construction of the pan-genome is complex.

Implementation of Bloom Filter is ideal when it is a case of repetitive data. A data structure called Bloom Filter Tree (BFT) [10] is proposed to represent the pan-genome data. It is used for indexing and compressing the pan-genome as a colored de Bruijn graph. It is reference- and alignment-free. Moreover, it is incremental. With the addition of a new DNA sequence, it does not require rebuilding the data structure. It is based on a burst tree that stores k-mers with a set of colors. It uses four Bloom Filters for quick recognition of the subsequence. Schilken et al. proposed a color-compressed graph for the representation of pan-genome data. To represent the pan-genome data, it constructs a colored de Bruijn graph using input sequences and their associated annotation. The annotation is a bit matrix. The column of the bit matrix is stored in different Bloom Filters. It is explained in more detail in Section 18.6.1.

18.6.3 Genetics of diseases

DNA sequencing helps in identifying the DNA sequence responsible for diseases such as cancer and Alzheimer's. It also helps in recognizing the mutation in the DNA sequence. Based on the genes, human diseases are classified into three categories [58], namely, monogenic, chromosomal, and multifactorial. Monogenic diseases are caused by a single gene alteration. Chromosomal diseases are caused by chromosome alteration. Multifactorial diseases are complex diseases caused by multiple gene variations. They may or may not be influenced by the environment. For example, cancer, birth defects, and diabetes. Scientists are more focused on finding a cure for such complex diseases. Moreover, it is not only the case in humans. Scientists also have focused on genetic diseases in plants. Some specific parasites cause diseases in specific plants. For example, Fusarium oxysporum fungus only attacks tomato plants. Genetics of diseases helps in making the plants' disease resistant.

Campbell et al. [59] used RNA sequencing (RNA-Seq) for comparison of gene expression of R. temporaria populations which suffered from ranaviral disease in the past with the population without the disease. Ranaviruses are viruses that cause disease in all classes of ectothermic vertebrates [60,61]. Bloom Filter is used for read filtering in RNA sequencing. A software package called BioBloom tool [44] is used to remove environmentally contaminated reads. BioBloom tool uses Bloom Filter for its memory and time efficiency advantage. The unique k-mers are extracted using ntCard program [56]. It helps to calculate the size of the Bloom Filter. The method inserts contaminated reads into the Bloom Filter. The new reads are first checked with Bloom Filter. If the read is present, it is discarded. Otherwise, the read is considered for further processing. Furthermore, Bloom Filter is constructed using genomic data of three frog species; namely, Xenopustropicalis [62], Nanoranaparkeri [63], and Lithobates (Rana) catesbeianus [64]. The contamination-free reads belonging to these species are considered for the experiment. All reads not matching Bloom Filter containing the contaminated reads and matching Bloom Filter containing genomic data of the three frog species are kept in their respective dataset. Otherwise, they are discarded from the dataset.

Sun and Medvedev [65] proposed VarGeno method for single nucleotide polymorphism (SNP) genotyping. Genotyping is the technology to determine genetic differences. SNP genotyping is the most widely used technique. It is used in human disease-related research [66]. VarGeno is built upon the LAVA tool [67]. It is improved by using

Bloom Filter and linear scanning. VarGeno constructs a Bloom Filter during index construction. VarGeno uses one hash function in Bloom Filter. Every k-mer is partitioned into r and $2k - r$ bits, where r is a parameter used by the VarGeno. The $2k - r$ bits of the reads are stored in Bloom Filter. During a query operation, first, the presence of the $2k - r$ bits of the read is checked. If found, then the upper neighbor k-mers are searched. Otherwise, searching for the upper neighbor, the k-mers are abandoned.

18.7 Discussion

Bloom Filter is a simple data structure. It is not a stand-alone technique. However, a proper implementation with any technique makes it a very adaptive data structure. DNA sequencing or assembly is a time- and compute-intensive process. It consists of many lengthy stages, which are individually time- and compute-intensive processes. Currently, Bloom Filter is explored for bioinformatics. The low time and space complexity of Bloom Filter make it ideal for the processing of genomic data, which consists of many repetitive data. In some cases, this repetition leads to error because the high similarity between them makes it difficult to identify the difference. However, Bloom Filter is capable of identifying unique DNA sequences. Furthermore, Bloom Filter is implemented in every stage to improve the performance of each stage.

Table 18.7 compares the Bloom Filter-based techniques of bioinformatics which are reviewed in the chapter. From the table, it is observed that the majority of techniques used the standard Bloom Filter which has many issues. Besides, these techniques are not taking any extra steps to reduce the FPP caused by the standard Bloom Filter.

TABLE 18.7 Comparison of Bloom Filter-based Bioinformatics Techniques: Variant, Bloom Filter variant; Reduce FP, technique used to reduce the number of false positives; Purpose, purpose of the technique.

Technique	Variant	Reduce FP	Purpose
Melsted and Pritchard [17], 2011	Standard	No	k-mer counting
Turtle [7], 2014	Blocked Bloom Filter and Counting Bloom Filter	No	k-mer counting
BARCODE [28], 2014	Cascading	Stores the reads in another Bloom Filter	Read compression
Lighter [9], 2014	Pattern-blocked	No	Error correction
Salikhov et al. [29], 2014	Standard	No	de Bruijn graph construction
LEON [26], 2015	Standard	No	Read compression
FADE [8], 2015	Counting	No	Error correction
LightAssembler [42], 2016	Pattern-blocked	No	DNA assembly
Bloom Filter Tree [10], 2016	Standard	No	Colored de Bruijn graph
Squeakr [20], 2017	Counting Quotient	No	k-mer counting
Mcvicar et al. [15], 2017	Standard	False positive responses are stored in a hash table	k-mer counting
DARRC [30], 2017	Standard	No	Read compression
deBGR [35], 2017	Counting quotient	Another CQF	de Bruijn graph construction
Faucet [36], 2017	Standard	No	de Bruijn graph construction
Kollector [43], 2017	Progressive	No	DNA assembly
ABySS 2.0 [45], 2017	Cascading	No	DNA assembly
SSBT [38], 2017	Standard	No	Searching
ChopStitch [54], 2017	Cascading	No	de novo annotation
Schilken et al. [57], 2017	Standard	No	Color-compression graph
Campbell et al. [59], 2018	Standard	No	Comparison of gene expression
Sun and Medvedev [65], 2018	Standard	No	Genotyping
kmcEx [21], 2019	Two standard arrays	Checks neighboring k-mers	k-mer counting
CQF-deNoise [22], 2020	Counting Quotient	No	k-mer counting
SWAPCounter [23], 2020	Standard	No	k-mer counting
kmtricks [24], 2021	Standard	No	k-mer counting
KCOSS [25], 2021	Standard	No	k-mer counting
MetaProFi [39], 2021	Standard	No	Indexing
PAC [40], 2022	Aggregated	No	Searching

The *k*-mer counting is the most important step of DNA assembly. However, it is very compute-intensive and takes the majority of the processing time of DNA assembly. The *k*-mer counting stage reads the genomic reads from the file and counts the *k*-mer. Essentially, this is a filtering stage; hence, Bloom Filter is the ideal data structure for filtering. Many techniques implement CBF; however, the issue with the CBF is decreasing the advantage gained due to the use of Bloom Filter. Instead, CBF variants should be explored. Another way of not using CBF is using two Bloom Filters. The first Bloom Filter saves all unique *k*-mers. The second Bloom Filter saves the *k*-mers present in the first Bloom Filter. However, considering the double or more time for processing *k*-mers is not efficient. Instead, a CBF variant can help in considering the *k*-mers having frequency more than a fixed value. Another issue is the unique *k*-mers which are still many. The CBF variant needs to be small in size to accommodate completely within the RAM while having low FPP. In this stage, Bloom Filter also helps in removing many errors. An error occurring due to a change of a base with an unknown character will make it a unique *k*-mer which will be discarded later due to low frequency. Usually, for the storage of genomic data, first, a graph is constructed to reduce the memory requirements. After the graph construction, the graph is compressed using any good compression method. Then the graph is stored in the disk for long-term storage. Graph construction is very time consuming because every read needs to be processed and inserted in an appropriate node in the graph. Many reads are repetitive; hence, the same reads are repetitively processed before ignoring. In such a scenario, Bloom Filter is the required data structure. All the operations of Bloom Filter have constant time complexity. So, verifying the uniqueness of reads before graph processing drastically reduces the graph construction time. Furthermore, Bloom Filter is itself a bit mapped vector which provides data security because it does not store the genomic read. Also, many genomic data can be insured in a small Bloom Filter. Hence, it drastically reduces the memory requirements. Thus, Bloom Filter helps in reducing the data compression processing time.

18.8 Conclusion

Bioinformatics tries to take a helping hand from technology to unlock the mystery of nature, i.e., biology. Becoming aware of DNA, it opened the door to know more about humans and other organisms. The study has helped solve many problems, such as finding a cure to many diseases. However, the huge size of genomic data and the huge volume of information the DNA produces makes it impossible to handle without the help of sophisticated technology. Bloom Filter is a simple data structure which is ideal for highly repetitive data such as genomic data. Bloom Filter is also implemented in every stage of the DNA sequencing process. This chapter explores the role of Bloom Filter in bioinformatics, more specifically in DNA assembly. We have also discussed the role of Bloom Filter in preprocessing filtering, i.e., *k*-mer counting, read compression, and error correction. The discussion also included the role of Bloom Filter in enhancing the performance of the construction of de Bruijn graphs and searching for genomic data in databases. Bloom Filter also found its place in other bioinformatics fields such as genome annotation, genetics of diseases, and so on. However, there is still a requirement for more efficient variants of Bloom Filter capable of scaling to accommodate all genomic data. Those variants should be able to go further than just membership checking and counting.

References

[1] N.M. Luscombe, D. Greenbaum, M. Gerstein, What is bioinformatics? A proposed definition and overview of the field, Methods Inf. Med. 40 (04) (2001) 346–358.
[2] Z.D. Stephens, S.Y. Lee, et al., Big data: astronomical or genomical?, PLoS Biol. 13 (7) (2015), https://doi.org/10.1371/journal.pbio.1002195.
[3] J.R. Miller, S. Koren, G. Sutton, Assembly algorithms for next-generation sequencing data, Genomics 95 (6) (2010) 315–327.
[4] D. Zerbino, E. Birney, Velvet: algorithms for de novo short read assembly using de Bruijn graphs, Genome Res. (2008) gr-074492.
[5] P. Pandey, M.A. Bender, R. Johnson, R. Patro, A general-purpose counting filter: making every bit count, in: Proceedings of the 2017 ACM International Conference on Management of Data, ACM, 2017, pp. 775–787.
[6] F. Putze, P. Sanders, J. Singler, Cache-, hash- and space-efficient Bloom filters, in: International Workshop on Experimental and Efficient Algorithms, Springer, 2007, pp. 108–121.
[7] R.S. Roy, D. Bhattacharya, A. Schliep, Turtle: identifying frequent *k*-mers with cache-efficient algorithms, Bioinformatics 30 (14) (2014) 1950–1957.
[8] A. Ramachandran, Y. Heo, W.-m. Hwu, J. Ma, D. Chen, FPGA accelerated DNA error correction, in: Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, EDA Consortium, 2015, pp. 1371–1376.
[9] L. Song, L. Florea, B. Langmead, Lighter: fast and memory-efficient sequencing error correction without counting, Genome Biol. 15 (11) (2014) 509.

[10] G. Holley, R. Wittler, J. Stoye, Bloom filter tree: an alignment-free and reference-free data structure for pan-genome storage, Algorithms Mol. Biol. 11 (1) (2016) 3.
[11] G. Holley, R. Wittler, J. Stoye, Bloom filter tree – a data structure for pan-genome storage, in: International Workshop on Algorithms in Bioinformatics, Springer, 2015, pp. 217–230.
[12] S. Meader, L.W. Hillier, D. Locke, C.P. Ponting, G. Lunter, Genome assembly quality: assessment and improvement using the neutral indel model, Genome Res. 20 (5) (2010) 675–684.
[13] J.-H. Choi, S. Kim, H. Tang, J. Andrews, D.G. Gilbert, J.K. Colbourne, A machine-learning approach to combined evidence validation of genome assemblies, Bioinformatics 24 (6) (2008) 744–750.
[14] G. Lunter, C.P. Ponting, J. Hein, Genome-wide identification of human functional DNA using a neutral indel model, PLoS Comput. Biol. 2 (1) (2006) e5.
[15] N. Mcvicar, C.-C. Lin, S. Hauck, *k*-mer counting using Bloom filters with an FPGA-attached HMC, in: Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on, IEEE, 2017, pp. 203–210.
[16] N.S. Movahedi, E. Forouzmand, H. Chitsaz, De novo co-assembly of bacterial genomes from multiple single cells, in: Bioinformatics and Biomedicine (BIBM), 2012 IEEE International Conference on, IEEE, 2012, pp. 1–5.
[17] P. Melsted, J.K. Pritchard, Efficient counting of *k*-mers in DNA sequences using a Bloom filter, BMC Bioinform. 12 (1) (2011) 333.
[18] J.T. Pawlowski, Hybrid memory cube (HMC), in: 2011 IEEE Hot Chips 23 Symposium (HCS), IEEE, 2011, pp. 1–24.
[19] L. Fan, P. Cao, J. Almeida, A.Z. Broder, Summary cache: a scalable wide-area web cache sharing protocol, IEEE/ACM Trans. Netw. 8 (3) (2000) 281–293, https://doi.org/10.1109/90.851975.
[20] P. Pandey, M.A. Bender, R. Johnson, R. Patro, Squeakr: an exact and approximate *k*-mer counting system, Bioinformatics 34 (4) (2017) 568–575.
[21] P. Jiang, J. Luo, Y. Wang, P. Deng, B. Schmidt, X. Tang, N. Chen, L. Wong, L. Zhao, kmcEx: memory-frugal and retrieval-efficient encoding of counted *k*-mers, Bioinformatics 35 (23) (2019) 4871–4878, https://doi.org/10.1093/bioinformatics/btz299.
[22] C.H. Shi, K.Y. Yip, A general near-exact *k*-mer counting method with low memory consumption enables de novo assembly of 106× human sequence data in 2.7 hours, Bioinformatics 36 (Supplement_2) (2020) i625–i633, https://doi.org/10.1093/bioinformatics/btaa890.
[23] J. Ge, J. Meng, N. Guo, Y. Wei, P. Balaji, S. Feng, Counting *k*-mers for biological sequences at large scale, Interdiscip. Sci. 12 (1) (2020) 99–108.
[24] T. Lemane, P. Medvedev, R. Chikhi, P. Peterlongo, kmtricks: efficient and flexible construction of Bloom filters for large sequencing data collections, Bioinform. Adv. 2 (1) (2022) 1–8, https://doi.org/10.1093/bioadv/vbac029.
[25] D. Tang, Y. Li, D. Tan, J. Fu, Y. Tang, J. Lin, R. Zhao, H. Du, Z. Zhao, KCOSS: an ultra-fast *k*-mer counter for assembled genome analysis, Bioinformatics 38 (4) (2021) 933–940, https://doi.org/10.1093/bioinformatics/btab797.
[26] G. Benoit, C. Lemaitre, D. Lavenier, E. Drezen, T. Dayris, R. Uricaru, G. Rizk, Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph, BMC Bioinform. 16 (1) (2015) 288.
[27] GB Editorial Team, Closure of the NCBI SRA and implications for the long-term future of genomics data storage, Genome Biol. 12 (3) (2011) 402, https://doi.org/10.1186/gb-2011-12-3-402.
[28] R. Rozov, R. Shamir, E. Halperin, Fast lossless compression via cascading Bloom filters, BMC Bioinform. 15 (9) (2014) S7.
[29] K. Salikhov, G. Sacomoto, G. Kucherov, Using cascading Bloom filters to improve the memory usage for de Brujin graphs, Algorithms Mol. Biol. 9 (1) (2014) 2.
[30] G. Holley, R. Wittler, J. Stoye, F. Hach, Dynamic alignment-free and reference-free read compression, in: International Conference on Research in Computational Molecular Biology, Springer, 2017, pp. 50–65.
[31] S. Deorowicz, S. Grabowski, Data compression for sequencing data, Algorithms Mol. Biol. 8 (1) (2013) 25.
[32] Y. Heo, X.-L. Wu, D. Chen, J. Ma, W.-M. Hwu, Bless: Bloom filter-based error correction solution for high-throughput sequencing reads, Bioinformatics 30 (10) (2014) 1354–1362.
[33] R. Chikhi, A. Limasset, S. Jackman, J.T. Simpson, P. Medvedev, On the representation of de Bruijn graphs, in: International Conference on Research in Computational Molecular Biology, Springer, 2014, pp. 35–55.
[34] B.J. Haas, A. Papanicolaou, M. Yassour, M. Grabherr, P.D. Blood, J. Bowden, M.B. Couger, D. Eccles, B. Li, M. Lieber, et al., De novo transcript sequence reconstruction from rna-seq using the trinity platform for reference generation and analysis, Nat. Protoc. 8 (8) (2013) 1494.
[35] P. Pandey, M.A. Bender, R. Johnson, R. Patro, deBGR: an efficient and near-exact representation of the weighted de Bruijn graph, Bioinformatics 33 (14) (2017) i133–i141.
[36] R. Rozov, G. Goldshlager, E. Halperin, R. Shamir, Faucet: streaming de novo assembly graph construction, Bioinformatics 34 (1) (2017) 147–154.
[37] ENA browser, https://www.ebi.ac.uk/ena/browser/about/statistics. (Accessed 20 February 2022).
[38] B. Solomon, C. Kingsford, Improved search of large transcriptomic sequencing databases using split sequence Bloom trees, in: International Conference on Research in Computational Molecular Biology, Springer, 2017, pp. 257–271.
[39] S.K. Srikakulam, S. Keller, F. Dabbaghie, R. Bals, O.V. Kalinina, Metaprofi: a protein-based Bloom filter for storing and querying sequence data for accurate identification of functionally relevant genetic variants, bioRxiv, https://dx.doi.org/10.1101/2021.08.12.456081.
[40] C. Marchet, A. Limasset, Scalable sequence database search using partitioned aggregated Bloom comb-trees, bioRxiv, https://dx.doi.org/10.1101/2022.02.11.480089.
[41] N. Nagarajan, M. Pop, Sequence assembly demystified, Nat. Rev. Genet. 14 (3) (2013) 157–167.
[42] S. El-Metwally, M. Zakaria, T. Hamza, LightAssembler: fast and memory-efficient assembly algorithm for high-throughput sequencing reads, Bioinformatics 32 (21) (2016) 3215–3223.
[43] E. Kucuk, J. Chu, B.P. Vandervalk, S.A. Hammond, R.L. Warren, I. Birol, Kollector: transcript-informed, targeted de novo assembly of gene loci, Bioinformatics 33 (12) (2017) 1782–1788.
[44] J. Chu, S. Sadeghi, A. Raymond, S.D. Jackman, K.M. Nip, R. Mar, H. Mohamadi, Y.S. Butterfield, A.G. Robertson, I. Birol, BioBloom tools: fast, accurate and memory-efficient host species sequence screening using Bloom filters, Bioinformatics 30 (23) (2014) 3402–3404.
[45] S.D. Jackman, B.P. Vandervalk, H. Mohamadi, J. Chu, S. Yeo, S.A. Hammond, G. Jahesh, H. Khan, L. Coombe, R.L. Warren, et al., Abyss 2.0: resource-efficient assembly of large genomes using a Bloom filter, Genome Res. (2017) gr-214346.

[46] T.D. Wu, C.K. Watanabe, GMAP: a genomic mapping and alignment program for mRNA and EST sequences, Bioinformatics 21 (9) (2005) 1859–1875, https://doi.org/10.1093/bioinformatics/bti310.
[47] J.T. Simpson, K. Wong, S.D. Jackman, J.E. Schein, S.J. Jones, I. Birol, Abyss: a parallel assembler for short read sequence data, Genome Res. (2009) gr-089532.
[48] D. Shaw, Searching the mouse genome informatics (MGI) resources for information on mouse biology from genotype to phenotype, Curr. Protoc. Bioinform. 5 (1) (2004) 1–7.
[49] F.J. Logan-Klumpler, N. De Silva, U.e.a. Boehme, GeneDB – an annotation database for pathogens, Nucleic Acids Res. 40 (D1) (2011) D98–D108.
[50] S. Tweedie, M. Ashburner, K.e.a. Falls, Flybase: enhancing drosophila gene ontology annotations, Nucleic Acids Res. 37 (suppl_1) (2008) D555–D559.
[51] L. Stein, Genome annotation: from sequence to biology, Nat. Rev. Genet. 2 (7) (2001) 493.
[52] K. Liu, S.V. Muse, Powermarker: an integrated analysis environment for genetic marker analysis, Bioinformatics 21 (9) (2005) 2128–2129.
[53] A. Pati, N.N. Ivanova, N.e.a. Mikhailova, GenePRIMP: a gene prediction improvement pipeline for prokaryotic genomes, Nat. Methods 7 (6) (2010) 455.
[54] H. Khan, H. Mohamadi, B.P. Vandervalk, R.L. Warren, J. Chu, I. Birol, ChopStitch: exon annotation and splice graph construction using transcriptome assembly and whole genome sequencing data, Bioinformatics 34 (10) (2017) 1697–1704.
[55] H. Mohamadi, J. Chu, B.P. Vandervalk, I. Birol, ntHash: recursive nucleotide hashing, Bioinformatics 32 (22) (2016) 3492–3494.
[56] H. Mohamadi, H. Khan, I. Birol, ntCard: a streaming algorithm for cardinality estimation in genomics data, Bioinformatics 33 (9) (2017) 1324–1330.
[57] I. Schilken, H. Mustafa, G. Rätsch, C. Eickhoff, A. Kahles, Efficient graph-color compression with neighborhood-informed Bloom filters, bioRxiv (2017) 239806.
[58] A. Vieira, Genes and disease, https://www.nature.com/scitable/topic/genes-and-disease-17. (Accessed 6 January 2019).
[59] L.J. Campbell, S.A. Hammond, S.J. Price, M.D. Sharma, T.W. Garner, I. Birol, C.C. Helbing, L. Wilfert, A.G. Griffiths, A novel approach to wildlife transcriptomics provides evidence of disease-mediated differential expression and changes to the microbiome of amphibian populations, Mol. Ecol. 27 (6) (2018) 1413–1427.
[60] R.E. Marschang, Viruses infecting reptiles, Viruses 3 (11) (2011) 2087–2126.
[61] R. Whittington, J. Becker, M. Dennis, Iridovirus infections in finfish – critical review with emphasis on ranaviruses, J. Fish Dis. 33 (2) (2010) 95–122.
[62] U. Hellsten, R.M. Harland, M.J. Gilchrist, D. Hendrix, J. Jurka, V. Kapitonov, I. Ovcharenko, N.H. Putnam, S. Shu, L. Taher, et al., The genome of the western clawed frog xenopus tropicalis, Science 328 (5978) (2010) 633–636.
[63] Y.-B. Sun, Z.-J. Xiong, X.-Y. Xiang, S.-P. Liu, W.-W. Zhou, X.-L. Tu, L. Zhong, L. Wang, D.-D. Wu, B.-L. Zhang, et al., Whole-genome sequence of the Tibetan frog Nanorana parkeri and the comparative evolution of tetrapod genomes, Proc. Natl. Acad. Sci. 112 (11) (2015) E1257–E1262.
[64] S.A. Hammond, R.L. Warren, B.P. Vandervalk, E. Kucuk, H. Khan, E.A. Gibb, P. Pandoh, H. Kirk, Y. Zhao, M. Jones, et al., The North American bullfrog draft genome provides insight into hormonal regulation of long noncoding RNA, Nat. Commun. 8 (1) (2017) 1433.
[65] C. Sun, P. Medvedev, Toward fast and accurate SNP genotyping from whole genome sequencing data for bedside diagnostics, Bioinformatics 35 (3) (2018) 415–420.
[66] J.N. Hirschhorn, M.J. Daly, Genome-wide association studies for common diseases and complex traits, Nat. Rev. Genet. 6 (2) (2005) 95.
[67] A. Shajii, D. Yorukoglu, Y. William Yu, B. Berger, Fast genotyping of known SNPs through approximate *k*-mer matching, Bioinformatics 32 (17) (2016) i538–i544.

Index

A
Activation period, 134
Actual Chain Offset (ACO), 62
Adaptive Bloom Filter (ABF), 58
Advanced Encryption Standard (AES), 176
Aggregated Interest Message (AIM), 105
All Name Prefix Matching (ANPM), 107
Application layer, 117
Application Specific Integrated Circuit (ASIC), 123
Arbitration Center (AC), 167
Architecture, 14, 65, 119
  Bloom Filter, 7, 8
  hierarchical, 79, 86
  network, 4, 89, 117
  SDN, 117, 129
Attack
  deletion, 151
  dictionary, 151
  DNS amplification, 146
  flooding, 145
    interest, 147
    link, 147
  NTP, 146
  pollution, 150
  query-only, 151
  vulnerability, 145
Attribute Bloom Filter (ABF), 154
Autoscaling Bloom Filter, 61

B
Backup Bloom Filter (BBF), 53
Backyard Cuckoo hashing, 60
Balanced counting Bloom Filters, 71
Behavioral biometrics, 187
Big Data, 161, 162, 165, 170
Big Data 2.0, 52
Binary search tree, 187, 192, 193
Binary tree, 81, 180, 187, 189
  Bloom Filter, 180, 189, 192
BioBloom tool, 210
Bioinformatics, 197, 209, 212
Biometric
  features, 187
  indexing, 187, 188
  template, 188
Biometrics, 5, 187
  behavioral, 187
  cancelable, 5, 187, 190, 192, 193
  chemical, 187
Bit array, 10
Bits
  fingerprint, 73
  metadata, 73
Blacklist, 146, 148, 152
Block Random Access Memory (BRAM), 166
Block-based Bloom Filter, 15
Blocked Bloom Filter, 58, 197
Bloofi, 60, 83, 165
  encrypted, 165
Bloom Filter, 3, 7, 8, 37, 38, 51, 52, 55, 103
  adaptive, 58
  architecture, 7, 8
  attribute, 154
  autoscaling, 61
  backup, 53
  balanced counting, 71
  binary tree, 180, 189, 192
  block-based, 15
  blocked, 58, 197
  cache-based, 16
    counting, 72
  cascading, 202, 209
  chained, 16
  compressed counting, 72
  conventional, 3, 7, 8, 14, 15, 18, 23, 24, 39, 47, 57, 77, 81, 83, 154, 157
  counting, 14, 23, 57, 65, 90
  Cuckoo, 62
  d-left counting, 70
  difference, 61
  double layer counting, 72
  dual counting, 74, 147
  dynamic, 15, 58
  encrypted, 169, 176, 179
  extensible, 95, 97
  false negative, 11
  false positive, 11
  fingerprint counting, 73
  first counting, 68
  flash/SSD-based, 16
  flat, 15
  floating counter, 71
  forgetful, 167
  garbled, 154, 180
  HDD-based, 17
  hierarchical, 13, 16, 59, 79, 80, 82, 83, 85, 86, 138, 189
  hierarchical counting, 80, 82
  Huffman counting, 81
  in-packet, 136
  index-split, 59
  initial, 54
  interest, 105
  invertible, 183
  L-counting, 69
  learned, 53, 124
    sandwiched, 54
  limitations, 52, 101
  linked counting, 69
  multi-granularities counting, 80
  multidimensional, 16, 62, 63, 165
  multilayer compressed counting, 81
  multilevel counting, 82
  multiple, 123
  multiple-partitioned counting, 82, 85
  non-counting, 14
  One Memory Access, 59, 62
  operations, 8
  primary, 209
  probabilistic, 92
  progressive, 209
  queue, 123
  RAM-based, 16
  response, 11, 57, 138, 151, 153
  retouched, 58
  reversible multilayer hashed counting, 82
  scalable, 58
  secondary, 209
  segmented, 201
  settings, 47
  space-code, 69
  spectral, 57, 81
  split, 206
  stable, 58
  standard, 23, 27, 57
  static, 15
  taxonomy, 13
  temporal counting, 72
  ternary, 61
  time-out, 93
  true negative, 11
  true positive, 11
  variable-length counting, 70
  weighted, 58
  wrap-around counting, 73, 146, 148
Bloom Filter array (BFA), 58
Bloom Filter Queue (BFQ), 92
Bloom Filter Tree (BFT), 210
Bloom Filter-Aided haSh Table (BFAST), 93
Bloom Filter-based Request Node Collaboration (BRCC), 107
Bloom Filter-based Unknown Tag Identification (BUTI), 140
Bloom Filters and Least Recent Used (BF-LRU), 90
Bloom table, 176
Bloom-Filter-aided Redundancy Elimination (BFRE), 134
Bloomier Filter, 58
Blooming Tree, 81
Bucket
  hash, 93, 94
  index, 58
  locations, 71
  time-out value, 93
Buffer counter, 70

C
Cache Information Base (CIB), 105

Cache privacy, 111
Cache privacy protection mechanism (CPPM-DAM), 112
Cache-based Bloom Filter, 16
Cache-based counting Bloom Filter, 72
Cached counting Bloom Filter (CCBF), 80
Caching age (CA), 107
Caching Index (CI), 107
Caching Information Table (CIT), 107
Caching Remain Age (CRA), 107
Cancelable biometrics, 5, 187, 190, 192, 193
Cascading Bloom Filter, 202, 209
Cell counter, 82
Chained Bloom Filter, 16
Chemical biometrics, 187
Cloud computing, 5, 175, 176, 179, 181
Cloud data storage management, 181
Cloud Service Provider (CSP), 167
Co-operative Virtual Network Embedding (COVE), 94
Completely distributed controllers, 119
Compressed Bitmap Index and traffic Downsampling (CBID), 152
Compressed counting Bloom Filter, 72
Content Addressable Memory (CAM), 123
Content Advertisement Interest (CAI), 103
Content Advertisement Request (CAR), 105
Content concentration, 107
Content privacy, 111
Content store, 101, 105, 106
Content-Centric communication, 101
Control layer, 117, 118
Controller-Based Caching and Forwarding Scheme (CCFS), 105
Conventional Bloom Filter, 3, 7, 8, 14, 15, 18, 23, 24, 39, 47, 57, 77, 81, 83, 154, 157
Convergent All-Or-Nothing Transform (CAONT), 182
Counter overflow, 76
Counting Bloom Filter (CBF), 14, 23, 57, 65, 90
Counting quotient filter (CQF), 73, 199, 205
Counting Quotient Filter deNoise (CQF-deNoise), 201
Cuckoo Bloom Filter, 62
Cuckoo Filter, 13, 23, 31

D
d-left counting Bloom Filter, 70
Data
  deduplication, 162, 183
  duplicate, 77, 162, 170, 176
  filtering, 161, 170
  flooding attack, 148
  indexing, 176
  layer, 118, 119
  packet, 4, 57, 102, 148
  privacy, 154, 176, 182
  searching, 176
  traffic, 52, 56
  user, 153, 175, 176, 179–181
Database, 165
Dataset, 47
DDoS, 145, 146
  attacks, 146
  traffic, 145
de Bruijn graph, 202, 204
Deduplication, 65, 77, 79, 107, 165, 170, 175
  data, 162, 183
  process, 13, 57
Deep Learning, 55, 56
Deletable Bloom filter (DlBF), 59
Delete operation, 8, 12, 23, 57, 63, 65, 67, 70, 77, 82, 83, 110, 124, 140
Deleted item, 12
Deletion, 9, 59, 65, 67, 83
  attack, 151
  operation, 9, 10, 14, 16, 19, 52, 57, 59–62, 65, 67, 73, 83, 98, 113, 151, 156, 188
Detached Counting Bloom array (DCBA), 71
Dictionary attacks, 151
Difference Bloom Filter (DBF), 61
Disjoint Set (DS), 27, 29, 39
Diversity, 190
Divided word line (DWL), 70
DLCBF, 58, 72, 73, 77
DNA, 197, 209
  assembly technique, 207
DNS amplification attack, 146
Domain Name System (DNS), 127, 146
DoS, 148, 149
Double Layer Counting Bloom Filter, 72
Downsampling threshold, 152
Dual Counting Bloom Filter (DCBF), 74, 147
Duplicate data, 77, 162, 170, 176
Dynamic Bloom Filter (DBF), 15, 58

E
East/WestBound, 118
Encoded packet, 134
Encrypted
  Bloofi, 165
  Bloom Filter, 169, 176, 179
  data items, 154
Error correction, 203
European Nucleotide Archive (ENA), 206
Evaluation, 155
Exact Name Matching (ENM), 105
Exclusion set, 125
Extensible Bloom Filter (XBF), 95, 97

F
Fake Pseudonym List (FPL), 138, 151
False negative, 12, 13, 24, 156
False positive, 12, 24, 67, 77, 83, 157
  analysis, 17, 45, 46
  probability, 3, 18, 37, 38, 40, 41, 45
  rate, 58
Fast Two-Dimensional Filter Pending Interest Table, 108
Fastest filters, 31, 33
FastHash, 38
  function, 48
Feedback Interest Message (FIM), 105
Fetching the Nearest Replica (FNR), 106
Field Programmable Gate Array (FPGA), 96, 166
Filter
  Cuckoo, 13, 23, 31
  membership, 7, 15, 23, 33, 34
  memory, 59
  remainder, 206
  SCBF, 69
  similarity, 206
  SRAM, 61
Filtering, 65, 77, 79, 171
  data, 161, 170
  malicious URL, 55
  membership, 7, 16
  preprocessing, 198
Fingerprint Counting Bloom Filter (FP-CBF), 73
First compressed form, 182
First counting Bloom Filter, 68
First Hop Hit Ratio (FHHR), 107
Flash Assisted Segmented Bloom Filter for Deduplication (FASBF), 164
Flash/SSD-based Bloom Filter, 16
Flat Bloom Filter, 15
FlexSketchMon, 90
Floating counter Bloom Filter, 71
Flooding attack, 145
ForCES, 120
Forgetful Bloom Filter (FBF), 167
Forwarding Information Base (FIB), 101, 109
Fuzzy folded Bloom Filter (FFBF), 182

G
Garbled Bloom Filter (GBF), 154, 180
Genetics of diseases, 210
Genome annotation, 209
Graphical user interface (GUI), 119

H
Hash bucket, 93, 94
Hash Bucket Matrix (HBM), 164
Hash table (HT), 108, 110
Hashing, 83
  Backyard Cuckoo, 60
  operation, 58
HDD-based Bloom Filter, 17
Hierarchical
  architecture, 79, 86
  Bloom Filter, 13, 16, 59, 79, 80, 82, 83, 85, 86
Hierarchical Bloom Filter (HBF), 138, 189
Hierarchical Bloom Filter-based Indexing (HBFI), 138
Hierarchical Counting Bloom Filters (HCBF), 80, 82
Hot data index, 164
Huffman counting Bloom Filter (HCBF), 81
Hybrid Memory Cube (HMC), 198
HyperSight, 92

I
Importance decay, 123
Importance value, 123
In-packet Bloom Filter (iBF), 136
Inclusion set, 125
Index vector, 180
Index-split Bloom Filter (ISBF), 59
Information proprietor (IP), 167
Infrastructure layer, 118
Initial Bloom Filter (IBF), 54

Initial counter value (ICV), 72
Initial Source Node (ISN), 133
Insert operation, 70, 71, 73, 80–82, 181
Insertion, 9, 66
function, 61
operation, 9, 42, 58, 62, 66, 67, 69, 71, 73, 80, 90, 96, 124, 127, 140, 150, 154
performance, 28, 40
Interest Bloom Filter (IBF), 105
Interest flooding attack, 147
Interest packet, 4, 101–103, 147
Internet Group Management Protocol (IGMP), 125
Internet Protocol (IP), 89
Internet Service Providers (ISP), 127
Internet-of-Things (IoT), 138
Interoperability, 126
Intrusion detection system (IDS), 151
Invertible Bloom Filter (IBF), 183
Invertible Bloom Lookup Table (IBLT), 165
Is-zero, 69
ItemID, 162

J
Join operation, 85, 166, 170, 171
Junction node, 205

K
k-mer counting, 198, 201, 212
Key objectives, 12

L
L-counting Bloom Filter, 69
Leaf node, 162
Learned Bloom Filter, 53–55, 124
sandwiched, 54
Least Recently Used (LRU), 107
Level Counter (LC), 80
Libbloom, 41
Limitations of Bloom Filter, 52, 101
Linear feedback shift registers (LFSR), 69
Link flooding attack, 147
Linked counting Bloom Filter, 69
Locality sensitive hashing (LSH), 176
Logging, 148
Logical block address (LBA), 164
Logical Chain Offset (LCO), 62
Logical functional block (LFB), 120
Logically centralized controllers, 119
Longest Name Prefix Matching (LNPM), 109
Longest prefix matching, 96
Lookup, 9, 67, 110
operation, 9–12, 23, 28, 61, 63, 67, 90, 96
performance, 28
Lookup Interest (LI), 108
LRU BF, 79

M
Machine learning (ML), 51, 147
Malicious packets, 92, 147
Malicious URL detection, 51
Malicious URL filtering, 55
Mapping Array (MA), 107
Mapping Bloom Filter (MBF), 107
MapReduce, 5, 85, 161, 165, 169, 171
framework, 169, 171
Masked Parallel Bloom Filter (MPBF), 121
MCBF, 150
Membership
filter, 7, 15, 23, 33, 34
filtering, 7, 16
query, 80
response, 86
Memory, 47
allocation, 15
banks, 72
consumption, 10, 23, 31, 45, 49, 62, 65, 77–79, 83, 148–150
filter, 59
MetaProFi, 207
PIT, 148
size, 3, 15, 18, 23, 41, 47, 207
Metadata, 120, 134, 164, 167
bits, 73
server, 79, 80, 85, 86, 164
workload, 86
Mixed Set (MS), 27, 29, 39
Mobile ad-hoc network (MANET), 131, 136
Modulus operation, 37
Morton Filter (MF), 23, 32–34
Multi-core controllers, 119
Multi-granularities counting Bloom Filter, 80
Multicasting data packets, 107
Multidimensional Bloom Filter (MBF), 16, 62, 63, 165
Multilayer compressed counting Bloom Filter (ML-CCBF), 81
Multilevel counting Bloom Filter (MLCBF), 82
Multipath routing, 107
Multiple Bloom Filters (MBF), 123
Multiple-Partitioned Counting Bloom Filter (MPCBF), 82, 85
Multiresolution SCBF, 69
Multiset query, 80
Murmur, 38
hash function, 7, 9, 18, 20, 23, 25, 28, 38, 48, 62

N
Name Prefix Tree (NPT), 110
Name privacy, 111
Named Data Networking (NDN), 101
packet, 102
router, 105, 112, 113
Named-data network, 4
Neighbor Filter (NF), 133
Network
administrator, 151, 152
architecture, 4, 89, 117
bottleneck, 117
failure, 157
flow, 97, 98, 125
ID, 96
MANET, 131, 136
monitoring, 92, 119
named-data, 4
nodes, 94, 117, 129, 156
SDN, 118, 119, 125, 126, 129
security, 4, 51, 145, 156, 157
topology, 89, 118, 119, 123–125, 141
virtual, 94
private, 94
wireless, 4
sensor, 131, 136
WSN, 135
Network Coding (NC), 103
Network Operating System (NOS), 118
Network Time Protocol (NTP), 146
Non-counting Bloom Filter, 14
Northbound, 118
Notification packet, 124
NTP attack, 146

O
One Memory Access Bloom Filter (OMABF), 59, 62
Open Networking Foundation (ONF), 118
OpenFlow, 119
Operation
delete, 8, 12, 23, 57, 63, 65, 67, 70, 77, 82, 83, 110, 124, 140
deletion, 9, 10, 14, 16, 19, 52, 57, 59–62, 65, 67, 73, 83, 98, 113, 151, 156, 188
hashing, 58
insert, 70, 71, 73, 80–82, 181
insertion, 9, 42, 58, 62, 66, 67, 69, 71, 73, 80, 90, 96, 124, 127, 140, 150, 154
join, 85, 166, 170, 171
lookup, 9–12, 23, 28, 61, 63, 67, 90, 96
modulus, 37
query, 4, 40, 69–71, 73, 80–82, 90, 92, 110, 113, 124, 127, 129, 138, 148, 167, 171, 182, 200, 204, 207, 211
Overhead, 13, 17, 19, 52, 53, 56, 57, 62, 126, 129
synchronization, 126
Overlapping sequence set, 201

P
Pache, 93
Packet
buffer, 105
classification, 123, 124, 135
data, 4, 57, 102, 148
multicasting, 107
encoded, 134
information, 108, 127–129, 138, 147
Interest, 4, 101–103, 147
length, 140, 152, 153
malicious, 92, 147
management, 4, 89, 92, 98
named data networking, 102
notification, 124
preprocessor, 124
RouteQuery, 137
RouteReply, 137
TreeAck, 137
TreeQuery, 137
Packet Behavior Query Language (PBQL), 92
Pan-genomics, 210
Partial Segmented Bloom Filter Array (PSBFA), 165
Partial-key Cuckoo hashing, 62

Pattern Aware Secure Search tree (PASStree), 181
Payload, 92
Payload Attribution System (PAS), 152
Pending Interest Table (PIT), 101, 107, 108
Performance
  insertion, 28, 40
  lookup, 28
  query, 19, 37, 40, 45
  robustBF, 3, 25, 28, 34, 40
Pipeline, 209
Platform, 16
Pollution attack, 150
Popper switch, 95
Port scanning, 139, 140, 152
  traffic, 138, 152
Prefix matching, 96
Preprocessing filtering, 198
  schemes, 198
Press-on-towards, 154
Primary Bloom Filter, 209
Prime numbers, 38
Privacy, 111, 126, 153, 156, 168, 170, 179, 183, 190
  cache, 111
  content, 111
  data, 154, 176, 182
  name, 111
  signature, 111
Privacy-preserving record linkage, 168
Probabilistic Bloom Filter (PBF), 92
Probabilistic data structure, 7, 8
Probabilistic data structure-based Intrusion detection system (ProIDS), 151
Progressive Bloom Filter (PBF), 209
Protocol oblivious forwarding (POF), 118
Pseudonymous Certification Authority (PCA), 138, 151

Q
Quasi-identifiers, 168
Query
  membership, 80
  multiset, 80
  operation, 4, 40, 80, 124, 138
    cost, 81
  performance, 19, 37, 40, 45
  time complexity, 72, 73, 81
  traffic, 51
  true response, 51, 56
Query Index (QI), 62
Query-only attack, 151
Queue Bloom Filter (QBF), 123
Quotient, 73
Quotient Filter (QF), 63
Quotienting, 63

R
RAM-based Bloom Filter, 16
Random Set (RS), 27, 29, 40
Read, 5, 198
  compression, 201
Recurrent neural network (RNN), 124
Reliability, 126
Remainder, 73
  filter, 206
Resource record (RR), 146
Response
  Bloom Filter, 11, 57, 138, 151, 153
  membership, 86
  true, 79, 81
Retouched Bloom Filter (RBF), 58
Reversible multilayer hashed counting Bloom Filter, 82
robustBF, 3, 23, 25, 28, 29, 33, 34, 39, 41, 42
  memory requirements, 33, 37
  performance, 3, 25, 28, 34, 40
RouteQuery packet, 137
Router, 94
  NDN, 105
RouteReply packet, 137
Routing, 94

S
Same Set (SS), 27, 29, 39
Sandwiched learned Bloom Filter, 54
Scalability, 13, 77, 83, 85, 126
Scalable Bloom Filter (SBF), 58
SCBF, 69, 107
  filter, 69
SDN, 128
  architecture, 117, 129
  network, 118, 119, 125, 126, 129
  planes, 117
Searching, 96, 206
  data, 176
Secondary Bloom Filter, 209
Secure hash functions, 37, 49
Secure set intersection (SSI), 169
Security, 107, 110, 126, 135, 151, 154, 175, 190
  network, 4, 51, 145, 156, 157
  policies, 117
Segmented Bloom Filter, 201
Sequence Bloom Tree (SBT), 206
Sequence Read Archive (SRA), 202, 206
Session Initiation Protocol (SIP), 150
Signature privacy, 111
Similarity filter, 206
Single nucleotide polymorphism (SNP), 210
Single point of failure, 126
SKGframes, 136
Southbound, 118
Space complexity, 77, 83
Space-Code Bloom Filter (SCBF), 69
Spectral Bloom Filter (SBF), 57, 81
Split Bloom Filter, 206
Split Sequence Bloom Tree (SSBT), 206
SQL Slammer worm, 146
SRAM filter, 61
Stable Bloom Filter, 58
Standard Bloom Filter, 23, 27, 57
Static Bloom Filter, 15
Static Random Access Memory (SRAM), 91
Structure-preserving feature rearrangement, 190
Subnet, 149
Subnetwork, 140
  gateway, 140
Summary store, 103
Support Vector Machine (SVM), 192
Synchronization overhead, 126

T
Tag, 123
Taxonomy of Bloom Filter, 13
Taxonomy of response, 11
TCP SYN flood, 146
Temporal counting Bloom Filter (TCBF), 72
Ternary Bloom Filter (TBF), 61
Time Complexity, 83
Time-out Bloom Filter (TBF), 93
Timer array (TA), 71
Track Server (TS), 106
Traffic
  data, 52, 56
    filtering technique, 134
  engineering, 119
  management, 89, 94, 97
  port scanning, 138, 152
  query, 51
  statistics, 90, 97, 119
TreeAck packet, 137
TreeQuery packet, 137
True
  negative, 11, 13, 19, 24, 52, 53
  positive, 11, 13, 19, 24, 52–54
  response, 79, 81
    queries, 51, 56

U
UDP flood, 146
Urgent Member Filter (UMF), 133
User data, 153, 175, 176, 179–181

V
Variable-length counting Bloom Filter, 70
Vehicular Ad hoc Network (VANET), 131, 137
VI-CBF, 59
Virtual network, 94
Virtual private network (VPN), 94
Vulnerability attack, 145

W
Weighted Bloom Filter (WBF), 58
Whitelist, 146, 147
Wireless communication, 131
Wireless network, 4
Wireless Sensor Network (WSN), 131, 136
Wraparound counting Bloom Filter (WCBF), 73, 146, 148
WSN network, 135

X
XOR Filter (XF), 23, 32–34
xxHash, 38
  function, 48

BLOOM **FILTER**

A Data Structure for Computer Networking, Big Data, Cloud Computing, Internet of Things, Bioinformatics and Beyond

Bloom Filter: A Data Structure for Computer Networking, Big Data, Cloud Computing, Internet of Things, Bioinformatics, and Beyond focuses on both the theory and practice of the most emerging areas for Bloom filter application, including Big Data, Cloud Computing, Internet of Things, and Bioinformatics. Sections provide in-depth insights on structure and variants, focus on its role in computer networking, and discuss applications in various research domains, such as Big Data, Cloud Computing, and Bioinformatics.

The conventional Bloom filter is a probabilistic data structure for a membership filter. Burton Howard Bloom introduced an approximate membership filtering data structure in 1970. Hence, it is called a Bloom filter. Since its inception, it has been extensively experimented with and developed to enhance system performance such as web cache. Bloom filter influences many research fields, including Bioinformatics, Internet of Things, computer security, network appliances, Big Data, and Cloud Computing. This book is a key resource for computer and data scientists, researchers and students in biomedical engineering and applied informatics, database architects and database management researchers.

Key Features

- Includes Bloom filter methods for a wide variety of applications
- Defines concepts and implementation strategies that will help the reader to use the suggested methods
- Provides an overview of issues and challenges faced by researchers

About the Authors

Ripon Patgiri, Assistant Professor, Department of Computer Science and Engineering, National Institute of Technology, Silchar, India

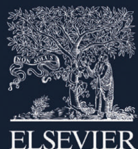
Dr. Ripon Patgiri is an Assistant Professor at the Department of Computer Science & Engineering, National Institute of Technology Silchar, since 2013. His research interests include Bloom filters, storage systems, security, and cryptography computing. He has published numerous papers in reputed journals, conferences, and books. Also, he has been awarded several international patents. Dr. Patgiri is a senior member of IEEE. He was the General Chair of ICACNI 2018 and BigDML 2019. He was also the Organizing Chair of FRSM 2020 and ADCOM 2020, as well as the Program Chair of CoMSO 2020, CoMSO 2021, and CoMSO 2022. Dr. Patgiri was an editor of several multi-authored books. Moreover, he has received two research project funding grants from SERB and DST, India.

Sabuzima Nayak, Research Scholar, Department of Computer Science and Engineering, National Institute of Technology, Silchar, India

Sabuzima Nayak has published numerous papers in reputed journals, conferences, and books. Her research interests include bioinformatics, Bloom filter, Big Data, and distributed systems.

Naresh Babu Muppalaneni, Assistant Professor, Department of Computer Science and Engineering, National Institute of Technology, Silchar, India

Dr. Naresh Babu Muppalaneni is the author of several books in the field of Computational Intelligence and Bioinformatics, including Computational Intelligence Techniques in Diagnosis of Brain Diseases, Soft Computing and Medical Bioinformatics, Computational Intelligence in Medical Informatics, and Computational Intelligence Techniques for Comparative Genomics, as well as Computational Study on Protein–Ligand Interactions for Anti-Diabetic: In Silico Study. He is a Senior Member of IEEE, and his research interests include Machine Learning, Computational Systems Biology, bioinformatics, Artificial Intelligence in Biomedical Engineering, applications of intelligent system techniques, image processing, and social network analysis.



ACADEMIC PRESS
An imprint of Elsevier
elsevier.com/books-and-journals