A Novel Technique for Efficient Storage and Retrieval of Massive Data Sets

*A **Thesis***

submitted in partial fulfillment of the requirements for the award of degree of

Doctor of Philosophy

by

Amritpal Singh

(901403023)

under the guidance of

Dr. Shalini Batra

Associate Professor

Computer Science and Engineering Department

Thapar Institute of Engineering and Technology (Deemed University)

Patiala -147004, INDIA

January 2018

Contents

List of Figures . . . vii

List of Tables . . . viii

List of Algorithms . . . ix

List of Abbreviations . . . x

Certificate . . . xiii

Acknowledgements . . . xiii

Abstract . . . xv

1 Introduction **1**

1.1 Big Data Evolution . . . 1

1.1.1 Background . . . 1

1.1.2 Big Data: Definition . . . 3

1.1.3 Characteristics of Big Data . . . 4

1.2 Current Scenario . . . 8

1.2.1 Streaming Data . . . 8

1.2.2 The Internet of Things . . . 10

Social IoT . . . 11

1.3 Probabilistic Data Structures . . . 12

1.3.1 Role of Probabilistic Data Structures in Big Data . . . 13

1.4 Applications of Big Data 15
1.5 Thesis Organization 18

2 Literature Review **20**

2.1 Hashing Techniques 21
2.1.1 Double Hashing 22
2.1.2 d-left Hashing 22
2.1.3 Partitioning Hashing 23
2.2 Approximate Membership Query 24
2.2.1 Bloom Filters 24
Categories of Bloom Filter 27
2.2.2 Role of Bloom Filter in Streaming Data 36
2.2.3 Role of Bloom Filter in Duplicate Detection 36
2.2.4 Applications of Bloom Filter 40
2.2.5 Quotient Filter 40
Advantages of QF over BF 44
Disadvantages of QF in comparison to BF 45
2.2.6 Applications of Quotient Filter 45
2.3 Similarity Search 46
2.3.1 Min Hash 48
Steps in Min-hash: 48
2.3.2 Locality-Sensitive Hashing 49
LSH with Hamming distance 52
LSH with Jacard Similarity 52
LSH with Euclidean Distance 53
LSH with Cosine Distance 53

2.3.3 Applications of LSH 54
2.3.4 Spam Detection in Social IoT 54
2.4 Problem Formulation 58
2.5 Objectives 58

3 Streamed Data Analysis Using Adaptable Bloom Filter 60

3.1 Introduction 60
3.1.1 Estimation Models 61
Kalman Filter 61
Discrete Kalman Filter 62
3.2 Adaptable Bloom Filter 63
3.2.1 Input and Hashing Phase 65
3.2.2 Storage Phase 66
3.2.3 Query in ATBF 68
3.2.4 Learning Array 70
3.2.5 Network Traffic Analysis for a Particular Time Slot 73
3.3 Observations and Analysis 75
3.3.1 Experiment 1: Uber Pickups Data Sets 76
3.3.2 Experiment 2: Incoming Data on a Network Server 78
3.3.3 Performance Evaluation of scalable BF and ATBF 80
3.4 Discussion 82

4 FingerPrint Based Duplicate Detection in Streamed Data 84

4.1 Duplicate Detection in Streaming Data 84
4.1.1 Problem Statement 84
4.2 FingerPrint Stable Bloom Filter 85

4.2.1 Detection 86

4.2.2 Deletion or Eviction of Data 88

4.2.3 Insertion 89

4.2.4 Stable Property 91

4.2.5 False Positives Analysis 93

4.2.6 False Negatives Analysis 95

4.3 Observation and Analysis 97

4.3.1 Theoretical Analysis 97

4.3.2 Experiment Evaluation 98

4.3.3 Application Domains 102

4.3.4 Discussion 103

5 Eb-SDF: Ensemble based Spam Detection Framework 105

5.1 Spam Detection 105

5.2 Spam Detection in Social IoT 107

5.2.1 Hybrid Sampling 108

5.3 Classifiers Considered in Eb-SDF 110

5.3.1 Classifier I: Blacklisted Domains Detector or URL checker 110

5.3.2 Classifier II: Near Duplicate Detector 113

5.3.3 Classifier III: Reliable Ham Tweet Detector 114

5.3.4 Classifier IV: Multi Model Classifier 116

5.3.5 Model Updation 118

5.4 Experimental Analysis of Ensemble based Spam Detection in Social IoT . 121

5.4.1 Performance Evaluation Metrics 122

5.4.2 Observations and Analysis 122

5.4.3 Discussion 127

6 Conclusion and Future Scope **129**

6.1 Conclusion . . . 129

6.2 Thesis Contributions . . . 130

6.3 Future Scope . . . 130

References **131**

List of Publications **146**

List of Figures

1.1 Characteristics of Big data 5

1.2 Variants of PDS 14

2.1 Insertion in Bloom filter 25

2.2 Query in Bloom filter 25

2.3 Categories of Bloom filter 28

2.4 Quotient filter work flow 42

2.5 Probability v/s distance measure in locality sensitive hashing 50

2.6 Locality-Sensitive Hashing framework 51

3.1 Framework for AdapTable Bloom Filter (ATBF) 64

3.2 An instance from data set of Uber pickups 76

3.3 Computational time complexity vs. number of hash functions 80

3.4 No. of slices required vs. number of iteration for dynamic dataset 81

3.5 Worst case query complexity vs. number of iterations 82

4.1 Structure of proposed Finger Print based Stable Bloom Filter 86

4.2 FP-SBF: Flowchart of detection process 87

4.3 FP-SBF: Flowchart of eviction process 89

4.4 FP-SBF: Flowchart of insertion process 90

4.5 Variation in false negatives with different parameters 99

4.6 Stable point 100

4.7 Variation in false positives with different parameters 101

5.1 Framework for Ensemble based Spam Detection (Eb-SBF) 109

5.2 Computational time (ms) for LSH and hash indexing with increase in number of tweets 123

5.3 Search time (ms) for QF, hash table and B+ tree 124

5.4 False positive rate vs. number of elements in Quotient Filter 124

5.5 Number of iterations vs. time (in hours) required for optimized updation . . 125

5.6 Precision of classifiers(I-IV) vs. number of iterations 125

5.7 Recall of classifiers(I-IV) vs. number of iterations 126

5.8 F1 score of classifiers(I-IV) vs. number of iterations 126

List of Tables

2.1 Variants of Bloom Filter 33

2.2 Bits significance in QF 43

2.3 Comparative analysis of all PDS studied 55

3.1 Nomenclature for Kalman filter 62

3.2 BF size prediction and Peak hour analysis for UBER pickup call for 1-Jan-2015 and 2-Jan-2015 77

3.3 Hourly analysis of Server Utilization, Peak Hour and BF size prediction for next time slot for incoming data on a network 79

List of Algorithms

3.1 ATBF: Insertion procedure 68

3.2 ATBF: Querying in proposed framework 69

3.3 ATBF: Learning array algorithm 73

3.4 ATBF: Approximate number of elements 75

4.1 FP-SBF: Duplicate detection in streams 86

4.2 FP-SBF: Detection process 88

4.3 FP-SBF: Deletion process 89

4.4 FP-SBF: Insertion process 90

5.1 EbSDF: PDS Based Classifiers in Ensemble Framework 117

5.2 EbSDF: Model Updation Algorithm 121

List of Abbreviations

ABF	Ageing Bloom Filter
ATBF	AdapTable Bloom Filter
BF	Bloom Filter
CBF	Counting Bloom Filter
CM	Characteristic Matrix
CR	Convergence Rate
DVS	Deletion Value Set
Eb-SDF	Ensemble based Spam Detection Framework
ER	Eviction Rate
FN	False negative
FP	False Positive
FPC	FingerPrint Cell
FP-CBF	FingerPrint Counting Bloom Filter
FP-SBF	FingerPrint Stable Bloom Filter
HS	Hybrid Sampling
IBF	Incremental Bloom Filter
IoT	Internet of Things
KF	Kalman Filter
LA	Learning Array
LRU	Least Recently Used
LSH	Locative Sensitive Hashing
MH	Min Hash
NN	Nearest Neighbor
PDS	Probabilistic Data Structure
QF	Quotient Filter
RSBF	Reservoir Sampling based Bloom Filter
SBF	Static Bloom Filter
SIoT	Social Internet of Things

SM	Signature Matrix
SP	Stable Property
SQF	Streaming Quotient Filter
VI-CBF	Variable Increment Counting Bloom Filter

Certificate

I hereby certify that the work which is being presented in this thesis entitled "**A Novel Technique for Efficient Storage and Retrieval of Massive Data Sets**", in partial fulfillment of the requirement for the award of degree of "**Doctor of Philosophy**" submitted in Computer Science and Engineering Department, Thapar Institute of Engineering and Technology (Deemed University), Patiala (India), is an authentic record of my own work carried out under the supervision of **Dr. Shalini Batra** and refers other research works which are duly listed in the reference section.

The matter presented in this thesis has not been submitted for the award of any other degree of this or any other university.

(**Amritpal Singh**)

Regn. No. 901403023

This is to certify that the above statement made by the candidate is correct and true to the best of my knowledge.

(**Dr. Shalini Batra**)

Associate Professor

Computer Science & Engineering Department

Thapar Institute of Engineering and Technology (Deemed University)

Patiala, 147004

Punjab, INDIA.

Acknowledgements

First and foremost, I would like to thank **Almighty**, for being my strength and pillar (as a humble being) to complete this task successfully. He has given me an opportunity to believe in my passion and pursue my dreams. I could never have done this without his divine blessings.

I would like to express my sincere appreciation to my supervisor, **Dr. Shalini Batra**, Associate Professor, Computer Science and Engineering Department, Thapar Institute of Engineering and Technology (Deemed University), Patiala (India), for being a pillar of support and encouragement throughout my research work. Her guidance helped me in all the time of research and writing this thesis. Her experience, strength, tenderness and willfulness, has taught me valuable lessons of life, which are going to be of immense help to me in taking decisions in going forward. I could not have imagined having a better advisor and mentor for my Ph.D study.

My sincere thanks are due to **Dr. Maninder Singh**, Professor and Head, Computer Science and Engineering Department, Thapar Institute of Engineering and Technology (Deemed University), Patiala (India), for providing me the necessary administrative assistance in completion of the work. I am thankful to my Ph.D. committee members, **Dr. Prashant Singh Rana** and **Dr. Vijay Kumar**, Assistant Professors, Computer Science and Engineering Department, and **Dr. M.D. Singh**, Assistant Professor, Electrical and Instrumentation Engineering, Thapar Institute of Engineering and Technology (Deemed University), Patiala (India), for their constructive comments and regularly ensuring the progress of my research work. I am thankful to all the **faculty** and **staff members** of Computer Science and Engineering Department, Thapar Institute of Engineering and Technology (Deemed University), Patiala (India),

for their support especially by **Dr. Neeraj Kumar**.

A very special gratitude goes out to **Ministry of Electronics and Information Technology Government of India** for providing financial support for the work.

I offer my deepest gratitude to my loving mother, **Mrs. Paramjeet Kaur** and father, **Mr. Ram Singh** whose love and affectionate blessings have been a constant source of inspiration in making my vision a reality. I am also thankful to my wife **Mrs. Mandeep Kaur** and all members in my family (especially to **Harpreet Kaur** and **Pritpal Singh**) for their love, motivation, encouragement and their confidence in me.

I dedicate this Ph.D. thesis to the little flowers of the family, my daughter **Anoop Kaur** and nephew **Shehbaaz Singh**, who are the pride and integral part of my life, always cheering me up during the entire phase of Ph.D.

I also acknowledge the cooperation and encouragement extended to me by my friends at Thapar University, especially **Ms. Monika Bharti**, **Mr. Abhishek Khana** and **Mr. Himashu Jindal**

Patiala

Jan, 2018 (Amritpal Singh)

Abstract

In today's world data is considered as one of the most valuable assets. With the coming up of plethora of web applications and technologies like sensors, IoT, cloud computing, *etc.*, the in-stream data generation resources have increased exponentially. Data originating from heterogeneous sources and real world applications is severely susceptible to inconsistent, incomplete and noisy data. To support data applications in different domains, data processing must be efficient and automated as much as possible. Further, timely and accurate analysis of available data is an intrinsic requirement.

Conventional databases and traditional data mining techniques are efficient for stored data analytic but for in-streamed data, where data is arriving continuously, it is not feasible to store the data into databases and then perform analysis since all such applications demand time bound query output. Moreover, traditional approaches demand that entire data should be stored in a formatted manner. Massive datasets require architectures and tools for data storage, handling, processing and mining of the bulk information in limited time and in single pass. One of the available alternative is use of Probabilistic Data Structures (PDS) in Big data analytics, which use some probability based approaches, approximation principals and hashing methods to reduce time and space trade off in storage, retrieval and search of data.

This thesis proposes three techniques for streamed data analysis. First one, a variant of scalable Bloom Filter (BF), called AdapTable Bloom Filter (ATBF), performs peak hour analysis and decides the size of dynamic BF apriori using Kalman filter and Learning Array (LA). In second approach, a variant of stable BF, called FingerPrint Stable Bloom Filter(FP-SBF), has been proposed for duplicate detection in streamed data. In the third approach, a semi-supervised technique for spam detection in Twitter has been proposed which employs ensemble based framework (Eb-SDF) comprising of four classifiers. The framework is based

on usage of PDS like Quotient Filter (QF) to query the URL database, spam users, spam words databases and Locality Sensitive Hashing (LSH) for similarity search.

Performance of the proposed approaches has been evaluated by comparative analysis of PDS with the similar data structures and through the standard evaluation parameters. ATBF has been compared with scalable BF for server utilization and hourly load analysis. FP-SBF has been compared with stable BF and reservoir based sampling BF and accuracy is determined for detecting duplicates in streaming data. Results are compared on different BF parameters which include counter size, size of bloom filter, number of hash functions, false positive and false negative analysis, *etc.* Eb-SDF has been tested on twitter dataset and comparative analysis is performed on the basis of precision, recall and F1- score.

Chapter 1

Introduction

The term Big data appeared sparingly in the early 1990s, and its prevalence and importance increased exponentially as years passed. Now a days Big data is often seen as an integral part of computer science research and development. Following section explains the evolution of Big data in the last three decades.

1.1 Big Data Evolution

1.1.1 Background

In late 1980's, a new business analysis idea of decision support and data warehousing was introduced to the industry. This analysis provided predictive results on the basis of trend in historical data and helped in selecting the optimal solution with least risk [1]. Later on, this business idea become so popular that it become a new industry in itself and world was introduced to the term 'Big data' [2].

As advancement in technology and communication protocols moved further, commercial internet was launched in 1995 leading to exponential increase in data generation resources [3]. Internet made sharing of information easy and communication became very fast leading

to worldwide revolution in technology, generation of new standards and business models. In terms of Big data, it can be rightly said that large new resources for data were generated which provided complex data in terms of volume and variety [4].

From 1997 to 2002, cellular mobile network redefined the mobility solutions. Ease in using them attracted more people and industries towards this change. Many industries started providing new products and services to mobile phone users, making them more interesting, resulting in more data complexity in terms of volume, variety, velocity and usage of data [5].

From 2000 to 2010 [2, 6], next big technical revolutions was witnessed where Internet giant Google and smart phones from Apple came up, fast and new technologies like 3G and Wi-Fi were initiated and social media revolution by Twitter and Facebook made this decade very important from the view point of Big data. All these changes redefined the traditional scientific methods, paving way for emergence of a new era of Big data processing [5].

With the coming up of plethora of web applications and technologies like IoT [7], cloud computing [8], *etc.*, the data generation resources are increasing exponentially. From the last few years, there is an exponential increase in the data movement across different smart devices with respect to network activities such as-search requests, logs, location data, tweets, e-commerce, data footprint of individuals, *etc* . The amount of data being produced everyday has increased from terabytes to petabytes [9].

Two major breakthrough helped to accelerate the solutions generation for Big data: one was cloud computing [6] which decreased the cost of storage significantly and amplified use of commodity hardware and another was distributed computing for data storage at different servers [10] that helped in new data base designs leading to new generation products like Hadoop, Map-reduce, No-SQL, *etc* [11].

1.1.2 Big Data: Definition

Initially, Big data refereed to the collection of huge amount of unstructured data (volume and variety) only. But, with the rise in continuous data generation resources like traffic data, GIS data, climate data, stock market data *etc.*, term velocity was introduced in Big data. Thus, Big data become collection of huge volume of data, including the complexity in terms of fast data generation rate (velocity) and variable structure of data (variety) [12, 13]. Their is no standard definition of Big data, it depends on the domain application and environment. Amongst the most cited definitions, some of them used to describe Big data by different sources are listed below:

- *Gartner* proposed a three fold definition in 2001, "Big data comprises of three Vs: *Volume, Velocity, Variety*. The primary focus of Big data is on the increasing size of data, the increasing rate at which it is produced and the increasing range of formats and representations employed" [14].
- *Oracle* stated that: "Big data is the derivation of value from traditional relational database driven business decision making, augmented with new sources of unstructured data." They assert that Big data is the inclusion of additional data sources to augment existing operations [15].
- *IBM* data scientists referred to Big data with four dimensions: *volume, variety, velocity and veracity*. "Big data gives you the ability to achieve superior value from analytics on data at higher volumes, velocities, varieties or veracities. With higher data volumes, one can take a more holistic view of the past, present and likely future. At higher data velocities, one can ground your decisions in continuously updated, real-time data. With broader varieties of data, one can have a more nuanced view of the matter at hand. And as data veracity improves; one can be confident that, working with the

truest, cleanest, most consistent data" [16].

- According to *Microsoft*: "Big data is the term increasingly used to describe the process of applying serious computing power - the latest in machine learning and artificial intelligence - to seriously massive and often highly complex sets of information" [17].

- *Google* referred to Big data as: "Data that would typically be too expensive to store, manage, and analyze using traditional (relational and/or monolithic) database systems. Usually, such systems are cost-inefficient because of their inflexibility for storing unstructured data (such as images, text, and video), accommodating high-velocity (real-time) data, or scaling to support very large (petabyte-scale) data volumes" [18].

- *Wikipedia* describes Big data as: "An all-encompassing term for any collection of data sets so large and complex that it becomes difficult to process using on-hand data management tools or traditional data processing applications" [19].

- *Casari* explain Big data in technical perspective as: "Big data is data whose scale, diversity, and complexity require new architecture, techniques, algorithms, and analytics to manage it and extract value and hidden knowledge from it" [20].

- According to *Forbes*, "It is a new tools which helps us find relevant data and analyze its implications" [21].

1.1.3 Characteristics of Big Data

Big data is often described using Vs which include Velocity, Volume, Variety, Veracity, Value *etc.* (Fig. 1.1). Characteristics of Big data along with challenges associated with them are described below :

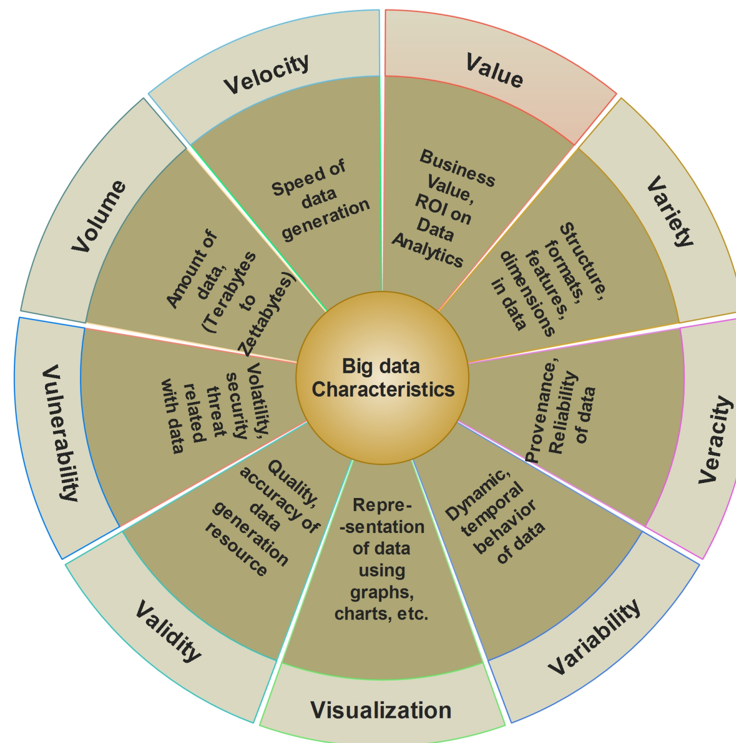

Figure 1.1: Characteristics of Big data

- *Velocity:* First V refers to the speed at which data is being generated, created, refreshed, collected and analysed. With the every passing day, the number of emails, twitter messages, photos, video clips, *etc.* are increasing manifold. Facebook claims 600 terabytes of incoming data per day [22]. Google alone processes on average more than "40,000 search queries every second," which roughly translates to more than 3.5 billion searches per day [23].

- *Volume:* It refers to the incredible amounts of data generated each second from social media, cell phones, credit cards, sensors, photographs, video, *etc.* The size of the data is not just limited to Terabytes but it has already reached Zettabytes. Big data is probably best known for Volume of data; more than 90 percent of all today's data was created in the past couple of years. Facebook alone has 10 billion messages, 4.5 billion times their "like" button is pressed, and over 350 million new pictures are

uploaded every day [24]. YouTube registers 300 hours of video uploads every minute [25]. In 2016 estimated global mobile traffic amounted to 6.2 exabytes per month [23]. Collecting and analyzing this data is clearly a big challenge. This voluminous data has become so large that storing and analysing data using traditional database technology seems infeasible. New Big data tools which use distributed systems to store and analyse data across databases located anywhere in the world is need of the hour.

- *Value:* This V characterizing the business value, ROI, and potential of Big data to transform an organization. All other characteristics of Big data are meaningless if one does not derive business value from the data. Value ensures that the effort and resources utilized for Big data analytics will be valuable for organization. Having endless amounts of data is one thing, but unless it can be turned into value it is useless. Substantial value can be found in Big data, including understanding your customers better, targeting them accordingly, optimizing processes, and improving machine or business performance.

- *Variety:* Variety means different types of data used for Big data analysis. It covers all aspects related to structure of data like complexity, number of features related with the data item, combinatorial explosion, variety of data types, and various data formats. When it comes to Big data, it includes structured, semi-structured and unstructured data. Majority of the data which includes audio, image, video files, social media updates, and other text formats like log files, click data, machine and sensor data, *etc.* seems to be unstructured. Thus, analysis of variety of the available data is one of the biggest challenges of Big data. Organizing the data in a meaningful way is no simple task, especially when the data changes rapidly.

- *Veracity:* Veracity refers to the provenance or reliability of the data source, its context, and how meaningful it is to the analysis. Basically it represents the quality or trustworthiness of the data. As Volume, velocity and variety increase, the veracity (confidence or trust in the data) drops. Knowledge of the data's veracity in turn helps us better understand the risks associated with analysis and business decisions

- *Variability:* Variability refers to the constantly changing meaning of data. It deals with dynamic, evolving, spatio temporal data, time series, seasonal, and any other type of non-static behavior in data sources, customers, objects of study, *etc.* Variability in Big data's context refers to different things. One is the number of inconsistencies in the data. These can be analysed by anomaly and outliers detection methods to provide meaningful analytic. Multitude of data dimensions resulting from multiple disparate data types and sources also comes under variability of data. Variability can also refer to the inconsistent speed at which massive data is loaded into the database.

- *Visualization:* Visualization is critical in today's world. Using charts and graphs to visualize large amounts of complex data is much more effective in conveying meaning than spreadsheets and large reports. Combine this with the multitude of variables resulting from Big data's variety and velocity and the complex relationships between them, and one can see that developing a meaningful visualization is not easy. Current Big data visualization tools face technical challenges due to limitations of in-memory technology and poor scalability, functionality, and response time.

- *Validity:* Validity deals with data quality, governance, master data management on massive, diverse, distributed, heterogeneous, "unclean" data collections. Similar to veracity, validity refers to how accurate and correct the data is for its intended use. According to Forbes, an estimated 60 percent of a data scientist's time is spent cleansing

their data before being able to do any analysis [26]. The benefit from Big data analytics is only as good as its underlying data, so one needs to adopt good data governance practices to ensure consistent data quality, common definitions, and metadata.

- *Vulnerability:* Big data brings new security concerns. Due to the velocity and volume of Big data, however, its volatility needs to be carefully considered. As reported by CRN: "In May 2016 a hacker called Peace, posted data on the dark web to sell, which allegedly included information on 167 million LinkedIn accounts and 360 million emails and passwords for MySpace users [27]."

1.2 Current Scenario

Advancements in application areas of IoT [28], social networks [29] and social IoT [30] have shifted the focus of Big data analytics towards streaming data. The terms volume, variety in Big data has changed paradigm of Big data from business solution to data mining [31] and now with the contribution of velocity, paradigm is shifted to streamed data analytics [32]. The following sections provide a brief introduction of to the domains of Big data covered in this thesis, *i.e.*, streaming data analytics and IoT; where techniques have been proposed for efficient storage and retrieval of massive data.

1.2.1 Streaming Data

Streamed data arriving from various resources requires fast processing and storage framework for handling huge amount of data. Given millions or even billions of data elements, developing efficient solutions for storing, updating, and querying them becomes increasingly important especially when data is available for short span [33]. Storing entire data requires lot of memory and usage of fixed size data structures will require lot of time for analysis

[34]. Moreover, the complexity of data and the amount of noise associated with the data is not predefined since the size of data is also unknown, one cannot determine how much memory is required for storage. The important issue in stream processing is that data diminishes with time so data must be processed in a particular time window in single pass [35]. Thus, it becomes difficult to capture, store and process the incoming data within the stipulated time. [36].

Some of the applications which need special attention in the real time analytics of streaming data include heavy hitters in data streams, frequency query for all items in the set, estimate the cardinality of massive dataset, find similar items in huge pool of items, membership query, *etc* [37]. In many application domains, fast and real time processing is required to make timely decisions accurately for *eg.* in finance sector, the analysis of stock market streaming data is an essential tool for predicting stock price of the companies and real time fraud detection in short time span [38]. In dynamic recommender applications, processing of streamed data is necessary for referral of products according to interest of user and promotion of new products in the market [39]. In network applications, managing data streams for system monitoring can be time-varying, volatile and unpredictable since tasks to be managed include accessing the server's utilizations in particular time frame, tracking the number of unique visitors on a network in a particular time, identifying common users between two time slots, or calculating maximum number of hits on network in particular span of time [40]. Results of network analysis can help to predict the resource usage over network, identify rush hours in network, management of network resources on the time slot basis and detect attacks like DoS and DDoS [41].

1.2.2 The Internet of Things

The Internet of Things (IoT) paradigm refers to a network of interconnected things. As the name indicates, IoT refers to connecting all things in the world to the internet [42]. The perception is that all the things will be identified automatically, will communicate with each other, and even make decisions by themselves. Thus, IoT is evolving as a new generation of information network and service infrastructure, amalgamating computer based systems with the corporal world. It is increasingly becoming a ubiquitous computing service, requiring huge volumes of data storage and processing. The growing number of Internet-connected devices, which is predicted to reach 20 billion units in 2020 has triggered a new breed of applications and services, a class of applications where the logic is powered by data and resources from the physical world [43].

Basic structure of IoT is composed of physical objects embedded with electronics, software, and sensors, which allows objects to be sensed and controlled remotely across the existing network infrastructure, facilitates direct integration between the physical world and computer communication networks, and significantly contributes to enhanced efficiency, accuracy, and economic benefits [44].

The term IoT was initially proposed to refer to uniquely identifiable interoperable connected objects with radio-frequency identification technology. Later on, researchers relate IoT with more technologies such as sensors, actuators, GPS devices, and mobile devices. Today, a commonly accepted definition for IoT is a dynamic global network infrastructure with self-configuring capabilities based on standard and interoperable communication protocols where physical and virtual 'Things' have identities, physical attributes, and virtual personalities and use intelligent interfaces, and are seamlessly integrated into the information network. IoT has been widely applied in various applications such as environment monitoring, energy management, medical healthcare systems, building automation, and transportation.

Social IoT

Internet revolution led to the interconnection between people at an unprecedented scale and pace. The next revolution will be the interconnection between objects to create a smart environment. A social approach can be used for the IoT to connect large number of objects in social networks like Twitter, Facebook, Instagram, *etc.* The network is normally intended as the IP network and the things are devices, such as sensors and/or actuators, equipped with a telecommunication interface and with processing and storage units. This communication paradigm should hence enable seamless integration of potentially any object into the Internet, thus allowing for new forms of interactions between human beings and devices, or directly between device and device, according to what is commonly referred to as the Machine-to-Machine communication paradigm or Social IoT (SIoT) [45].

The main aim of SIoT is to integrate the social network concepts into the IoT, use social networking elements in the IoT to allow objects to autonomously establish social relationships. The goal is to understand how the objects in the SIoTs process the information and build a reliable system based on the behaviors of objects. These massive datasets require architectures and tools for data storage, processing, mining, handling and leveraging of the information to offer better services.

The above discussed domains of Big data clearly indicate that in last few years, there is an exponential increase in the data generation resources which has provided a new direction to Big data wave. To handle this large volume of data, traditional algorithms cannot go beyond linear processing. Conventional databases and traditional data mining techniques are efficient for stored data analytic but for in-streamed data, where data is arriving continuously, it is not feasible to store the data into a database and then perform mining operations since all such applications demand time bound query output. Moreover, traditional approaches

demand that entire data should be stored in a formatted manner. These massive datasets require architectures and tools for data storage, processing, mining, handling and leveraging of the information to offer better services [46].

Sincere efforts by researches all over the world have been put up to extract intelligence out of this huge amount of knowledge base. Major bottleneck in this effort is that there exists no standard method to efficiently map and store the Big data on a compatible data structure. Further the memory required to store such huge bulk of enormous data and the computational time associated along with it for retrieval are increasing the difficulty in designing a standard framework for mapping huge data sets [47].

After having a thorough review of Big data and various approaches adopted for efficient retrieval task, it was realized that the deterministic data structures and algorithms had few limitations while dealing with massive datasets. One of the available alternative is use of PDS in Big data analytics, which has the advantage of hashing based approximation in retrieving fast results. Rapidly increasing data on Internet have raised the significance of analyzing the network traffic efficiently. Moreover, PDS also show significant improvement in results when applied on streaming data problems.
Next section (Section 1.3) provides detailed discussion of PDS and their application domains.

1.3 Probabilistic Data Structures

Enormous and unlimited growth of data has led to a paradigm shift in storage and retrieval patterns from traditional data structures to Probabilistic Data Structures (PDS).

Definition: *Probabilistic Data Structures* [48] are, tautologically speaking, data structures which have a probabilistic component. In other words, probabilistic data structures use some probability based approaches, approximation principals and hashing methods to reduce time

and space trade off in storage, retrieval and search of data.

These probabilistic components are used to reduce time or space tradeoffs. PDS cannot give a definite answer, instead they provide with a reasonable approximation a way to approximate these estimations. They are useful for Big data and streaming applications because they can decrease the amount of memory needed (in comparison to data structures that give exact answers). These data structures use hash functions to compactly represent a set of items in stream-based computing while providing approximations with error bounds so that well-formed approximations get built into data collections directly [49]. Further, they avoid high-latency analytical processes. Moreover, PDS offer several advantages which include:

- They use small amount of memory (one can control how much).
- They are easily parallelizable (hashes are independent).
- They have constant query time.

1.3.1 Role of Probabilistic Data Structures in Big Data

- PDS cannot give a definite answer, instead they provide with a reasonable approximation of the answer and a way to approximate this estimation. Comparing with error-free approaches, these algorithms use much less memory and have constant query time.
- They usually support union and intersection operations and therefore can be easily parallelized. These are useful for Big data and streaming applications because they can decrease the amount of memory needed (in comparison to data structures that give exact answers).
- The traditional way of dealing with large datasets is to use some kind of deterministic data structure like HashSet or Hashtable for such purposes. But when the data sets with

which application is dealing becomes very large, then deterministic data structures are not feasible because the data is too Big to fit in the memory. It becomes even more difficult for streaming applications which typically require data to be processed in one pass and perform incremental updates.

- One can chose a PDS according to requirement of application and get results is minimum time. Different variants of PDS categorized according to their application are highlighted in Fig. 1.2.

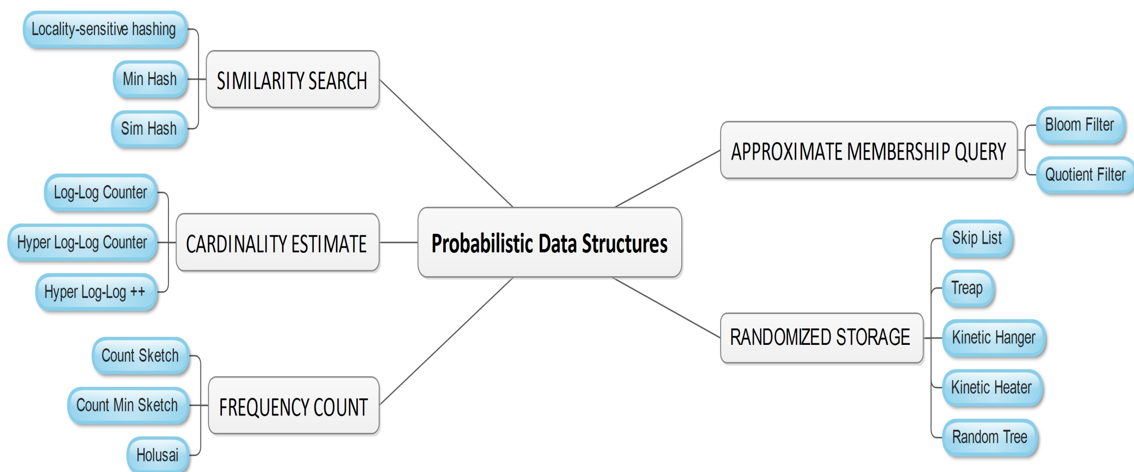

Figure 1.2: Variants of PDS

PDS plays major role in the following scenarios:

- *Approximate Membership Query:* Store bulk data in small space and respond to user's membership query efficiently in the given space S [50, 51].

- *Frequency Count:* Find cardinality, *i.e.*, number of cardinal (basic) members in a set in the massive data set [52, 53].

- *Cardinality Estimate:* Count the number of times a data item has arrived in the huge data sets [54, 55, 56].

- *Similarity Search:* Identify similar items, *i.e.*, find the approximate nearest neighbors (most similar) to the query in the available dataset [57, 58].

1.4 Applications of Big Data

Applications of Big data can be seen in many scientific disciplines like Healthcare, Economic Development, Public schemes for the society, Agriculture related datasets, Skill and Education, Weather and GIS, Biological and Genomic data, Social networks, Banking and Financial Markets, Remote Sensing, ad-hoc networks, privacy preserving, *etc* [59, 60, 61] . Some of the important fields are:

- *Health care*

  Big data is used for analyzing data in the electronic medical record system with the goal of reducing costs and improving patient care. This data includes the unstructured data from physician notes, pathology reports, *etc.* Big data and healthcare analytics have the power to predict, prevent and cure diseases [62]. It allows real-time monitoring of patients, which leads to proactive care. Sensors and wearable devices can collect patient health data even which can be monitored by health-care institutions to provide remote health alerts and lifesaving insights to their patients [63]. The most important aspect is health data acquisition, transmission, management and visualization; which help to build a social network in health care contexts. Further, domains like sensor based healthcare applications and medical image processing and visualization help in better prediction [64].

- *Public administration*

  The public sector is becoming increasingly aware of the potential value to be gained from Big data. Governments generate and collect vast quantities of data through their

everyday activities, such as managing pensions and allowance payments, tax collection, national health systems, recording traffic data, and issuing official documents. Using data analytics to benefit the public enabling policy makers to make policy proposals by understanding their public requirements [65]. Public welfare schemes can be implemented more efficiently if fraudulent and bogus identity holders are identified. Correlation of multiple sources of data will help government economists to come up with more accurate financial forecasts [66]. Information from both traditional and new social media (websites, blogs, twitter feeds, *etc.*) can help policy makers to prioritize services and be aware of citizens' interests and opinions [67].

- *Social network analysis*

  Social network analysis techniques can be applied to study structures of any types of interactions/relationships between any kinds of entities. Using SNA tools on collaboration and/or information-sharing networks, managers can easily find the "important person" and build appropriate management strategies to improve efficiency [29]. Combating terrorism is another field where SNA techniques have important and successful applications [68]. SNA techniques also have been successfully applied in epidemiology. Researchers are trying to analyze the spread of diseases based on the interactions between people. Number of recent successful applications of SNA include discovering emergent communities of interest amongst faculty at various universities; revealing cross-border knowledge flows based on research publications; determining influential journalists and analysts in the IT industry; map executive's personal network based on email flows, *etc.* [69].

- *Business*

  Big data is changing the competitive landscape of every business domain. Big data is used by organizations today for analyzing sentiments of the target customers and

providing them better services to increase the business. Analysis of such huge volume of data has also helped business in cutting down their expenditures in various sectors wherever possible. It also helps industries in demand forecasting; what kind of products to be developed to increase sales, to maintain position in market, *etc.* [70].

- *Remote Sensing*

  With the huge amount of data generated from sensors it is easy to predict weather forecast, drought indices, *etc.* Applications based on remote sensing image, such as agriculture, analysing climatic changes and its effects on the environment, land cover and land use planning, hydrology, *etc*, can be easily accomplished by Big data analytics [71].

- *Agriculture*

  Big data can help in increasing the production of high value seeds with help of machine learning models to understand the ecology and environment of a particular area. As part of green data revolution, analytics process can be applied to optimize the management, breeding and fertilization to increase the yields. Biotechnology revolution can also be an integrated part of data science through specifications of selection traits, chemistries and microbials [72].

- *Scientific research*

  Big data technology and services has formed a fertile foundation for scientific applications in the past several years. In biological and genomic data processing Big data have many applications like microarray gene analysis, mRNA expression profiling, precision medicine, biomarkers discovery, adverse drug reactions [73]. Web-based applications encounter Big data frequently, such as recent hot spots social computing (including social network analysis, online communities, recommender systems, repu-

tation systems, and prediction markets), Internet text and documents, Internet search indexing, *etc.* [13, 31]. Many research publication websites like ACM and Science Direct also have suggestion for reading research articles related to user's search query which help users to find papers related to their work areas. Big data is changing the media and entertainment industry, giving users and viewers a much more personalized and enriched experience. Big data is used for increasing revenues, understanding real-time customer sentiment, increasing marketing effectiveness and ratings and viewership [74].

1.5 Thesis Organization

The thesis has been organized in the following chapters:

Chapter 1: Introduction: This chapter provides an overview of Big data, it's characteristics, significance, and describe the application areas of Big data. It also introduce PDS and explain their role in Big data. Further, it explains the role of Big data in analytics in streaming data and IoT .

Chapter 2: Literature Review: This chapter provides a comprehensive review of PDS and highlights various domains where they are used. A comparative study of the existing PDS has also been included in this chapter. Major focus of this chapter is on PDS dealing with approximate membership query and similarity search domains.

Chapter 3: Streamed Data Analysis using AdapTable Bloom Filter (ATBF): Stream processing requires real time analytics of data in motion and that too in a single pass. In this chapter a framework proposed for hourly analysis of streamed data using bloom filter is discussed in detail. ATBF has been experimentally evaluated on various parameters like peak hour analysis, server utilization, *etc.*

Chapter 4: Duplicate Detection in Streaming Data: This chapter provides a scheme for duplicate detection in streaming data named as FingerPrint Stable Bloom Filter (FP-SBF), a variant of stable bloom filter. The performance of FP-SBF has been compared with various bloom filters used for stream data duplication detection and results achieved have been discussed.

Chapter 5: Storage and Retrieval of Massive Datasets in Social IoT: Social IoT has simplified the task of dynamic discovery of services and information. Detecting spam in social media, especially when massive data flows continuously requires fast processing and storage framework for handling huge amount of data and attributes associated with it. In this chapter, proposed semi-supervised ensemble based spam detection framework with the help of PDS is explained in detail.

Chapter 6: Conclusion and Future Scope

Thesis concludes with this chapter by highlighting the contributions made through the proposed research work. It also provides an insight into the future directions for working in this area.

Chapter 2

Literature Review

This chapter provides an extensive literature survey on Big data and various PDS along with there areas of applications.

Big data processing includes large volume of data processing in very small time and complex calculations in analysis part (value analysis and depth analysis). One of the major issues is optimal storage and retrieval of Big data volume with velocity along with variety in its data formats. Traditional data structures and algorithms are not sufficient in such a cases and some advanced approaches are required to fulfil Big data requirements [34]. Using traditional data base approaches which include performing filtering and analysis after storing the data is not efficient for real time processing of streamed data. Since the size of incoming data is unpredictable, data structures used for the storage of data should be dynamically adjustable, but changing size in each iteration may lead to extra computational overhead [47]. Further, data structures used for storage of massive data should provide efficient format for fast retrieval of data, as and when required. Here main concern in optimization is dynamic allocation of memory, fast search results with minimum computational cycles on the trade off some false positive and negative errors, which are predefined and user is ready to adjust these errors [49]. In applications where efficiency is more important than accuracy, use of

probabilistic approaches and approximation algorithms can serve as a key ingredient in data processing. Probabilistic methodologies provide quick answers with an allowable error rate compared to a deterministic approaches which give exact match which are slow and memory consuming [75]. Each PDS provides solution for a particular problem; like approximate membership query, similarity search, cardinality estimation, frequency estimation, *etc.*

Recent research directions in the area of Big data processing, analysis and visualisation clearly indicate the importance of PDS. These data structures use hash functions to compactly represent a set of items in stream-based computing while providing approximations with error bounds so that well-formed approximations get built into data collections directly. Probabilistic alternatives to deterministic data structures are better in terms of simplicity and constant factors involved in actual run-time, use much less memory and constant time in processing complex queries. They are suitable for large data processing, approximate predications, fast retrieval and storing unstructured data, thus playing an important role in Big data processing.

Since hashing is an integral part of PDS which decides the efficient working and accuracy of PDS, section 2.1 starts with important hashing techniques. In following sections, we elaborate the two major PDS applications domains which are considered in this thesis work, *i.e.*, Approximate Membership Query (Section 2.2) and Similarity Search (Section 2.3) along with details about PDS used and their applications in different domains.

2.1 Hashing Techniques

Hash functions are the key building block of PDS. Various hashing techniques which include Perfect Hashing, Double hashing, Multiple hashing, d-left hashing, partitioned hashing, *etc.*

are used according to different application domains. Specially in BF, hashing is an important aspect to determine the accuracy. In the coming section, some important hashing techniques (used in the proposed approaches) are discussed.

2.1.1 Double Hashing

Double hashing is a popular collision-resolution technique in open-addressed hash tables [76]. Collisions are resolved by comparing only two different hash values for the searched element. To generate hash values in double hashing, only two independent hash functions $I_1(x)$ and $I_2(x)$ are used to generate k hash functions such that $\forall i | i < k$. For an item $u_i \in U$, hash functions are calculated by:

$$ {}_{i=1}^{k}(H_i(u_i) = \{I_1(u_i) + (i \times I_2(u_i))\} \ mod \ \vartheta) \tag{2.1} $$

Major advantage of using only two hash functions to generate all k hash functions is that it decreases the computational overhead [77].

2.1.2 d-left Hashing

Another important hashing, d-left hashing refereed as perfect hashing technique, can be used to reduce the collisions by great extent [78]. d-left hashing is a minimal perfect hashing approach, where a hash table of size M is divided into d sub tables of size m_d, where $m_d = \frac{M}{d}$. Each element in sub table is refereed as bucket B_i and b denotes the maximum capacity of bucket. During insertion operation, k hash functions are used to compute the hash indexes and select the buckets. New item is placed in the bucket which is least loaded. If there is tie between two buckets then bucket on left side is selected to perform insertion operation. For an element $u_i \in U$, hash functions are calculated as:

$$\forall_{i=1}^{k}\Big(\kappa_i(u_i) = \ell_1(u_i) + i \times \ell_2(u_i)\ mod\ m\Big) \tag{2.2}$$

Each hash function $\kappa_i(.)$ selects i^{th} bucket from a sub array. Double hashing is used for hash function generation, where two independent hash functions $\ell_1(x)$ and $\ell_2(x)$ are used to generate k hash functions $s.t.$ $(\forall i | i < k)$. To get best results of this scheme, m should be prime, $\Phi(x)$ function is used to compute value of m [79]. $\Phi(x)$ returns an optimal number p, $s.t.$, $\big[p \leftarrow \Phi(p \geq x \text{ and p is prime})\big]$. This technique leads to less inter-hash function collision; further, usage of only two hash functions to generate all k hash functions decreases the computational and pre-processing overhead. Theoretical and experimental evidences prove that it helps in reduction of collision by a huge factor [80].

2.1.3 Partitioning Hashing

Partitioning hashing is a technique where small portion of large table is uniquely allocated to each hash function such that hash key l_i generated by hash function, ℓ_i, is randomly distributed over a small part of the array, *i.e.*, each hash function is allocated to a sub part of an array [81]. In BF, an array of m bits is partitioned into k disjoint arrays of size $\left(\vartheta = \frac{m}{k}\right)$ bits and k hash functions are used corresponding to each part. For an element $u_i \in U$, hash functions are calculated as:

$$\kappa_i(u_i) = \ell_1(u_i) + i \times \ell_2(u_i)\ mod\ \vartheta \tag{2.3}$$

Each hash function $\kappa_i(.)$ changes bit in i^{th} array where $(i | 1 < i < k)$. This technique leads to less inter-hash function collision, further usage of only two hash functions to generate all k hash functions decreases the computational overhead.

2.2 Approximate Membership Query

Definition: *Membership Query:* For a given set $S = \{x_1, x_2, ..., x_N\}$ with N items, membership query confirms the presence of queried element x_q in the set using deterministic approaches. The result of membership query is in binary form, *i.e.*, *one* indicates ($x_q \in S$) and 0 indicates that ($x_q \notin S$). Space and computational costs of these approaches are depended on the size of dataset considered.

Definition: *Approximate Membership Query (AMQ):* For a given set $S = \{x_1, x_2, ..., x_N\}$ with N items, AMQ checks the presence of queried element x_q in the set by using some approximation or probabilistic approach for fast results. Query returns results with some approximation; here x_q indicates "*possibly in set*" or "*definitely not in set*". The query complexity is independent of the size of dataset and space complexity is significantly reduced.

2.2.1 Bloom Filters

Storing bulk data in a small space and querying the items in the given space S can be accomplished by the usage of Bloom Filter (BF), a randomized data structure that supports set membership query. BF [50], a space efficient PDS, is used to represent a set ($S \subset U$) of n elements which helps in approximate membership testing. It consists of an array of m bits, denoted by $BF[1, 2, ...m]$, initially all bits set to *zero*. The filter uses k independent hash functions $\left(\stackrel{j \leq k}{_{j=1}} H_j(.)\right)$ with their value $\left(\stackrel{j \leq k}{_{j=1}} h_j(.)\right)$ ranging between (1 *to* m), assuming that these hash functions independently map each element in the universe to a random number uniformly over the range. For each element $x_i \in S$, bits $BF[h_j(x_i)]$ are set to *one*, ($\forall j | 1 < j < k$.)

Given an item ($y_i \in Q$) where Q is set of query elements, its membership is checked by examining whether the bits corresponding to hash functions $\left(\stackrel{j \leq k}{_{j=1}} H_j(y_i)\right)$ are *one* in the

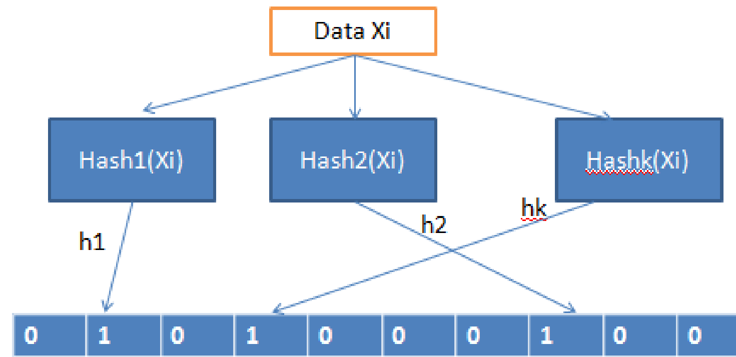

Figure 2.1: Insertion in Bloom filter

$BF[1...m]$ array. If all hash positions, *i.e.*, $\left(\sum_{j=1}^{j\leq k} h_j(y_i)\right)$ are set to *one*, then y is considered to be part of S other wise not.

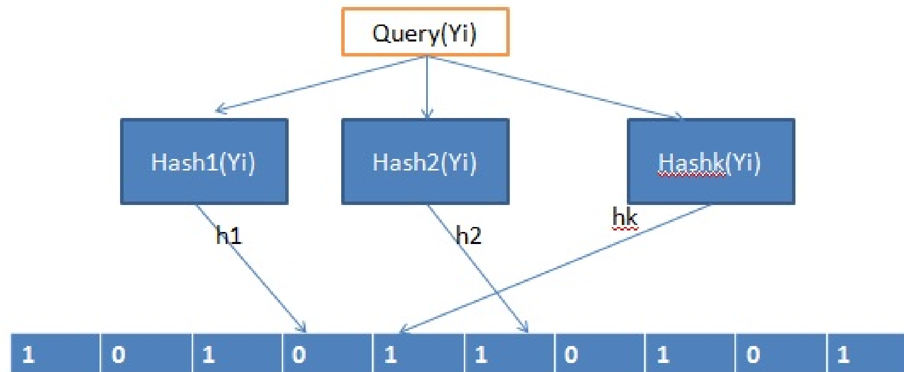

Figure 2.2: Query in Bloom filter

Its space efficient representation comes at the cost of false positives, *i.e.*, elements can be erroneously reported as members of the set although they are not. In practice, the huge space savings often outweigh the false positives if kept at a sufficiently low rate. Given a BF with m bits and k hash functions, insertion and membership query time complexity is always $O(k)$. Their detailed working has been illustrated in Figs. 2.1 and 2.2.

For getting the false positives probability of a BF, it is assumed that hash functions used in BF are universal random functions, *i.e.*, probability for selecting each BF bit is equally likely [82]. When inserting an element into the filter, the probability that a certain bit is not

set to one by a hash function is $\frac{1}{m}$. Since k hash functions are used, probability that none of them sets the specific bit to one is $(1-\frac{1}{m})^k$. After performing n insertions in BF, the probability that a given bit is still zero is:

$$\left(1-\frac{1}{m}\right)^{kn} \tag{2.4}$$

and consequently the probability that the bit is one is:

$$1-\left(1-\frac{1}{m}\right)^{kn} \tag{2.5}$$

In querying process, if all the hash positions corresponding to the hash functions ${}_{j=1}^{j\leq k}H_j(y_i)$ in BF are set to *one*, the BF claims that the element belongs to the set. The probability of false positive, *i.e.*, probability when the element is not part of the set and BF claims its part of set is given by:

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \tag{2.6}$$

Using approximation principle it can be concluded that:

$$\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^{k} \approx (1-e^{-kn/m})^k \tag{2.7}$$

Thus false positive probability, f_p for BF is given by:

$$f_p = (1-e^{-kn/m})^k \tag{2.8}$$

The false positive probability decreases as the size of the BF (m) increases and f_p increases as the number of elements increase. By using more hash functions the probability

of collision decreases. Further, user can predefine false positives according to application's requirement. The accuracy of BF depends on the filter size m, the number of hash functions k, and the number of elements n. To minimize the f_p with respect to k, the optimal value of k_{opt} is given as follows:

$$k_{opt} = \frac{m}{n} ln2 \tag{2.9}$$

The space advantage of BF is dependent on the error rate acceptable for the application considered. To maintain a fixed f_p, with the n number of elements, the size of a BF m is given by:

$$m = -\frac{(n \times ln(p))}{((ln(2))^2)} \tag{2.10}$$

Categories of Bloom Filter

Based upon the applications domains, many variants of BF have been proposed which can be broadly classified into four categories. These are as shown in Fig. 2.3.

- ***Static Bloom Filter (SBF):*** BF with fixed size of array is known as SBF. This type of BF has constant false positive rate and works efficiently on static data sets. Based on the number of elements (n), parameters like size of BF (m) and number of hash functions used (k) can be decided. Moreover, another essential determinant is the potential range of the elements to be inserted; if it is limited, a deterministic bit vector can do better. Distance-sensitive BF [83], weighted BF [84], *etc.*, are some of the BFs which falls under this category. Some of the problems associated with SBF are that collision rate increases exponentially as size of the incoming data increases and deletion operation is not allowed because a particular bit which is set to one earlier will be set to zero after deletion operation but this will unmark another elements too which had marked that bit as one. Static BFs are suitable for representing static datasets where size is known in advance, *i.e.*, it does not varies with time.

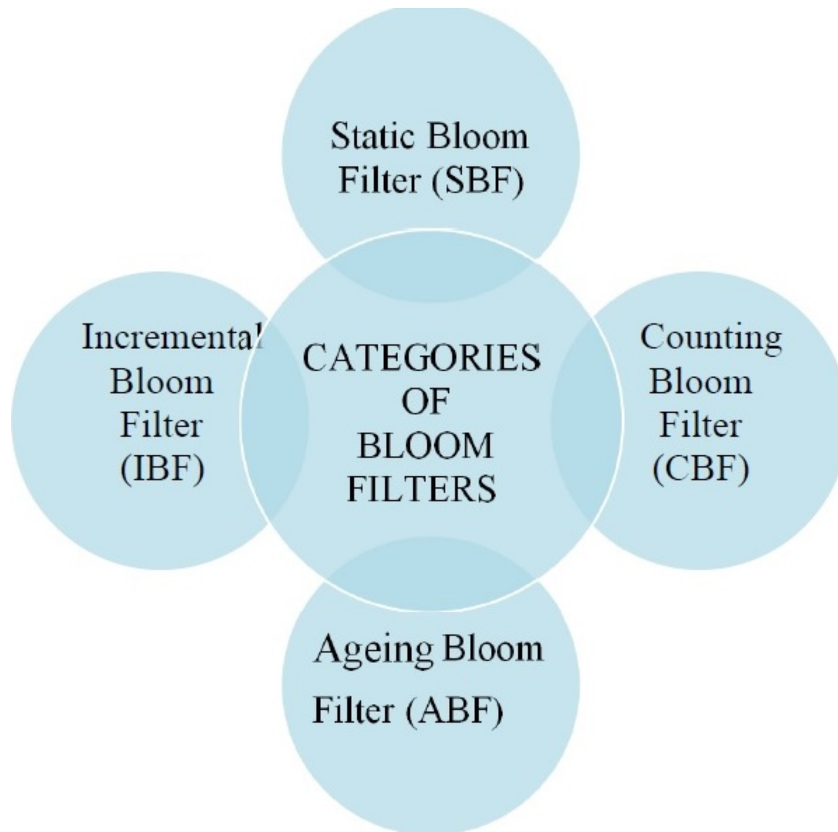

Figure 2.3: Categories of Bloom filter

- ***Counting Bloom Filter (CBF):*** Counting BF, introduced by Fan *et al.* [85], uses a counter of c bits where range of counter is $\{1, 2^c - 1\}$. Based upon the hash indexes computed, *i.e.*, $\left(^k_{i=1} h_i(x) =^k_{i=1} H_i(x)\right)$, insertion and deletion operations are performed on the counters. Whenever an element is added or deleted from the CBF, the corresponding counters are incremented or decremented, respectively. CBF is primarily used to answer frequency queries. Although it can be efficiently used for applications where deletion operation is required, the memory overhead for CBF as compared to standard BF is significantly large and determining value of counter is a difficult process. Number of variants of CBF have been proposed to overcome such issues and maximize the storage of counting BF with minimum memory requirements and less false positive and false negative rates.

Some important variants of CBF are discussed in this section. In Spectral BF, [86] value of smallest counter is increased when a new element is inserted and query process returns minimum count for an element from the counter selected. It is mainly used for storing the multi sets data and supports frequency query. Another improvement in CBF was done by improving hashing technique called d-left counting BF [87], in which CBF with d-left hashing was used to calculate index values of hash functions by dividing BF into sub tables. Use of this efficient hashing in CBF leads to less number of collisions. It is used for element lookups and fingerprint matching applications. Deletable BF [88], another variant of CBF, optimizes the process of deletion by using probabilistic approach for element removal. It keeps the record of regions with high collisions, *i.e.*, the regions where probability of deletion is quite high. The main aim of this variant is to minimize false negatives. It finds application in source routing to avoid loops, middle-box services like load balancer, firewalls, *etc.* Although it can be efficiently used for applications where deletion operation is required, it increases the memory overhead by a larger factor and determining value of counter is quite cumbersome process.

- ***Incremental Bloom Filter (IBF):*** The BFs which are adaptive in nature, *i.e.*, change their size according to the incoming data; fall in IBF category. The basic idea of incremental BFs is to represent a dynamic set D with a dynamic bit matrix and accommodate incoming data by adding new filters at runtime. If rough estimate of the number of elements to be inserted is not available, then a hash table or an incremental BFs is the better option. Dynamic BFs [89], Scalable BFs [90], *etc.*, belong to this category.

  Dynamic and scalable bloom filters deal with the scalability problem by adding bit arrays of varying sizes as the incoming data increases. In Dynamic Bloom Filter (DBF),

an array of size m is added repeatedly, once the fill capacity of the existing DBF exceeds the defined threshold. But this addition in DBF causes the significant increase in error rate.

Scalable Bloom Filter [91] is a *BF* variant that can adapt dynamically to the number of incoming elements, with an assured maximum false positives f_p^0. In addition to the initial array of size m_0, scalable BF includes two additional parameters: expected growth rate (s) and the error probability tightening ratio (ρ) ($0 < \rho < 1$); insert operation in scalable BF for an element x using k hash functions and parameters s and ρ is given by $Insert(ScalableBF[.], x,_{i=1}^{k} h_i(x), f_p^0, s, \rho)$. When $m_0 = \log_2(f_p^0)^{-1}$ exceeds the defined threshold, a new array $\left(m_1 = m_0 + log_2\rho^{-1}\right)$ is added and error probability for new filter $f_p^1 = f_p^0\rho$. Size of additional i^{th} array m_i is:

$$m_i = \log_2(f_p^i)^{-1} = m_0 + i \times log_2\rho^{-1} \tag{2.11}$$

For flexible growth in scalable BF size, exponential growth factor (s) is added, generating i individual filters of size $\left(m_0, m_0 s, m_0 s^2, .., m_0 s^{i-1}\right)$. When the fill ratio t_h for one filter exceeds the defined threshold, another filter is added to it with a well defined growth parameter s. Elements stored in i^{th} filter are approximately:

$$N_i \approx m_0 s^i (ln(t_h)) \tag{2.12}$$

At a given time, error probabilities of all i individual filters $\left(0 < i < (i-1)\right)$ is $\left(f_p^0, f_p^0\rho, f_p^0\rho^2, ..., f_p^0\rho^{i-1}\right)$. The compounded error probability for the SBF is:

$$f_p^{SBF} = 1 - \prod_{x=1}^{i}(1 - f_p^0\rho^{x-1}) \tag{2.13}$$

Query process in scalable BF is accomplished by testing the presence of query element in each filter, starting from active filter to oldest filter. At the time of query, if N be the total incoming elements and N_0 be the elements accommodated in initial filter size, *i.e.*, m_0. Then, total number of arrays added in scalable BF is $\frac{N}{N_0}$. Hence, search complexity for worst case analysis is:

$$O(k(\lfloor \log(N/N_0+1) \rfloor)+1) \tag{2.14}$$

However, major drawback of these types of BFs is that query complexity increases as the size increases. Initial size of filter is important factor in such cases; assigning small initial size array leads to computational overhead, slice addition and query complexity overhead; on the other hand using larger size for initial DBF size may lead to memory wastage.

- ***Ageing Bloom Filter (ABF):*** Some network applications require high-speed processing of packets. For this purpose, BFs array should reside in a fast and small memory. In such cases, due to the limited memory size, stale data in the BF needs to be deleted to make space for new data. The answer to such type of applications is ABFs. These BFs work similar to Least Recently Used (LRU) cache. Stable BF[92], A^2 buffering, double buffering [80], *etc.*, are some of the examples of ABF. To accommodate such type of issues, number of solutions are proposed by domain experts, one of these is by using only one buffer [93], *i.e.*, allocating a buffer for insertion of elements coming from a particular network stream. For each new element, the buffer can be checked, and the element may be identified as distinct if it is not found in the buffer, and duplicate otherwise. When the buffer reaches its fill ratio, whole data is evicted from the buffer, *i.e.*, buffer is reset to it's original value. Search time complexity and false posi-

tive rate is determined for a particular time interval in ABF. Another solution proposed for aging scheme using similar concept is double buffering [94]. In this approach, concept of buffering is used but with two filters. Initially data is filled in first filter and once the threshold exceeded, data is filled in next filter but as soon as the threshold of second BF is crossed, data is evicted from first filter and this process continues. Advantage of this approach is that we can store data for more time by using double memory than simple buffering approach. Example of aging BF include techniques like A^2 buffering where one buffer is divided into two parts and then double buffering is performed. One of the short comings of this approach is that size of filter used is static and rough prediction of size of filter required may affect the accuracy of membership query.

Because of simplicity in design and adaptive nature BF has been successfully used in a large number of application domains. Variants discusses in Table I explains the modifications proposed in BF which make it a successful candidate for applications in different domains. It has been observed that recently the focus has shifted to BF which deals with streaming data, *i.e.*, to those belonging to dynamic or ageing BF category.

Table 2.1: Variants of Bloom Filter (SBF - Static Bloom Filter, CBF - Counting Bloom Filter, IBF- Incremental Bloom Filter and ABF - Ageing Bloom Filter)

S.No.	Authors	Filter Name	Category	Special Features	Areas of applications
1.	Burton H. Bloom [50]	Bloom Filter	SBF	Uses compact and probabilistic way to represent sets	Search in databases and dictionaries
2.	Fan *et al.* [85]	Counting Bloom Filter	CBF	Frequency query for an element can be answered by using counters instead of simple bits	Used in conjunction with web caches, supports frequency query
3.	Michael Mitzenmacher [95]	Compressed Bloom filter	SBF	Compression of BF for transmission purpose	Web cache, P2P networks and distributing routing table over network
4.	Cohen and Matias [86]	Spectral Bloom filter	CBF	Filter increases the smallest counter value corresponding to hash values to obtain the minimum count of an element	For storing multisets, supports frequency query
5.	Kumar *et al.* [96]	Space-coded Bloom Filter	SBF	Represents a multiset and support frequency query	Measure per-flow traffic, anomaly detection, blind streaming
6.	Goh E. J. [97]	Secure Bloom Filter	SBF	Secures indexes by applying pseduo random function twice to each element	Privacy preserving applications
7.	Shanmuga sundaram *et al.* [98]	Hierachical Bloom Filter	SBF	Hierarchical construction of bloom filters, Low false positive rate	Mainly used in sub string matching
8.	Chazelle *et al.* [99]	Bloomeir Filter	SBF	Encode functions in BF and some times number of BF are used in pipeline	Check membership of function values stored in BF
9.	Zhong *et al.* [100]	Split Bloom Filter	IBF	Matrix of $s \times n$ is used, where s is predefined constant based on the estimated cardinality of data set and n is number of bloom filters used. More space is allocated in advance to increase the capacity of BF	Used for membership query operation for dynamically increasing data set.
10.	Chang *et al.* [101]	Filter Banks	SBF	Mapping of elements and sets determines which element of set S matches with element X.	Routing and forwarding in networks
11.	Lu *et al.* [102]	Variable Length Signatures	ABF	Works as Double Buffer. Based on the flow of data, signature length (hash functions) can be changed.	Network Flow management

Variants of Bloom Filter (SBF - Static Bloom Filter, CBF - Counting Bloom Filter, IBF- Incremental Bloom Filter and ABF - Ageing Bloom Filter) (continued)

S.No.	Authors	Filter Name	Category	Special Features	Areas of applications
12.	Bonomi *et al.* [87]	d- left counting Bloom Filter	CBF	Used with counters by dividing BF in sub-tables. Less number of collisions are reported	Element lookups, fingerprint matching
13.	Deng *et al.* [92]	Stable Bloom filter	ABF	Ensures that, with time, expected number of zeros remain same in the BF. Helps to identify whether X is previously seen in streaming set S.	Duplicate detection in streaming data
14.	Donnet *et al.* [103]	Retouched Bloom Filter	SBF	Random Bit clearing process, less false negatives reported	Used efficiently in distributed network topology applications where information about large sets of nodes must be shared among route tracing monitors.
15.	Kirsch and Mitzenmacher [83]	Distance-senstive Bloom Filter	SBF	Identifies closeness of an item in S, implemented using LSH	Used for speed and space requirement improvement in network and database applications
16.	Bruck *et al.* [84]	Weighted Bloom Filter	SBF	More bits are allocated to elements which are queried more frequently	Used in binary classification and Zipf distributed data sets
17.	Bruck *et al.* [104]	Adaptive Bloom Filter	CBF	Uses varying number of hash functions, takes less memory than CBF as it uses Adaptive counters	Suited for applications where upper bound of data is not known apriori and large number of counters are required
18.	Almeida *et al.* [90]	Scalable Bloom Filter	IBF	Changes the size of BF according to input requirement under a tighter upper bound to maintain constant false positive rate	Suitable for dynamic or streaming datasets membership query
19.	Zhong *et al.* [105]	Data Popularity conscious Bloom Filter	SBF	Uses more hash functions for important elements and less hash functions for rarely queried ones, popularity awareness with off line tuning	Show good results for data having skewed distribution
20.	Ahmadi and Wong [106]	Memory-Optimized Bloom Filter	SBF	Uses single hash function through random hashing (use different hash function for each element)	Used in multidimensional packet classifier based on the tuple space search.

Variants of Bloom Filter (SBF - Static Bloom Filter, CBF - Counting Bloom Filter, IBF- Incremental Bloom Filter and ABF - Ageing Bloom Filter) (continued)

S.No.	Authors	Filter Name	Category	Special Features	Areas of applications
21.	Kirsch and Mitzenmacher [80]	Less hash Bloom Filter	SBF	Uses double hashing and partition hashing	Finds usage in applications where computational cost is a critical factor *e.g.* cryptographic process in network, streaming data monitoring, *etc.*
22.	Guo *et al.* [89]	Dynamic Bloom Filter	IBF	Dynamically increases size of BF as the amount of incoming data increases by adding same size filter as the original one	Used in dynamic sets in distributed environment
23.	Goel and Gupta [107]	Layered Bloom Filter	SBF	A layered BF with multiple BF layers. Keeps track of number of time the element is added by querying multiple layers, starting from deepest one.	Supports frequency query for large datasets
24.	Rothenberg *et al.* [88]	Deletable Bloom Filter	CBF	Use probabilistic approach for element removal by keeping record of region with high collisions, supports deletion without false negative	Used in source routing for avoiding loops, used in middlebox services like load balancer, firewalls, *etc.*
25.	Laufer *et al.* [108]	Generalized Bloom Filter	SBF	Two set of hash functions are used: one for setting bits, another for resetting them. Membership query can be performed with comparatively low error rate.	Used to reduce error in streaming applications where eviction of data is mandatory to make more space in the filter.
26.	Dautrich *et al.* [109]	Decaying Bloom Filter	ABF	Works on time window protocol.	Used for duplicate element detection and hint base routing in wireless sensors and data stream.

2.2.2 Role of Bloom Filter in Streaming Data

Stream processing systems were born nearly a decade ago, due to the need for low-latency processing of large volumes of highly dynamic, time-sensitive continuous streams of data from sensor-based monitoring devices.

In some applications like in-stream data coming from sensors, network traffic, *etc.* the data is generated dynamically and size of the data set being generated cannot be determined a priori. When size of incoming data is not known, use of static BF will either lead to high collision rate or result in wastage of storage space. In such cases it is difficult to determine the optimal BF parameters (m,k) in advance, so a target false positives threshold cannot be guaranteed. In order to accommodate the growing data size, one of the major requirements in BF is that filter size m should grow dynamically. For this purpose BFs from the incremental BF category are useful *eg.* DBF and scalable BF can be used to accommodate the dynamically growing data.

Some network applications require high-speed processing of packets and BFs array should reside in a fast and small memory for such type of applications. Applications like content forwarding which can detect strings in streaming data without degrading network throughput [96], XML packet filtering, XML path query as a query string, etc. come under this category [110]. Streamed data can also be accommodated by the use of BF from aging BF category, in which stale data is evicted at regular intervals to make space for new data. Some examples of these BF are Double buffering [93], A^2 buffering [94] and stable BF [92].

2.2.3 Role of Bloom Filter in Duplicate Detection

Duplicate detection is the task of detecting unique entries in unbounded data streams. Duplicate elimination is a crucial intermediate step in data processing and analytics of the incoming streams. The problem of finding approximate duplicate items has been studied in the contexts of data management and web applications. Data streams are generally unbounded and traditional DBMS approaches of accommodating whole stream in the memory leads to

very high memory utilization. We need to provide results for the query in minimum time with less computational cost and minimum memory requirements.

On-line monitoring of data streams and eliminating duplicates is an important issue in stream analytics. For redundant data elimination, primary focus should be on differentiating between duplicate and distinct element in the data stream. Detecting duplicates in streams become more difficult because of unbounded nature of data coming at high rate and necessity of processing data in one pass by using limited amount of memory [111].

There are many solutions for the duplicate detection problem [112] but our main focus is on the solutions based on usage of BFs for efficient detection of duplicate datasets in streaming data. In window based approaches, for every new input arriving from the stream, an old entry is evicted by adjusting the size of window. Recent work done in window based framework for duplicate detection using BF is by Metwally *et al.* [113]. In their work three type of window based approaches are defined: landmark window, sliding window and jumping window. In landmark window approach, in-coming data is processed as disjoint portions of the stream, which are separated by landmarks. Landmarks can be defined either in terms of time, *e.g.* on daily or weekly basis, or in terms of the number of elements observed. In landmark window approach, whole data from BF is deleted upon reaching the new landmark. Since individual element deletion is not performed in this approach, simple BF is used. In sliding window approach size of window is not fixed, it grows as the in-coming data from stream grows. The element deletion is also not allowed in this approach so simple BF is used in implementation. The full size of window is maintained to query old data. Querying process in sliding window has more computational cost as compared to landmark window. Landmark window requires less space as compared to sliding window but there are chances of some missed values in case of landmark window because upon reaching new landmark previous data is deleted. The jumping window is based on idea of dividing the individual element window into smaller sub windows and usage of counting BF to accommodate more number of elements. The latest sub window active for insertion is referred as jumping window or

current window. As the latest sub-window crosses its threshold, jumping window is moved to next sub-window and oldest sub-window is deleted. So these window based approaches either need dynamic size that lead to extra computation and expensive query process or they remove a large chunk of data together which may lead to the higher false negatives [114].

Another solution is use of buffering and caching methods, used in many database, network applications, URL caching and web crawling. Some bloom based buffering techniques include double buffering and A^2 buffering. In double buffering [93] two different buffers of size $\left(\frac{m}{2}\right)$ are maintained. As one filter crosses its threshold, insertion starts in second and when second filter is full, *i.e.*, this filter reaches its threshold then whole data from first one is drained and insertion is again started in first filter. For query process, first active filter, *i.e.*, current filter in which insertion is being done recently is checked and if query return false then previous filter is scanned. In A^2 buffering [94] similar approach is used, in this when active filter reaches its 50% capacity insertion is started in both filters and when active filter is full its data is evicted by resetting it to zero. In buffering approaches element wise deletion is not allowed so simple BF is used. Buffering approaches store data for short time with large redundancy. Handling data streams with buffering leads to wastage of memory; further it is computationally costlier to check the threshold of BF at every step.

Recently variant of BF for duplicate detection in stream named as Stable Bloom Filter has been proposed by Deng and Rafiei [92]. Counting BF with c bits in counter is used as the base of its implementation. Before inserting any element in the stream, it is checked that whether the in-coming element is queried earlier in the stream or not. If query returns *true*, *i.e.*, values of counters corresponding to hash indexes are high then element is marked as duplicate and element is not inserted but if query return *false* indicating that the element is observed for the first time in the stream, element is added. Before insertion begins; stable BF makes space for in-coming elements by randomly decrementing the values of counter by one in each iteration. In insertion process, k hash functions are computed and correspondingly hash value counters are set to Max, where ($Max = 2^c - 1$). The main advantage of this

approach is that it provides query results on streaming data in $O(k)$, independent of the size of in-coming data, using constant memory.

Another technique proposed for duplicate detection is Reservoir Sampling based Bloom Filter (RSBF), a hybrid of reservoir sampling and BF [115]. It uses k BFs each of size s bits. Insertion of an element is performed after query process. One hash function of each BF is reserved. If hash index associated with each BF is found *high* then element is considered as duplicate and insertion is not performed. Else, insertion is performed by setting corresponding k bits *high* in k BFs. To accommodate streaming data in fixed size BFs, random deletion is done by resetting k randomly selected bits to *low*. RSBF uses same amount of memory as SBF, has fast convergence rate and shows significant improvement in false negatives but it does not provide significant improvement in false positives.

Recent variants of CBF include Variable Increment Counting Bloom Filter (VI-CBF) and FingerPrint based Counting Bloom Filter(FP-CBF). In VI-CBF [116] two different set of hash functions are maintained, first set of hash functions, *i.e.*, $\left(H_{i=1}^{k}\right)$decide the indexes on which increment is performed and another set of hash functions, *i.e.*, $\left(G_{i=1}^{k}\right)$ decide the value from a set $D_L = \{L, L+1, ...2^L - 1\}$ by which increment is performed on the selected indexes. Same process is followed in deletion operation. It decreases the false positives by a huge factor as compared to standard CBF. Some limitations of VI-CBF are that implementation is more complex, calculations for two hash functions leads to extra computational cost and more bits are required to avoid overflow of counters.

S.Pontarekki *et al.* proposed FP-CBF [117], where each counter has some extra bits denoted as fingerprint bits. The main idea behind the use of these bits is to provide a second level check in querying operation and reduce false positive and false negatives. Here each index of d bits is divided into counter bits (c) and fingerprint bits ($f = d - c$). In insertion process, indexes for updation are computed (based on the k hash functions used) and corresponding to these indexes all counters are incremented by one. A separate hash function H_{fp} is used to update the fingerprint cell of all selected indexes with *xor* operation. In querying

process, values of counters are checked and if all values are high then second level check is performed by matching H_{fp} value of queried element with fingerprint cells. If value of H_{fp} in all fingerprint cells match, query returns *true* value. FP-CBF provide more accurate results with minimum computational effort.

2.2.4 Applications of Bloom Filter

Initially BF was used to represent words in a dictionary. Gradually, BF was widely used in many networking and security algorithms like authentication, IP trace-backing, string matching, reply protection [82, 118], *etc*. Presently it is used in fields as diverse as accounting, monitoring, load balancing, policy enforcement, routing, clustering, security of network [119]. Cellular networks use device-to-device communication using BF based approach to identify mobile applications [120]. BF is applied in VANET applications and cloud platforms for DDoS attack prevention [41, 121] and privacy preservation [61, 122].

Industrial applications: Technical giants are also using BF and its variants in different fields. Quora uses a shared BF in the feed back-end to filter out stories that people have seen before. In Facebook, type-ahead search, fetching friends and friends of friends to a user typed query is performed using BF. It uses a BF to avoid redirecting users to malicious websites. Oracle uses BFs to perform bloom pruning of partitions for certain queries. Apache HBase uses BF to boost read speed by filtering out unnecessary disk reads of HFile blocks which do not contain a particular row or column [123]. Few concerns related to BF are: it is not possible to retrieve original key after hashing in BF and there is a probability of false positives and false negatives in some variants which support deletion.

2.2.5 Quotient Filter

BF and its variants work efficiently only when entire BF array resides in the main memory. If the size of BF exceeds the available RAM of system then complexity due to number of operations required for fetching from main memory and checking different parts increases the

query time manifold. If such process continues, BF will lose the purpose of its use. Another PDS named Quotient Filter (QF) [51] can be efficiently used for approximate membership query; it supports multi layer design, buffering and hash localization which results in fast and efficient querying of the elements even on secondary memory.

QF is a space efficient and cache friendly data structure which use quotienting technique of hashing [124] to store a set S efficiently. It supports insertion, deletion, querying and merging operations. The detailed working of the same is shown in Fig. 2.4.

Each element $x \in S$ is mapped to $h(x)$, a primary hash function, for converting x into set of p bits called fingerprint of x, *i.e.*, $\left(h(x) \mapsto \{0,...,2^p-1\} \Rightarrow fp(x)\right)$. $fp(x)$ is an open hash table of size ($m = 2^q$) buckets having $(r+3)$ bits per bucket. It is used for storage where r denotes the least significant bits in $fp(x)$ and ($q = (p-r)$) most significant bits in quotienting technique represent three extra metadata bits used with each element. To insert fingerprint of an element $fp(x)$ in QF, remainder $f_r \leftarrow (fp(x) \mod 2^r)$ and quotient $f_q \leftarrow (\lfloor fp(x)/2^r \rfloor)$ are computed, where f_q denotes the index of bucket to be used for insertion and f_r denotes the value to be inserted in bucket f_q. The main advantage of QF over BF is that we can reconstruct the $fp(x)$ from the f_q and f_r, where $fp(x)$ is given by:

$$fp(x) = f_q.2^r + f_r \tag{2.15}$$

In a QF, for two given fingerprints $fp(x)$ and $fp(y)$, it is stated that if $f_q(x) < f_q(y)$ then $f_r(x)$ is always stored before $f_r(y)$.

The quotienting technique tries to generate a unique remainder and quotient for every $x_i \in S$ although there are some chances of collision. On the basis of remainder and quotient of $fp(x)$, collisions are divided into two types:

$$\text{Collisions in QF} = \begin{cases} \text{Soft, if} & f_q(x) = f_q(y) \\ Hard, \text{if} & \{(f_q(x) = f_q(y)) \ \&\& \ (f_r(x) = f_r(y))\} \end{cases} \tag{2.16}$$

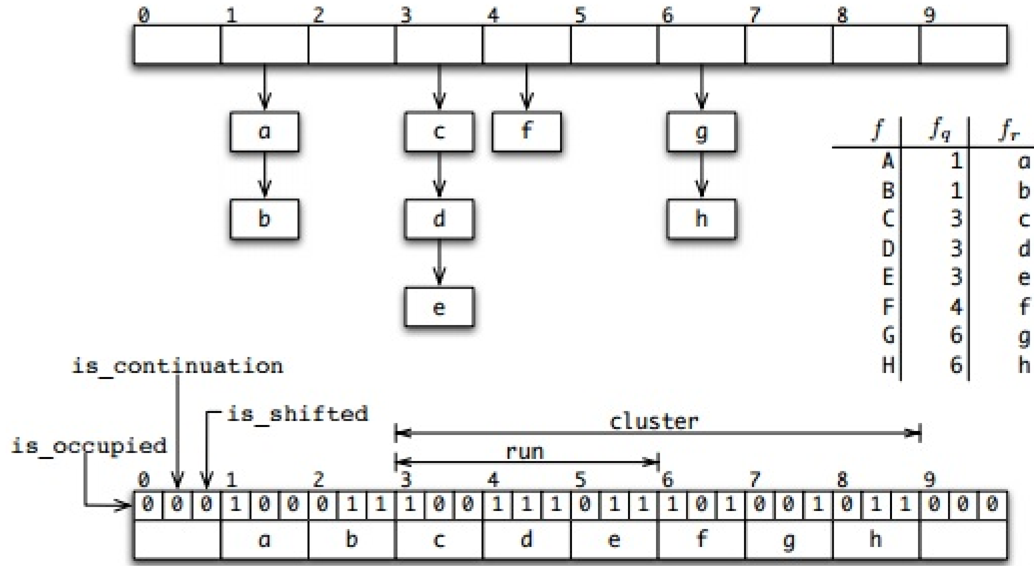

Figure 2.4: Quotient filter work flow

In case of soft collision (when f_q of two items collide, but they have distinct f_r), linear probing is used as a collision resolution strategy, where remainders of different fingerprints having same f_q are stored contiguously; called *run* in QF. If necessary, remainders associated with different f_q are shifted and corresponding metadata bits are updated for each bucket. In QF, *cluster* refers to the sequence of one or more consecutive runs without having any empty bucket in between them. A cluster is immediately followed by a empty slot. Canonical slot for a fingerprint x ($fp(x)$) is the original bucket for the insertion of $f_r(x)$ indicated by $f_q(x)$. The terms *run* and *cluster* are used to identify suitable position to insert or query the element after various shifts have been performed with the use of metadata bits. The false positive in QF are encountered due to hard collisions(when f_q and f_r of two items collide). Assuming that $h(x)$ is distributed uniformly, the probability of hard collision(Pr_{HC}) is given by:

$$Pr_{HC} = 1 - (1 - \frac{1}{2^p})^n \approx (1 - e^{-n/2^p}) \tag{2.17}$$

Metadata bits are used to find optimal location of elements which have been shifted from canonical slot because of soft collision; f_r of element belongs to run of a slot f_q which is

Table 2.2: Bits significance in QF

is_occupied bit	is_continuation bit	is_shifted bit	Significance
0	0	0	Empty Bucket
0	0	1	Bucket is holding start of *run* that has been shifted from its quotient (f_q) bucket (canonical slot).
0	1	0	ϕ (Not used)
0	1	1	Bucket is holding continuation of *run* that has been shifted from its quotient (f_q) bucket (canonical slot).
1	0	0	Bucket is holding start of *run* that is in same bucket.
1	0	1	Bucket B_i is holding start of *run* that has been shifted from its quotient (f_q) bucket (canonical slot). B_i is also occupied with some f_r but its remainder is shifted right.
1	1	0	ϕ (Not used)
1	1	1	Bucket B_i is holding continuation of *run* that has been shifted from its quotient (f_q) bucket (canonical slot). B_i is also a slot in same run but its remainder is shifted right.

stored at different location. Most significant bit, referred as `is_occupied bit` is set *high* for i^{th} bucket if for any $fp(x) \in S$ quotient satisfy $f_q = i$ condition, *i.e.*, i^{th} bucket is canonical slot for some element in dataset. Middle bit known as `is_continuation bit` helps the decoder in searching process to identify group of items belonging to same bucket. Least significant bit named as `is_shifted bit` is used to identify where the f_r (associated with i^{th} bucket) is stored. The significance of these bits is provided in Table 2.2.

f_q denotes the index of bucket in which element needs to be inserted or queried. In insertion operation, suitable position, (sp), to insert the remainder is at the end of *run* of bucket denoted by f_q. For this, all elements after sp are shifted to right, same operations are repeated till the end of the cluster and then element is inserted and metadata bits are updated. For query operation in QF (where queried element is x_q) $fp(x_q)$ is calculated and then corresponding quotient $f_q(x_q)$ and remainder $f_r(x_q)$ are computed. Start of cluster containing f_q is identified and then the start of *run* corresponding to f_q is identified. In querying process, instead of shifting elements only remainder of queried element is checked in concerned run. Deletion process is reverse of the insertion operation.

Biggest advantage of QF is that, in QF original data, although hashed while storing, can be retrieved back through quotienting hashing technique.

The time required in insertion and deletion process can dominate the advantage of using QF since single cluster is scanned. In each operations Chernoff bound can be used to limit

the size of cluster.

Definition: For a QF of m slots, if number of items stored is $\alpha \times m$, then

$$Pr[\exists(\text{A cluster of length}) \geq k] < m^{-\varepsilon} \tag{2.18}$$

here ε is allowable error and $\alpha \in [0,1)$ is a random variable. k, the limit of cluster length (derived from number of slots in QF) is given by [51] :

$$k = (1+\varepsilon)\frac{ln(m)}{(\alpha-1)ln(\alpha-1)} \tag{2.19}$$

The length of largest cluster can be controlled by setting value of m high and $(0 \leq \alpha < 1)$.

Advantages of QF over BF

- In QF, all operations are cache friendly, only single cluster is modified in one operation. Since cluster size can be fixed (Eq. (2.19)) so cluster fits into cache lines easily. Less data fetch-up operation is required for large datasets stored in secondary memory. In BF, secondary memory fetching time for concerned bit for all hash functions increases the complexity of task.

- Since QF supports in-order or linear scan, so results are obtained quickly as compared to BF constructed by adding new slices to existing one, thus search complexity of BF is comparatively high.

- Resizing of QF is possible without rearranging all the hashed data which is not possible in BF.

- Merging of two QF into a larger one can be done easily and false positives do not increase in this operation whereas merging in BF may amplify the error.

- QF performs deletion operation accurately whereas standard BFs does not allow deletion and variants which support deletion may include false negatives.

Disadvantages of QF in comparison to BF

- Storage size of QF is quite large as compared to BF.
- Time required to retrieve the original value from the hashed value is quite large if the data lies in a cluster.

Variants of QF like cascade filter, buffered quotient filter, *etc*. work on similar principle and support working with SSD memory [51].

QF has major advantages over BF in terms of memory and computational time. Use of QF is beneficial when fast insertion and querying is required from the data stored in secondary memory. Further, merging two QFs without any change in accuracy is an added advantage.

2.2.6 Applications of Quotient Filter

QF is widely used in network application. Deep packet inspection is a platform to monitor the incoming and outgoing traffic on a data centre and identifying the malicious user from the packets is a time consuming task. Moreover, this matching process consume a lot of memory and CPU resources. Al-Hisnawi *et al.* used the QF to store the malicious users to make searching task fast and efficient as the size of the incoming data increases [125]. QF has been successfully implemented in automatic terms extraction for domain-specific corpora for fast results. It has also been used for warehouse management to locate the items efficiently [126].

Dutta *et al.* proposed Streaming Quotient Filter (SQF), a quotient filter based streaming model to count the duplicate entries in the streaming data with predefined fixed memory and fast search facility [127]. In proposed model, Dutta *et al.* use Quotient filter for approximate

membership query instead of BF. SQF maintains a hash table and bucket associated with each element in hash table. Hashing is done at two levels: first element is mapped to hash table and then corresponding to hash table entry, hashing is done in bucket by using quotienting technique. SQF performs efficiently in detecting false positives and false negatives but the pre-processing time for converting data into binary form for quotienting and hashing at two levels increases the computational time many folds. Another drawback associated with SQF is that because of clustering, operations like insertion and searching are difficult to perform in parallel.

2.3 Similarity Search

Finding similar items in a set is a process of checking all items and identifying the closest one. To categorize the data set into particular class, one needs to find how much two items of data set are similar to each other. Problems related to finding of similar items are often solved by identifying nearest neighbors of object. Such type of problems have number of mathematical solutions in terms of distance measures like hamming distance, cosine and sine similarity measure, Jaccard's similarity coefficient, Pearson's similarity coefficient, *etc.* [128]. Searching a huge database for similar items using linear search or brute force approach increases the computational complexity exponentially. Such solutions are efficient for small data sets but when massive data sets are considered, all such solutions face two major problems: first is how to store and represent items for finding similarity in massive data sets. Second is how to pairwise compare billions of items especially in such high dimensional data sets [129].

Solution for above stated problem is to either reduce dimensionality of the data set or make structural assumptions about the data for maintaining integrity of data as done in data structures like trees and hashes. Trees show good results for low dimensional data, but as the dimensions of data increases, query complexity and tree construction cost becomes too much. In hashes, data is mapped on hash table using random hash functions. Items mapped

close in hash table are assumed to be close neighbor, but type of hash functions used and hashing collisions can significantly affect the results. Finding nearest neighbor in big data with n points and d dimensions using linear search has $O(nd)$ complexity at its best.

One of the known solution proposed by researchers for finding nearest neighbour in big data with n points and d dimensions is k-d tree. K-d trees or k-dimensional tree, proposed by Jon Bentley in 1975, is a binary tree where every node is a k-dimensional point and each level of the tree represents one dimension [130]. Each level of a k-d tree splits all children along a specific dimension, using a hyperplane that is perpendicular to the corresponding axis. For insertion or query operation k-d tree needs recursive scanning of the tree which is quite time consuming task. In the worst case, search is close to scanning whole tree. Balancing operation in k-d tree requires extra computational effort to generate sorted k-d tree in multiple dimensions. With an increase in dimensions, new levels need to be added, increasing the size of kd-tree exponentially.

Nearest neghbiour problem with approximation rules for d dimensional dataset is defined as following:

Definition: *c-approximate NN problem* [131]: Let $\Re^d$ is dataset with n points, d dimensions and $Q \subset \Re^d$ is set of Query elements, for any query ($q \in Q$) find a point $p \in \Re^d$ such that $\forall p^{'} \in \Re^d, d(p,q) \leq c.d(p^{'},q)$, where c= $(1+\varepsilon)$ is approximation factor and ε is allowable error rate.

Definition: *(R,c)-NN problem* [132]: Let $\Re^d$ is data set having n points and d dimensions with a constant $R > 0$ and $Q \subset \Re^d$ is set of query elements, for any query $q \in Q$ following decision related to closeness can be made:

- if $\exists p \in \Re^d$ such that $d(p,q) \leq R$ then return YES and a point $p^{'} \in \Re^d \leq c.R$.
- if $d(p,q) > c.R$, $\forall p \in \Re^d$ then return NO

2.3.1 Min Hash

In the era of Big data, problems like similarity search on large datasets needs reliable, fast and computationally efficient solutions. One of the solutions to the above mentioned similarity search problem is given by a hashing based sampling algorithm called Min Hash (MH). MH, a probabilistic technique given by Andrei Broder in 1997 [133], is used to find similarity between two items by computing Jaccard similarity $J(A,B)$ between the items being considered for finding similarity.

To find the similarity between members of a set $S = s_1, s_2, ...s_n$ MH uses set of k hash functions $H_k(S) \mapsto Z$. h_{min} which stores the minimum value from the set of hash function $h_{min} \leftarrow min(H_k(.))$. Two elements s_1 and s_2 of set S are considered similar if $h_{min}(s_1) = h_{min}(s_2)$ [133].

$$Pr[h_{min}(s_1) = h_{min}(s_2)] \Rightarrow \frac{|s_1 \cap s_2|}{|s_1 \cup s_2|} \approx J(s_1, s_2) \tag{2.20}$$

If mH is a random variable, similarity of two items s_1 and s_2 is given by:

$$mH = \begin{cases} 1, & \text{if } h(s_1) = h(s_2) \\ 0, & \text{otherwise} \end{cases} \tag{2.21}$$

Here $mH \in (0,1)$ is unbiased estimator of similarity. High variance in mH is reduced by averaging the number of observations.

For finding similar items in massive datasets, MH represents bulk data into compressed form called signature matrix and Locality Sensitive Hashing (LSH) is used to shortlist and narrow down pairwise comparison by identifying the pairs of possible similar items in the dataset.

Steps in Min-hash:

Shingling

Pre processing of data is required to adjust the data into the compressed form for space

saving and uniformity. Shingle is a contiguous sub-sequence of tokens of length k. (k can vary according to application) k should be large enough that probability of any given shingle appearing in any random document is low. Value of k is decided by, $c^k >> l$. Where c is number of available characters and l denotes the average length of document.

Characteristic Matrix (CM)

Shingles computed for each document are hashed to compute jaccard similarity and the matrix set generated from them is called *characteristic matrix*.

Signature Matrix (SM)

$(s \times n)$ signature matrix (SM) is derived from $(m \times n)$ characteristic matrix (CM) having similarity same as that of entire set, where $(m >> s)$. To generate SM, a hash function $\phi(.)$ is used which picks a row randomly from SM and then rows are permuted across the columns to generate more random results. Repeating this process m times, a $(m \times n)$ signature matrix is generated.

2.3.2 Locality-Sensitive Hashing

The basic principle used in Locality-Sensitive Hashing (LSH) is projection of higher dimensional data in low dimensions subspace, using the fact that points close in many dimensions remain close in two dimensions too.

Let $x_i \in \Re^d$—$1 < i < n$, be a set of n points with d dimensions and H be a family of hash functions, mapping $\Re^d \rightarrow U_s$ data set to some universe. For any two points $x_i, x_j \in \Re^d$ select hash function $s.t.$, $(\exists h \in H)$ and analyze the probability of $h(x_i) = h(x_j)$. Let D be the distance measure for pair of $x_i, x_j \in \Re^d$, d_1 and d_2 be two distance ranges between any pair of $x_i, x_j \in \Re^d$ then P_1 and P_2 are the probabilities that x_i and x_j will reside in same bucket. The family of H is called locality sensitive or $(d_1, d_2, P_1, P_2)-$ sensitive [134].

Definition: A family H of hash functions is said to be (d_1, d_2, P_1, P_2)-sensitive [131] if:

- $D||x_i, x_j|| \leq d_1$ then $Pr_H[h(x_i) = h(x_j)] \geq P_1$

- $D||x_i, x_j|| \geq d_2$ then $Pr_H[h(x_i) = h(x_j)] \leq P_2$

for all cases $d_1 < d_2$ and all queries satisfy $P_1 > P_2$, here $D||x_i, x_j||$ denotes the distance between two points. If x_i and x_j are close in $\Re^d$ space, *i.e.*, $D||x_i, x_j||$ is less then there is high probability that P_1 will reside in the same group. The same has been illustrated in Fig. 2.5.

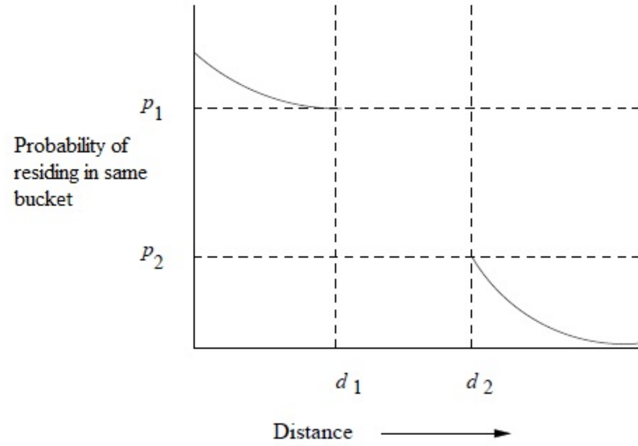

Figure 2.5: Probability v/s distance measure in locality sensitive hashing

LSH is used to solve (R,c)-Nearest Neighbor(NN) problem. (R,c)-NN problem is decision version of c-approximate NN problem. For (R,c)-NN problem in Locality Hash function $r_1 = R, r_2 = cR$, where $c > 0$.

Definition: *Distance Measures* [135]: $\Re^d$ is data set with n points and d dimensions. Distance measure function on $\Re^d$ space between two points $x_i, x_j \in \Re^d$ is $D|(x_i, x_j)|$ and produces a real number, and satisfies the following axioms:

- $D|x, y| \geq 0$, No Negative Values.
- $D|x, y| = 0$ if and only if $(x = y)$
- $D|x, y| = D|y, x|$, *i.e.*, distance is symmetric.
- $D|x, y| \leq D|x, z| + D|z, y|$, triangle inequality or length of shortest path rule.

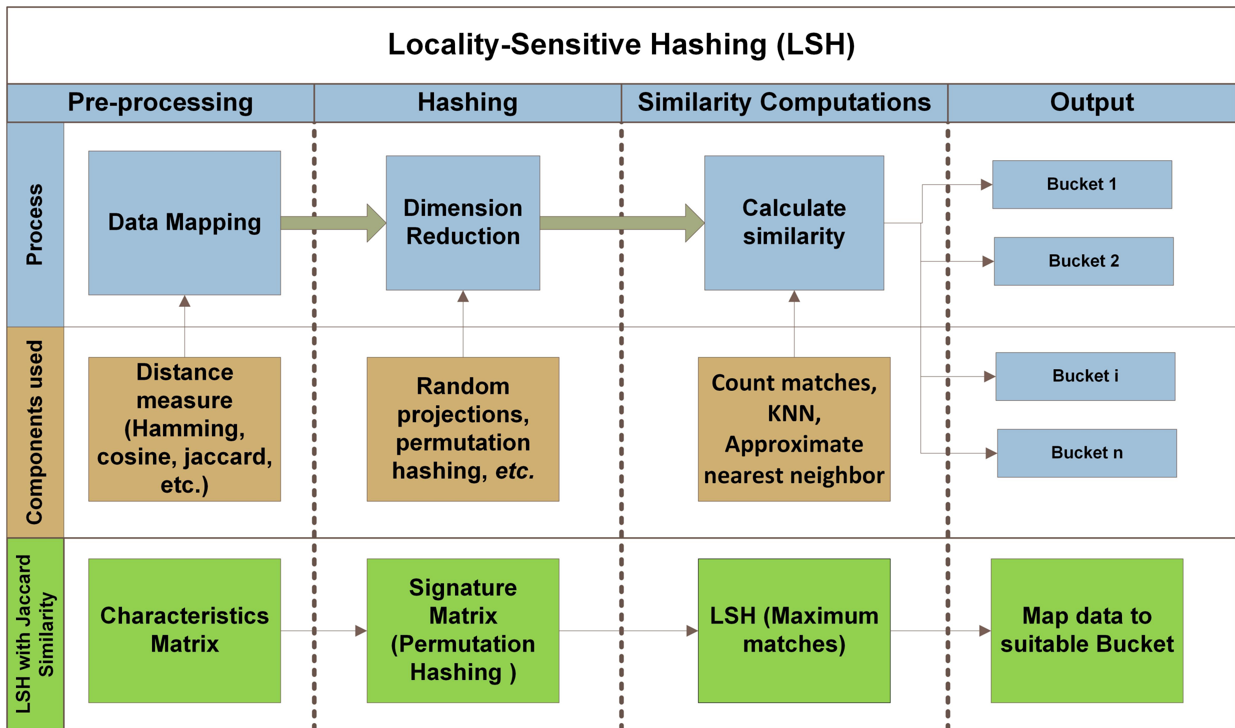

Figure 2.6: Locality-Sensitive Hashing framework

The key idea of the LSH approximate nearest neighbor (NN) algorithm is to construct a set of hash functions such that the probability of nearby points being close after transformation with the hash function is larger than the probability of two distant points being close after the same transformation. The range space of the function is discretized into buckets and we say that there is a 'collision' when two points end up in the same bucket [136].

LSH works by using a carefully selected hash function that causes objects or documents which are similar to have a high probability of colliding in a hash bucket. LSH consists of three phases: pre processing where data is mapped using different distance measures, hash generation where the hash tables are constructed, and similarity search, where the hash tables are used to identify similar items. Entire data is placed in n buckets such that similar items are placed in same bucket. The detailed operations of LSH are illustrated in Fig. 2.6.

Number of variants of LSH have been proposed depending upon the universe on which original data is mapped, *i.e.*, on the basis of distance coefficients which satisfies From the definition of distance measure, not every distance measure have a corresponding LSH fam-

ily. Depending on the random function chosen and its locality sensitive properties, LSH is divided into various categories which are discussed in following sections.

LSH with Hamming distance

LSH on binary string was proposed by Indyk and Motwani [131], where a data set $\Re^d$ is mapped to binary vector from $\{0,1\}^d$ and distance between two objects ($p,q \in \Re^d$)is calculated by hamming distance represented by $||p-q||_h$. H is family of random hash functions. First step is mapping of data to hamming space, *i.e.*, $\left(p \in \Re^d \rightarrow \{0,1\}^d\right)$ and for dimension reduction either each feature of an element is represented by using binary logic to concatenate all feature objects $p_i \in \Re^d$ represented as binary string or $Unary(p)$ function is used for replacing each coordinate of p_i with equivalent binary string of γ bits. Hash functions for hamming space are constructed by selecting k bits randomly from the binary string $b(.)$ of an element. ℓ hash functions are calculated, given by:

$$\forall_{i=1}^{\ell} H_i \leftarrow Random_k(b(.)) \tag{2.22}$$

Instead of comparing each binary string with other, only hash values are compared and element having most similar hash values are considered as nearest neighbor.

LSH with Jacard Similarity

LSH with jaccard similarity [129] can be computed with the help of Min-hash. The SM act as a input. If m_1 and m_2 are the MH based distance between two rows of signature matrix then probability that both rows will appear in same band after applying banding technique will be $(1-m_1)$ and $(1-m_2)$ corresponding. After banding techniques similar items are grouped in same buckets with very high probability. According to definition of LSH family, this is a $(m_1,m_2,(1-m_1),(1-m_2))-$ sensitive LSH family.

LSH with Euclidean Distance

LSH on points distributed in d-dimensional $\Re^d$ space was given by Datar *at el.* [137]. Let $a_i \in \Re^d$, where $a_i \rightarrow \{x_1, x_2 ... x_d\}$ is represented in co-ordinate system then Ed_{a_i,a_j} denotes the Euclidean distance between two points.

For applying LSH on given points in l_s space, first hash function (h_i) is generated by a random line (L_i) in a given plane. L_i is divided into buckets of equal size (s) and orthogonal projection from the given points to the line is drawn. If points are close enough they lie in same bucket on the line L_i. Let $a_i, a_j \in \Re^d$ be two points in plane having an angle θ between them, for two points to be in same bucket necessary condition is $|a_i.a_j| cos\theta \leq s$.

LSH with Cosine Distance

LSH on vectors in d-dimensional $\Re^d$ space is solved by using cosine similarity as a distance measure [138]. Let $v_i \in \Re^d$ is set of vectors, where $v_i \rightarrow \{v_1, v_2 ... v_d\}$ is represented in hyperplanes and θ_{v_i,v_j} denotes the cosine between two points

Let $v_i, v_j \in \Re^d$ be two vectors in a hyperplane having same origin . In case of cosine similarity, hash function (h_g) is generated by a randomly chosen hyperplane (RP_g) passing through the origin with a normal vector rv_g. RV_g is a hyperplane which is selected randomly so that angle between rv_g and v_i, v_j varies, so dot product $DP_i = rv_g.v_i$ and $DP_j rv_g.v_j$ varies. If hyperplane is random such that rv_g lies between v_i and v_j then DP_i and DP_j have same signs other wise they have different signs.

Here only four LSH families, which are normally found in the literature, have been discussed but they can always be further extended. Locality sensitive hashing helps to solve the approximate or exact Near Neighbor Search in high dimensional spaces in sub-linear search time. Initially, all the data is mapped to low dimensional space and then hash based similarity measures are used to find closest cluster for queried item. There are few issues in LSH which need improvement in LSH. Preprocessing in LSH amplify the error rate leading to

increase in the computational overhead many folds. Dynamic changes in the data sets are difficult to incorporate in LSH as it leads to computational overhead of redoing all preprocessing work. Since hashing used in LSH is independent of the nature of data hashing bias may be observed in some cases.

2.3.3 Applications of LSH

Some of the applications which require identification of similar items include similarity in ranking of a product by two users in recommender systems, finding near duplicates corresponding to a particular query document in web documents, identifying similar type of truncations in databases, *etc*. In recent years, LSH has been used for many applications which require fast computational process [139, 140]; in pattern matching which include video identification [141]. Latest area of LSH applications use modified hashing techniques for faster computational process. In mobile services, LSH is used for detecting clones in Android applications [142]. Bertine *et al.* [143] have used LSH for assembling large genomes with single-molecule sequence in bio-informatics.

Table 2.3 summaries the important features of all the PDS covered in this paper.

2.3.4 Spam Detection in Social IoT

Social media has become an important part and effective way to express one's social bondage. Millions of people use social platforms like Facebook, Twitter, Linkden *etc.* to express their emotions and share their experiences. It was initially designed to connect people, but with the advancement in data science, it has been efficiently applied to many real time applications like election campaigns to influencing voters, advertising and market for recommendations on different aspects like social anlaysis, content analysis, sentiment analysis, *etc* [144]. Since it has become a rich data source, processing and analysing such data is considered as one of the most important research issue. But with such a wide platform

Table 2.3: Comparative analysis of all PDS studied

S.No.	PDS → / Parameters↓	Bloom Filter	Quotient Filter	Locality Sensitive Hashing
1.	Hashing	$\surd$	$\surd$	$\surd$
2.	Deletion	$\star$	$\surd$	$\times$
3.	Querying	$\surd$	$\surd$	$\times$
4.	Merging	$\star$	$\surd$	$\times$
5.	Retrieval of original data set	$\times$	$\surd$	$\times$
6.	False Positives	$\surd$	$\surd$	$\surd$
7.	False Negatives	$\star$	$\times$	$\surd$
8.	Element Count	$\star$	$\times$	$\times$
9.	Similarity search	$\times$	$\times$	$\surd$
10.	Cardinality Estimate	$\star$	$\times$	$\times$
11.	Time Complexity	*LOW*	*LOW*	*HIGH*
12.	Space Complexity	*LOW*	*MEDIUM*	*MEDIUM*
13.	Computational Cost	*LOW*	*MEDIUM*	*HIGH*

$\star$ indicates that some variants of PDS may support the particular feature

of users and their data, social networks attracts the spammers or malicious users too, who use this media to promote their malicious activities or try to mislead followers and affect sentiments of particular social groups [145].

Spam is an unsolicited or undesired message [146], whose major intention is to mislead users, infect systems or promote products and services. Spam is a common problem in online media existing in the form of emails [147], websites [148], videos [149], microblogs [150], comments [151], reviews [152], *etc.* Spam emails contain unwanted and malicious information such as malwares, fraud schemes, fake advertisements, *etc.*

Detecting spam accounts is an important issue which definitely needs lot of attention as nature of spammers and amount of damage they can cause is a point of major concern. Detecting spam in social media, especially when massive data flows continuously requires fast processing and storage framework for handling huge amount of data and attributes associated with it. Different approaches have been proposed to detect spam in social networks, but no single model is efficient enough to detect all type of spam. Further, lack of adaptiveness

with changing environment is observed in majority of proposed solutions.

Spammer apply various approaches to hack the users accounts. Sometimes spammers try to duplicate the known and famous person's account by unethical hacking techniques or by creating an account in the name which resembles the Ham account to mislead the followers of that user. Spammers exploit Twitter user's courtesy behaviour to identify their followers in return [153] and thus follow random users to obtain more followers [154]. Another approach used by spammers is to exploit short URLs to camouflage their spam URLs. Instead of sharing any text, a URL of unwanted website or service is shared from different spam accounts [155]. Characteristics of such short URLs on Twitter are analysed using click traffic data [156], and it has been identified that links shared by legitimate users and spammers are significantly different [157]. Many a times trending topics are targeted by the spammer to spread their messages beyond their followers through hashtags [158].

In recent years, social bots, programs that automatically produce content and interact with people on social media, have been exploited for spamming on Twitter. To elude spam detection systems, social bots post tweets about popular and focused topics and try to track the users who follow the bots [159]. Using such strategies, bots can gain high influence on Twitter and pollute time line of users.

Generally spam detection is done by using machine learning techniques which separate spam and non-spam users and tweets. Many approaches have been considered for identifying the spam accounts. Lee *et al.* analyzed the features like demographics, followers targeted, tweet content and user behavior to identify the spammers [160]. Hu *et al.* proposed a solution to handle this problem dynamically, *i.e.*, topological and network information of already detected spammer are updated in database to make spam detection task more efficient [161]. Geo *et al.* proposed a similarity based technique to detect social spam based on the existing clusters [162].

The spam bots which try to replicate legitimate users can be detected by social honeypot trap [154] approaches which include using features derived from temporal behavior, tweet

content, user profile, *etc.* [163]. It has been realized that identifying and removing spam accounts does not solve the problem as the spammers try to create new accounts which look like a Ham user's account or use more social bots to perform the task. More efficient and fast spam detection techniques are required while dealing with dynamic data sets. Semantic and context analysis of tweet can also helps to provide more refine decisions. Santos *et al.* have used standard classifiers like KNN, random forest and logistic regression *etc.* for spam detection [164]. Approach adopted by Romo and Araujo is to detect tweet by comparing title of trending topics with the URL shared in the tweet [165]. Castillo *et al.* used user's tweeting response as a technique to analyze the credibility of tweet on the trending topic [166]. Wu *et al.* considered a semi supervised approach to identify voice over IP call and another semi-supervised technique to detect a malware [167].

In machine learning, ensemble modeling is the process of running more than two different models to solve a particular problem and then provide predictive analysis with more accuracy by combining the result of all models in data mining applications. In Twitter, ensemble is used in analysis based task like sentiment analysis, advertising, *etc.* Kanakaraj and Guddeti provided ensemble-based model to increase the accuracy of classification by including natural language processing techniques especially to perform sentiment analysis [168]. Hassan *et al.* used bootstrapping ensemble to quell class imbalance, sparsity, and representational richness issues in sentiment analysis [169]. The bootstrapping ensemble framework proposed by Giorgis *et al.* builds sentiment time series that can efficiently reflect events and the results achieved have important implications for social media analytics and social intelligence. In another ensemble based work, weighted ensemble is used for sentiment analysis in Twitter; specifically for the message polarity classification subtask [170]. Tsakalidis *et al.* [171] used ensemble based framework to perform polarity analysis of social media content, to analyze the importance of various applications in particular network. Wang and Zechen evaluated airline services using Twitter data by applying an ensemble based sentiment classification approach [172].

2.4 Problem Formulation

After doing the exhaustive survey on the Big data and PDS, it was realized that challenges in Big data analysis include data inconsistence and incompleteness, scalability, timeliness and data security. As the prior step to data analysis, data must be well-constructed. However, considering variety of data sets in Big data, it is a big challenge for researchers to purpose efficient representation, access and analysis of unstructured or semi-structured data. Sincere efforts have been put up by researches to extract intelligence out of huge amount of knowledge base. Major bottleneck in this effort is that there exists no standard method to efficiently map and store the Big data on a compatible data structures. Further the memory required to store such huge bulk of enormous data and the computational time associated along with it for retrieval are increasing the difficulty in designing a standard framework for mapping huge data sets. Although efficient data structures exist to represent a graph having few or large number of nodes and vertices; but when analyzing bulk of data, the off-the-shelf techniques and technologies used to store and analysis of data cannot work efficiently and satisfactorily. Thus, it has been realized that PDS are the promising field of research for the future Big data analytics and streaming data applications. To store dynamic data and for optimal retrieval and searching operations, data structures like PDS are the one of the best models to use. The intent of this thesis work is to propose new data structures and heuristics for handling bulk data which will optimize the storage and retrieval of massive data sets.

2.5 Objectives

Main objectives of this work are:

- To study, analyze and explore various methods available for efficient storage and retrieval of massive data sets.
- To propose a novel technique for efficient storage and retrieval of Big data.

- To test and validate the proposed technique on a simulated environment using various evaluation parameters.

Chapter 3

Streamed Data Analysis Using Adaptable Bloom Filter

This chapter discusses PDS based techniques for efficient storage and retrieval of streaming data. Proposed technique is a variant of scalable BF named as AdapTable Bloom Filter (ATBF) helps to perform analysis on streaming data and make BF size adaptive according to the incoming flow of data. Computational overhead in computing hash functions is reduced by use of hybrid hashing and with the use of Kalman filter ATBF can adapt size according to prediction of incoming data, slice addition overhead is also reduced by great extent.

3.1 Introduction

Fast matching of arbitrary identifiers to the values of incoming data and real time response are the basic requirements for majority of streaming data applications. Thus, some adaptive storage mechanism is required which performs predictive analysis to determine size of data structures being used. Provisions should also be available for adjustments on the basis of previous incoming data or on the basis of real time data flow.

3.1.1 Estimation Models

An estimator is a rule for calculating an estimate of a given quantity based on observed data. There are point and interval estimators. The point estimators yield single-valued results, although this includes the possibility of single vector-valued results. This is in contrast to an interval estimator, where the result would be a range of possible values.

Their are number of estimators like Wiener filter, Kalman Filter, Extended Kalman filter, Generalized filter, Particle filter *etc*. Wiener filter deals with static data only; Kalman filter , a generalization of Wiener Filter [173] allows dynamic data with noisy parameters as input. Predictor model based on polynomial regression [174] uses combination of number of linear regression models which increases the computational complexity of calculations for each prediction manifolds.

The Kalman filter's dynamics model, simplicity of it in implementation and less memory requirement makes it a wonderful candidate for predicting the size of BF in streaming data.

Kalman Filter

Kalman Filter (KF) is a linear system model derived from stochastic process, making it ideal for systems which are continuously changing. In KF, recursive approach is used where a common model is formulated and all future calculations are performed on the same equations without any modification. It is easy to implement and requires less memory since it does not keeps record of old data except the previous state. Further, less computational cost makes it suitable for real time problems.

KF is a powerfull mathematical tool mainly used for stochastic estimation from noisy sensor data or data streams occurring at regular intervals. The basic assumption of KF is that system should be continuous and can be modeled as a normally distributed random process X, with mean μ and variance σ (the error covariance), *i.e.*, $X \sim N(\mu, \sigma)$ [175]. KF addresses the problem of estimation of state x_k of a discrete-time controlled process on the

Table 3.1: Nomenclature for Kalman filter

Notations	Description
$\hat{x}_k$	Posteriori state estimate
$\hat{x}_k^-$	Priori state estimate
$\hat{P}_k^-$	Priori estimated error
$\hat{P}_k$	Posteriori state estimate
K	Kalman gain
v_k	Measurement noise
R	Co-variance for measurement noise
w_k	Process noise
u_k	Control signal
Q	Co-variance for process noise
z_k	Measured value
A,B,H	Constants according to process

basic of previous state x_{k-1} using following equation:

$$x_k = A.x_{k-1} + B.u_k + w_{k-1} \tag{3.1}$$

with a measured value z_k for k^{th} state given by:

$$z_k = H.x_k + v_k \tag{3.2}$$

where $Pr(w) \sim N(0,Q)$ and $Pr(v) \sim N(0,R)$.

Discrete Kalman Filter

KF is a set of mathematical equations that build a predictor-corrector type estimator model to optimally minimizes the estimated error covariance. It provides estimate of a process for k^{th} state by using a feedback control model. In this, filter first estimates the value for k^{th} state based on the current information of the process and then obtains feedback from some measured value, *i.e.*, noisy input. Based on the error in estimated value, Kalman gain is calculated which helps in minimizing error in further iterations. The algorithm converges to

the near optimal result after few iterations.

KF is divided in two groups: time update equations and measurement update equations. The time update equations help in projecting the priori current state value ($\hat{x}_k^-$) and priori error covariance estimates ($\hat{P}_k^-$) for the next step. The time update equations act as predictor equations for estimation model [176].

$$\hat{x}_k^- = A.\hat{x}_k + B.u_k \tag{3.3}$$

$$\hat{P}_k^- = A.\hat{P}_k.A^T + Q \tag{3.4}$$

The measurement update equations provide feedback to the time update equations for incorporating new measurement in priori estimate to obtain an improved posteriori estimate. The measurement update equations are also known as corrector equations.

$$\hat{x}_k = \hat{x}_k^- + K_k.(z_k - H.\hat{x}_k^-) \tag{3.5}$$

$$K_k = \frac{\hat{P}_k^-.H^T}{H.\hat{P}_k^-.H^T + R} \tag{3.6}$$

$$\hat{P}_k = (1 - K_k.H)\hat{P}_k^- \tag{3.7}$$

3.2 Adaptable Bloom Filter

When size of incoming data is not known, it is difficult to determine the optimal BF parameters (m,k) in advance, thus a target threshold for false positives (f_p) cannot be guaranteed. To perform timely analysis on streaming data, an adaptive data structure is required which performs analysis in one pass with minimum computational complexity and less storage overhead. For a stream of network data $S:(x_1,x_2...x_n)$ over a time based window of h time

slots, *i.e.*, $T:(t_1,t_2...t_h)$, this technique propose to addresses following points:

- Analysis of network traffic for a particular time slot.
- Predicting amount of in-coming data in the next slot.
- Allocation of memory for the next time slot based on prediction in present time slot.

The prime focus of the proposed framework is to efficiently query the incoming data within the limited time domain. To deal with instream data and store the information for short time the proposed framework uses a scalable BF [91] with ageing BF properties, *i.e.*, evicting data after fixed time interval. Hybrid hashing,. *i.e.*, combination of partition hashing and double hashing is used which shows effective results in-terms of inter-hash function collision and computational overhead in calculating hash indexes. In the ATBF, Kalman filter [175] is used to make scalable bloom filters adaptive in terms of size and reduce the computational overhead of adding extra filters at run time. Fig. 3.1 provides the basic framework and coming section elaborate the ATBF along with its phases. Following sections provides the details about each phase in ATBF framework.

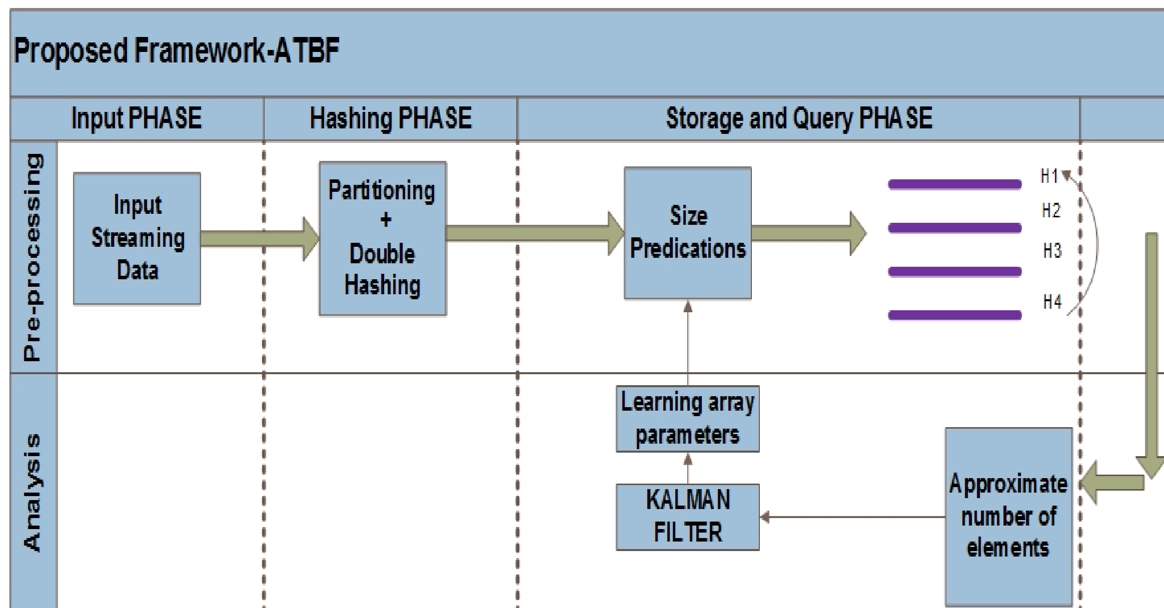

Figure 3.1: Framework for AdapTable Bloom Filter (ATBF)

3.2.1 Input and Hashing Phase

A stream of data $S = (x_1, x_2 ... x_n)$ coming from any resource like sensor, social networking websites, network data and mobile data *etc.*, is assumed to be the input for the proposed framework. It is assumed that data is available only for limited time and hence it has to be processed in the single pass in the defined time frame. Data may be in varied formats like IP address for network data, website names, email address, *etc.* In the proposed scheme, the format of the incoming data is not an issue as all the inputs irrespective of the format (numeric, alphanumeric, text) are hashed using a combination of double hashing and partition hashing. Two independent hash functions $h_1(x)$ and $h_2(x)$ are used to generate k hash functions such that each hash function has a disjoint range of $\left(p = \frac{m}{k}\right)$ (p must be prime for efficient hashing) consecutive bit locations (bucket) instead of having one shared array of m bits *i.e.* partitioning hashing is used, where m is size of array, k is number of hash functions and p is prime number denoting the buckets in an array.

$$(\forall i | i < k)\left[g_i(x) = \{h_1(x) + i \times h_2(x)\} \ mod \ \ p\right] \tag{3.8}$$

To achieve uniformity in maintaining bucket for each hash function at runtime, a parameter σ^i has been introduced, with initial value ($\sigma^0 = p$). For each element $x \in S$ $(\forall i | i < k)$:

$$H_i^j(x) = (h_1(x) + (i-1)h_2(x)) mod \ \ \sigma^j \tag{3.9}$$

Corresponding to j^{th} slice added in i^{th} slot of ATBF, new σ^j is defined to synchronize bucket size for each hash function. $\Phi(x)$ function returns an optimal number p, *s.t.*, $\big[p \leftarrow \Phi(p \geq x$ and p is prime)$\big]$.

3.2.2 Storage Phase

After hashing is done for each incoming element, *i.e.*, ($\forall x_i \in S$), next task is to store the data in the array for a defined time slot say one hour or two hours. For each time slot *i.e.* $t_i \in T$, a BF($ATBF_i[]$) is maintained to store (Fig. 3.1).

Selection of initial size of the BF for each slot in every iteration is critical task because it affects time and query complexity. For the very first iteration, an array of size m_0 is allocated and for next time slots, size of BF is decided based on data received in the previous slot. Initial array size for each time slot t_h is decided on the basis of number of elements accommodated in previous slot t_{h-1}, using an array called Learning Array (LA) which keep the track of size of BF in each slot. The intent of providing an additional counting array is to reduce the slice addition overhead at the run time. The size of the array required for the next time slot is predicted through KF and each slot is provided the required slices at the beginning in the form of a single array instead of multiple chunks called slices. This process helps in adjusting the size of $ATBF_i[]$ to accommodate the dynamic input and reduces search time since query is done on single array instead of slices where we traverse from latest to oldest slice one by one. To maintain the uniformity in the partition hashing, size of slice is decided on the basis of $\phi()$ function. Insertion is performed by setting all hash indexed values *one* in the active slice of filter.

After t time slots, when maximum number of time slot for which data records are maintained is reached, insertion is performed in first slot, *i.e.*, $ATBF_1[]$, by evicting its old data. The proposed model works in round robin manner, *i.e.*, slots after 'h' hours perform insertion on same array during next iteration, *i.e.*, $\left((t_i + o \times h) \leftrightarrow t_i\right)$ where $(o \in Z)$. After completion of insertion in each time slot, $InsertLA()$ function is invoked to update the values for performing size estimation for next time slot (Algorithm 3.1).

One of the major issues in scalable BF is how to measure the defined threshold for addition of new slice. Number of solutions have been proposed for this issue which include 50% percent rule, *i.e.*, threshold is reached when maximum number of one's which a BF

can accommodate reaches 50% of its original capacity; but how to find that a filter is 50% occupied is again a tedious task.

One of the options is to maintain a counter which increment every time an element is added or the number of one's in filter have to be counted after regular intervals. Another method is to keep the track of false positives after every insertion to check whether the results are within the desired false positive rate or not, but these solutions lead to extra computational overhead as one needs to continuously check when a filter get saturated and such operations will definitely dilute the very purpose of using BF.

Proposed scheme addresses the issue of finding threshold for addition of new filter by the usage of buckets generated through partition hashing. Instead of calculating the threshold of the entire array, a function named $CheckFp()$, which uses standard threshold calculation technique, is used to find the threshold value of the randomly chosen bucket. Such technique limits the threshold calculation to a single bucket instead of entire array, reducing the overall computation time. To avoid calling $CheckFp()$ after every iteration, a function $Random()$ has been defined which returns a random value through which $CheckFp()$ function is called, leading to further optimization of the entire process.

For experimental analysis, data is considered for varying time slots *e.g.*, one time slot is equal to four or six hours, *i.e.*, all the hashed data of first time slot is added to the array $ATBF_{t1}$, data of second time slot moves to array $ATBF_{t2}$ and size of $ATBF_{t2}$ is determined by LA, based on the traffic in t_1 time slot. The proposed approach is flexible enough to accommodate n time slots, with each time slot represented by one array. Based on data stored in these BFs, *i.e.*, $ATBF_{1...t_n}$ further analysis like peak hour analysis, detecting approximate number of users in each time slot and server utilization are performed.

Input : Insertion($ATBF[.], p, S, T, _{i=1}^{i=k} H_i$)
Output: Insert $x_i \in S$ for $t_i \in T$ in $ATBF_i$ array

```
1  for ∀i|i ≤ T do
2      LA[i][][] ← InsertLA()
3      r_i ← 1
4      σ^{r_i} ← φ((LA[i] × p)/k)
5      ATBF_i[r] ← SizeOf(σ^{r_i} × k)
6      C_Slice ← 1
7  end
8  for ∀x_j ∈ S do
9      while t_c == t_i do
10         if t_h ATBF_i[r] > thresVal then
11             σ^{r_i} ← φ((s^{r−1} × p)/k)
12             r ← r + 1
13             SizeOf(ATBF_i[r] ← σ^{r_i} × k)
14             C_Slice ++
15         else
16             for ∀z|z ≤ k do
17                 h_z(x_j) ← H_z(x_j)
18                 ATBF_i[r](h_z(x_j)) ← HIGH
19             end
20         end
21         if Random() == TRUE then
22             CheckFp(ATBF[r])
23         end
24     end
25     InsertLA(FR_{LA[i]}, C_slice)
26 end
```

Algorithm 3.1: ATBF: Insertion procedure

3.2.3 Query in ATBF

To determine the occurrence of a particular element in a time window, $Query()$ function is used. $Query(LA[], p, Q, T, _{i=1}^{i=k} H_i)$ in ATBF checks each BF, *i.e.*, $\left(ATBF_{t_i} | \forall t_i \in T\right)$ from latest array to oldest array and in each BF all slices (if added), *i.e.*, from r to 1 are checked corresponding to the queried element. Query process is made fast by calculating hash functions at the run time, *i.e.*, for a particular query, all hash functions are not computed in advance, each hash function is calculated and comparison is performed in defined bucket of hash function. If bit at hash index is *one* then next hash function is computed and comparison is performed

otherwise query process terminates. Query process terminates as soon as first *zero* is encountered in a bucket and thus time is saved as remaining hash functions for other buckets are not calculated. The query process is terminated successfully if element is found, *i.e.*, all *ones* are returned (Algorithm 3.2).

```
    Input  : Query(LA[], p, Q, T, ^{i=k}_{i=1} H_i)
    Output: Return TRUE or FALSE for Query(y ∈ S)
 1  for ∀ Query elments(y)|y ∈ Q do
 2      for ∀ Time slots(t)|t ∈ T do
 3          for l = (ATBF_t[.]...1) do
 4              if (ATBF_t[l](^{i≤k}_{i=1} h_i(y)) == 1) then
 5                  Return Element Found
 6              end
 7          end
 8      end
 9      Return Element Not Found
10  end
```

Algorithm 3.2: ATBF: Querying in proposed framework

Lemma 3.2.1. *Worst case query time complexity in proposed model for filter with h time slots, assuming r slices in each slot with k hash functions is always less then* $O(rhk)$*.*

Proof. Searching starts with hashing of the query element y, *i.e.*, $\left(\forall i | y \in Q, h^{k}_{i=1}(y) \leftarrow H^{k}_{i=1}(y)\right)$ and corresponding hash indexes are checked for value zero. Query process begins from latest time slot to oldest one *i.e.* $t_{h\text{to}1}$ and same is followed in search from slices $s_{r\text{to}1}$ in BF. During search operation when hash indexed value *zero* is encountered, searching for that particular array is terminated and previous slice is not searched. In such case, number of evaluated for unsuccessful query, *i.e.*, not finding the element queried is always less then k hash functions. For a BF with r slices, it will be always less then $O(rk)$. Thus, for h time slots from $t_{1...h}$ having r slices each, worst case query complexity is always less than $O(rhk)$. □

3.2.4 Learning Array

Since the amount of incoming data will keep on varying in every time slot, the size of array will change. Calculating the threshold after every addition and providing new slice accordingly in every time slot at run time requires lot of computation which can be saved if record of size of array *i.e.* a counter C_{slice} is maintained which keeps the count of number of slices added in a particular $ATBF_{ti}$ in a particular time slot. Initially a constant size BF m_0 is allocated for first time slot and if the incoming data increases, more slices are added and counter c_{slice} is incremented. To make proposed framework adaptive, a Learning Array $LA[value][c]$ is initially added. The main role of *LA* is to record the array size of $ATBF_{ti}$ after filling of data in each time slot. This helps in predicting the array size required in the next time slot.

With the help of *LA* an optimal size of $ATBF_i[]$ required for successive time slots is decided. If for a time slots no slices are added, indicating unused BF bits then value of *LA* is decremented for next time slot (Algorithm 3.3).

To make the functioning of *LA* more efficient, KF is used for predicting array size. The approximate number of elements are estimated through Algorithm 3.3 and the number of slices 'x' added to the initial filter in a particular time slot serve as input parameters to KF. After observing incoming data patterns for particular t_i, proposed model decides the optimal size required for next time slot, *i.e.*, t_{i+1}, reducing the overhead of slice addition at run time for each time slot, thus improving the search time complexity of $ATBF_i[]$.

Number of slices (s_n) added to a particular time slot is recorded in *LA*, from this we can compute total size of filter required for particular time slot, *i.e.*, $\hat{S}_s$. The number of elements n_a accommodated by ATBF is given by:

$$n_a \approx m_0 s^i (ln(t_h)) \tag{3.10}$$

From the approximate number of elements accommodated, size of filter, *i.e.*, $\hat{S}_e$ is calcu-

lated as:

$$\hat{S}_e = n_a \times k \tag{3.11}$$

These two estimates for the size of BF act as input for KF and help the framework to predict the approximate size for coming time slots in further iterations.

Since the incoming data is one dimensional, KF parameters A, B, H, Q and R in Eq. $(3.1 - 3.7)$ have constant values in the proposed model. u_l is assumed to be zero because no control signal is used in the model. $\hat{S}_l$ denotes posterior estimated size and $\hat{S}_l^-$ denotes priori estimated size for i^{th} time slot and for l^{th} iteration. Thus

Time Update:

$$\hat{S}_l^- = \hat{S}_{l-1} \tag{3.12}$$

$$\hat{P}_l^- = \hat{P}_{l-1} \tag{3.13}$$

Measurement Update:

$$K_l = \frac{\hat{P}_{l-1}^-}{\hat{P}_{l-1}^- + R} \tag{3.14}$$

$$\hat{S}_l = \hat{S}_{l-1}^- + K_l(z_l + \hat{S}_{l-1}^-) \tag{3.15}$$

$$\hat{P}_l = (1 - K_l)\hat{P}_{l-1}^- \tag{3.16}$$

$$z_l = .5(\hat{S}_s + \hat{S}_e) \tag{3.17}$$

Lemma 3.2.2. *Use of Kalman Filter based LA in proposed model reduces the query complexity of ATBF in handling in-stream data compared to scalableBF by approximate $O(\frac{1}{r})$ i.e. $\approx< O(k)$, where r is number of slices added and k is number of hash functions considered.*

Proof. In case of scalable BF, when an array crosses the defined threshold, a new slice is added and insertion is performed. Assuming N_s is elements in stream, let's assume scalable BF needs r slices to accommodate the incoming data. Query process in scalable BF is ac-

complished by testing the presence of query element in each filter, starting from active filter to oldest filter. Search complexity for worst case analysis is $O(k \times r))$.

In $ATBF$ first time slot is functionally similar to scalable BF, but size for next time slot can be predicted using *LA* and KF. Predicting size for next time slot leads to decreased computational overhead as addition of new slices at run time is not required. Further, since the size of new array is combination of initial array and additional slices, inter-function collisions are reduced especially when partition hashing is used. From second time slot onwards the query complexity is always less than $O(rk)$, because from the the second array onwards the number of new arrays added will always be less than r. In best case, when no extra slice is added in future time slots *i.e.* the input data arrival rate is constant, search complexity is equal to standard BF $\approx O(k)$. Thus, for the 'h' time slots, search time complexity for $(h-1)$ slots is reduced drastically.

□

Input : InsertLA($LA[i], j$)
Output: Return predicted size for next iteration

```
1  if (j > 1) then
2  |  if LA[i] < j then
3  |  |  LA[i][c]++
4  |  end
5  end
6  if (j == 1) then
7  |  if FR_{LA[i]} < thres_{fill} then
8  |  |  LA[i][] -= 1
9  |  |  Exit()
10 |  end
11 end
12 ŝ_n ← LA[i][c]
13 Ŝ_s ← m_0 Σ_{i=1}^{s_n} {i × m_0/k}
14 Ŝ_e ← Count(ATBF_i[], r)
15 Set Ŝ_1^- = 0
16 Set P̂_1^- = 1
17 for (l : 1 to ℓ) do
18 |  z_l = .5(Ŝ_s + Ŝ_e)
19 |  Time updation
20 |  Ŝ_l^- = Ŝ_{l-1}
21 |  P̂_l^- = P̂_{l-1}
22 |  Measurement updation
23 |  K_l = P̂_{l-1}^- / (P̂_{l-1}^- + R)
24 |  Ŝ_l = Ŝ_{l-1}^- + K_l(z_l + Ŝ_{l-1}^-)
25 |  P̂_l = (1 - K_l) P̂_{l-1}^-
26 end
27 LA[i][] ← Ŝ_ℓ
```

Algorithm 3.3: ATBF: Learning array algorithm

3.2.5 Network Traffic Analysis for a Particular Time Slot

The standard algorithms for counting number of element in streams like Count Min Sketch(CMS), probability based counter and "Datar-Gionis-Indyk-Motwani"(DGIM) algorithm are quite accurate but need lot of extra space and have computational overhead. Proposed model provides a rough estimate of number of elements using KF.

To calculate the approximate number of elements in a particular time slot t_i, $Count_i()$ is

used with initial parameters like slices added in the array (r), threshold fill ratio (f_r), number of hash function (k), initial size of filter (m_0) and prime number used in first filter (p). Two methods have been used to calculate the number of elements in a particular time slot and results are verified by both methods (Algorithm 3.4). In the first method, growth parameter (s) is considered as ($s = 2$) for slow growing data and ($s = 4$) for fast growing data with optimal threshold t_h value as 50% same as that considered in scalable BF [91]. Total number of elements accommodated by BF(N_i) is given by:

$$N_i \approx \left(m_0 \times 2^i \times (.693)\right) \tag{3.18}$$

Second method is to calculate the total size of the BF used and then predict the number of elements accommodated by it. Since σ^g is prime for g^{th} slice, *i.e.*, size of bucket and number of buckets are equal to number of hash functions (k), total size of an array with r slice of σ bits, is given by:

$$\begin{aligned} Total\ Size &= (r \times \sigma) \qquad (3.19) \\ where:\ & r\ isnumber\ of\ Slices \\ & \sigma\ is\ Size\ of\ Slice \end{aligned}$$

Thus total size(t_s) of $ATBF_i$ with r slices is given by:

$$t_s \leftarrow \sum_{g=1}^{r} (\sigma^g \times k) \tag{3.20}$$

Bits available for insertion in ATBF are determined by threshold fill ratio, (f_r), total available bits t_a are:

$$t_a \leftarrow t_s \times f_r \tag{3.21}$$

Thus, maximum number of elements (E_a) accommodated by $ATBF_i$ are:

$$E_a \leftarrow \frac{t_a}{k} \tag{3.22}$$

Input : $Count_i(ATBF_i[], r)$
Output: Return approximate number of items in i^{th} time slot

1. *Method 1:*
2. $N_i \leftarrow ln(f_r) \times m_0.s^r$
3. *Method 2:*
4. **for** *g:1 to r* **do**
5. $\quad \sigma^g \leftarrow \phi((g \times p)/k)$
6. **end**
7. $t_s \leftarrow \sum_{g=1}^{r}(\sigma^g \times k)$ $E_a \leftarrow \frac{t_s.f_r}{k}$

Algorithm 3.4: ATBF: Approximate number of elements

3.3 Observations and Analysis

All the experiments have been performed on $i7-3612QM$ CPU @ 2.10 GHz with 8 GB of RAM. To maintain the uniformity in the results *CityHash* 64 bit library [177] is used to compute the hash functions for PDS.

Two data sets from different application domains have been considered for evaluating the performance of ATBF, one is data of pickup calls of UBER cabs [178] and other is incoming data generated for network server. Results are represented for first few iterations only, which can be extended to n number of iterations according to application's requirements. In all experiments five hash functions have been used with initial size of the filter m_0 as 1285 bits, slice size σ^0 for all the iterations is considered as 275 $\left(s = \frac{1285}{5}\right)$ for first array in all iterations.

Table 3.2 and 3.3 provide the count of actual number of users and number of users identified using KF. $\hat{S}^{-}$ represents the size of BF in current iteration by considering the previous one, initially size of BF is set to m_0. Number of slices added in $ATBF_i$ is maintained by

c_{slice} counter. $\hat{S}$ is the array size predicted by KF for the next iteration. Peak hours analysis is performed by "peak hour ranking" with 1 indicating maximum and 5 as minimum value. Peak hour rank help in identifying changing patterns of data in current iteration in relation to the previous iteration. Initially for all iteration, peak hour rank is set to a default value of -1. This ranking system helps in allocating resources in accordance with the frequency of incoming data.

3.3.1 Experiment 1: Uber Pickups Data Sets

Data of 14,270,479 trips of Uber Pickups in New York City from Jan-2015 to June-2015 for around 265 different locations is considered for 12 hours a day as input. The data set is time series based having attributes like date, time, location id and base number. A snapshot of an instance of data is shown in Fig. 3.2. In proposed model, "location id" is used as insertion element in BF and attributes "Pickup_date" and "Time" are used to select the size of time slot.

	Dispatching_base_num	Pickup_date	Affiliated_base_num	locationID
1	B02617	2015-05-17 09:47:00	B02617	141
2	B02617	2015-05-17 09:47:00	B02617	65
3	B02617	2015-05-17 09:47:00	B02617	100
4	B02617	2015-05-17 09:47:00	B02774	80
5	B02617	2015-05-17 09:47:00	B02617	90
6	B02617	2015-05-17 09:47:00	B02617	228
7	B02617	2015-05-17 09:47:00	B02617	7
8	B02617	2015-05-17 09:47:00	B02764	74
9	B02617	2015-05-17 09:47:00	B02617	249

Figure 3.2: An instance from data set of Uber pickups

Table 3.2 shows the result of two days for peak time slot in Uber pickups, for date 1-Jan-2015 and 2-Jan-2015 using two time slot ranges : four hours as a single time slot and six hours as a single time slot respectively.

Table 3.2: BF size prediction and Peak hour analysis for UBER pickup call for 1-Jan-2015 and 2-Jan-2015

Iteration	**No. of actual users**	**Initial array size ($\hat{S}^{-}$)**	**No. of slots added (C_{slice})**	**No. of users pre-dicted by ATBF**	**Error (In %)**	**Size of BF pre-dicted for next time slot (in bits) ($\hat{S}$)**	**Previous Peak hour ranking**	**Current Peak hour ranking**
Time Slot = 4 hours 1/1/2015								
Time Slot 1(1 to 4)hrs.	5864	1285	11	5746	2.01	28160	-1	1
Time Slot 2 (5 to 8)hrs.	2389	28160	0	2358	1.3	24320	-1	3
Time Slot 3 (9 to 12)hrs.	2922	24320	0	2935	-0.4	20736	-1	2
2/1/2015								
Time Slot 1(1 to 4)hrs.	1765	1285	4	1732	1.9	8960	-1	3
Time Slot 2 (5 to 8)hrs.	2437	8960	2	2387	2.1	14336	-1	2
Time Slot 3 (9 to 12)hrs.	2534	14336	0	2456	-0.9	20736	-1	1
Time Slot = 6 hours 1/1/2015								
Time Slot 1 (1 to 6)hrs.	7314	1285	12	7287	0.4	36660	-1	1
Time Slot 2 (7 to 12)hrs.	2326	36660	0	2342	-0.6	32256	-1	2
2/1/2015								
Time Slot 1 (1 to 6)hrs.	2915	1285	7	2867	1.7	17408	-1	1
Time Slot 2 (7 to 12)hrs.	3312	17408	0	3264	1.5	17408	-1	2

3.3.2 Experiment 2: Incoming Data on a Network Server

Table 3.3 provides the results for server utilization and peak hour analysis. Experiment is done for six time slots of one hour each. The results are simulated on network traffic with maximum per hour capacity of server as 15000 users. In table 3.3, Eqs. (3.12-3.17) are used for estimates. In table 3.3, the column III ("Inital size of array") acts as $\hat{S}_l^-$, in Eq.(3.12). Value of error variance *i.e.* $\hat{P}_l^-$ in Eq.(3.13) is computed from the difference in column VII ("Size of BF for next slot (in bits)") and column III("Initial size"). To compute the Kalman gain in Eq. (3.14), value of $\hat{P}_l^-$ from Eq. (3.13) is used and value of constant variable R, considered for this experimental setup, is 1. In Eq. (3.17), value of z_l represented in column V ("No. of user predicted"), is calculated by taking the mean of values coming from two different approaches, *i.e.*, using Eqs.(3.10,3.11). All these values act as input for Eq.(3.15), which predicts the size of BF for next slice. Server utilization is given by ($\frac{n}{N} \times 100$), where n is approximate number of users detected and N is server capacity. The network data has IP address, date and time as its attributes. IP address is used as primary element for insertion in proposed model.

Table 3.3: Hourly analysis of Server Utilization, Peak Hour and BF size prediction for next time slot for incoming data on a network

Iteration1	**No. of actual users**	**Initial array size ($\hat{S}^{-}$)**	**No. of slots add- ed**	**No. of users pre- dicted by ATBF**	**Error (In %)**	**Size of BF pre- dicted for next time slot (in bits) ($\hat{S}$)**	**Server utiliza- tion (%)**	**Previous Peak hour ranking**	**Current Peak hour ranking**
Time slot 1	10000	1285	15	9975	0.25	51200	68.53	-1	2
Time slot 2	12000	51200	2	12145	-1	62210	82.97	-1	1
Time slot 3	9000	62210	0	9216	-2	46080	61.44	-1	3
Time slot 4	6000	46080	0	6052	-0.8	32256	43.01	-1	5
Time slot 5	8000	32256	2	7952	0.6	41216	54.97	-1	4
Time slot 6	4000	41216	0	3924	1.9	20736	27.65	-1	6
Iteration 2									
Time slot 1	9000	1285	14	8982	0.2	46080	60.84	2	3
Time slot 2	14000	46080	5	14248	-1.7	74240	98.99	1	1
Time slot 3	13000	74240	0	13184	-1	70400	92.17	3	2
Time slot 4	7000	70400	0	7013	-1	36352	48.09	5	4
Time slot 5	3000	36352	0	2989	0.4	21365	23.21	4	6
Time slot 6	4000	21365	0	3968	0.8	20736	27.65	6	5

3.3.3 Performance Evaluation of scalable BF and ATBF

The performance of scalable BF and ATBF is compared on the basis of computational time taken for hashing, querying and extra slice addition as the incoming data increases. Fig. 3.3 provides comparative analysis on the basis of hashing complexity of scalable BF and $ATBF$. In scalable BF, for every input, hash value is computed for all hash functions (k) while in $ATBF$ only two hash functions have been used to generate k hash functions, leading to major decrease in computational overhead. In the figs.(3.3 3.5 3.4), SBF stands for scalable BF.

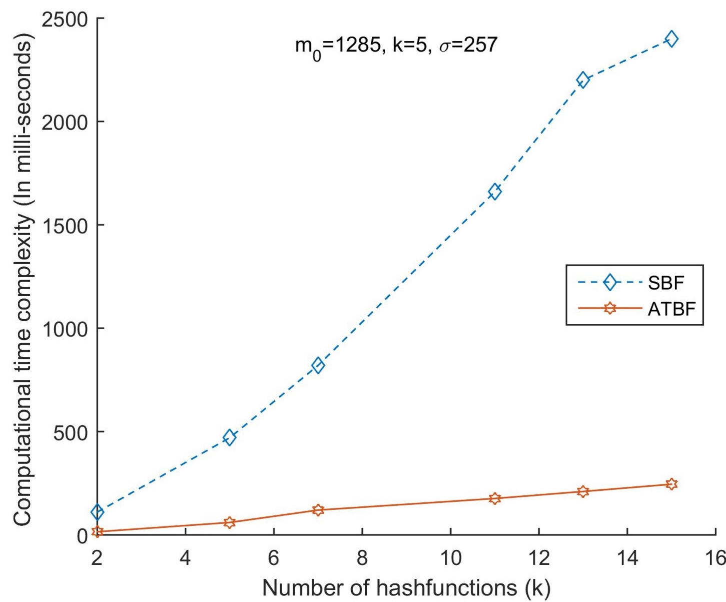

Figure 3.3: Computational time complexity vs. number of hash functions

Query complexity and slice addition overhead for both the filters is checked on dynamically growing environment. Both filters, *i.e.*, scalable BF and ATBF start with size $m_0 = 1285$ and 5000 elements have been considered for first iteration and each iteration adds 1000 element to previous value.

Fig. 3.4 depicts the analysis performed on the basis of number of slices needed to accommodate the dynamically growing data. In scalable BF, filter starts with size m_0 and as the number of incoming elements increase more slices are added in each iteration. In case of ATBF, as the incoming data increases the size of t_{n+1}^{th} iteration is predicted in advance, based

on the elements accommodated per iteration in t_n^{th} using Kalman filter.

Based upon the data considered for experiments, *i.e.*, 1000 elements increase from previous value per iteration, in t_{n+1}^{th} iteration only one slot is added to accommodate additional elements in ATBF. The graph of ATBF becomes constant after first iteration since one slice is added in every successive iteration and no overflow of data is registered (Fig. 3.4). Hence overhead of adding new slices at the run time is reduced to a large extent in the proposed scheme.

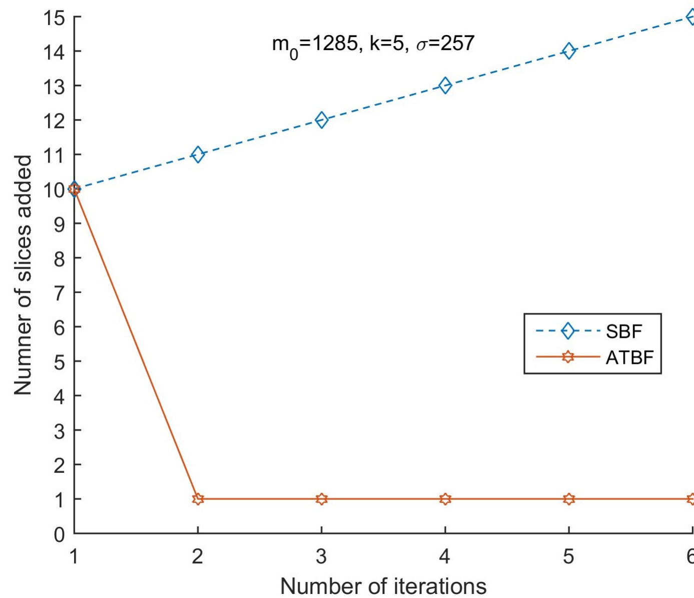

Figure 3.4: No. of slices required vs. number of iteration for dynamic dataset

Fig. 3.5 depicts the comparative analysis of worst case query complexity for an element (when element is not present in the set), *i.e.*, scenario where all slices need to be scanned. As the size of data grows in each iteration in scalable BF, more slices are added to accommodate the data elements. In scalable BF, all slices need to be scanned in query process which increases the query complexity many folds. ATBF has the advantage of size adaptation from second iteration onwards. For the first iteration process is similar to scalable BF, but from the ith iteration (where $i \neq 1$), size of BF is predicted on the basis of previous $(i-1)^{th}$ iteration. The predicted size of BF is added as single BF. So, in querying process only one BF needs to be scanned, thus the total cost is $O(k)$. As the data grows, the number of slices added are

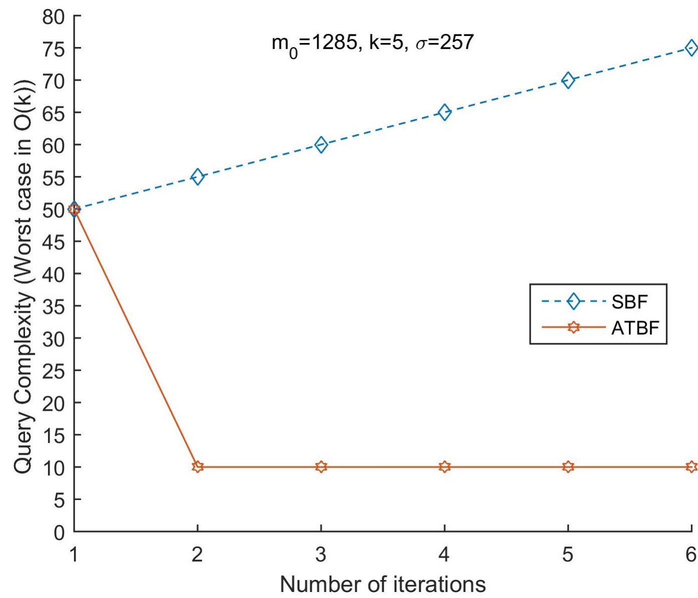

Figure 3.5: Worst case query complexity vs. number of iterations

always less than scalable BF for same number of elements thus search time complexity of ATBF shows significant improvement.

3.4 Discussion

ATBF is suitable for the application where variation in incoming data is dependent on time. Combination of partitioning hashing and double hashing reduces collisions and computational overhead. Further, query complexity of dynamic data has also been reduced by using learning array in the proposed system. The major contributions of the proposed ATBF are described below:

- Partition hashing has been used which leads to less inter-hash function collision. Further usage of double hashing where only two hash functions are used to generate all k hash functions decreases the computational overhead.

- A learning array has been introduced which stores the size of BF required in next iteration and KF has been used to predict the size of BF required for next iteration.

- ATBF is suitable for the application where variation in incoming data is dependent on the time slots of incoming data.

Next chapter elaborate the second proposed technique, *i.e.*, FingerPrint based Stable Bloom Filter, which is used to detect duplicates in the incoming streaming data.

Chapter 4

FingerPrint Based Duplicate Detection in Streamed Data

In this chapter, a novel data structure named as FingerPrint Stable Bloom Filter (FP-SBF) is proposed (Section 4.2), a variant of stable BF for duplicate detection in streaming data using fixed memory and constant query time.

4.1 Duplicate Detection in Streaming Data

4.1.1 Problem Statement

Given a input stream $S = \{x_1, x_2 \ldots x_i \ldots x_n\}$ with N elements; where $(N \rightarrow \infty)$, *i.e.*, unbounded stream of data, identify whether x_i appears in S or not in a given space M, where $(M << N)$. Challenges associated with duplicate detection include:

i. Preprocessing effort for each element should be minimum for fast results.

ii. Eviction of stale data in necessary as the memory size of the filter is fixed.

iii. Response time for query should be minimum and independent of the size of data.

4.2 FingerPrint Stable Bloom Filter

The proposed FP-SBF uses stable BF with fingerprint bits and optimization mechanism which reduces the computational time and decreases the false positives as well as false negatives using d-left hashing. Optimized deletion mechanism is used to evict the data from BF to make more space for in-coming data. It uses constant space irrespective of the size of in-coming data and accommodates more number of elements before reaching the saturation stage.

FP-SBF is an array of indexing $[1...m]$ of size M where each element of array is represented by d bits, *i.e.*, $(M = m \times d)$. d bits of each element are further divided into two parts: c bits for buckets and f bits for fingerprint as indicated in Fig. 4.1. Bucket bits (c bits) are used as counter, where range of counter values is $\Re_c \leftarrow \{1, \{Max = (2^c - 1)\}\}$ and $f = (d - c)$ bits are fingerprint bits, also called FingerPrint Cell (FPC). FP-SBF uses d-left hashing with $(k+1)$ hash functions, where k hash functions are used to select the appropriate index from m elements and update bucket's counter to Max and one hash function H_{fp} is used to update fingerprint cell of each selected index. Since available memory space is fixed, old data should be evicted to make room for new data. Parameter ξ called Eviction Rate (ER) is used to control the eviction of stored data in BF.

The task of detecting duplicate from the streams using FP-SBF consist of following steps (Algorithm 4.1):

i. **Detection:** Query the existing data to find whether current data element x_i exists in the filter or not.

ii. **Deletion:** Remove the data randomly by decrementing the values of buckets to make spaces for new in-coming elements.

iii. **Insertion:** Insert the data in the filter by updating the corresponding bucket and fingerprint cells, if the element is not present in the array.

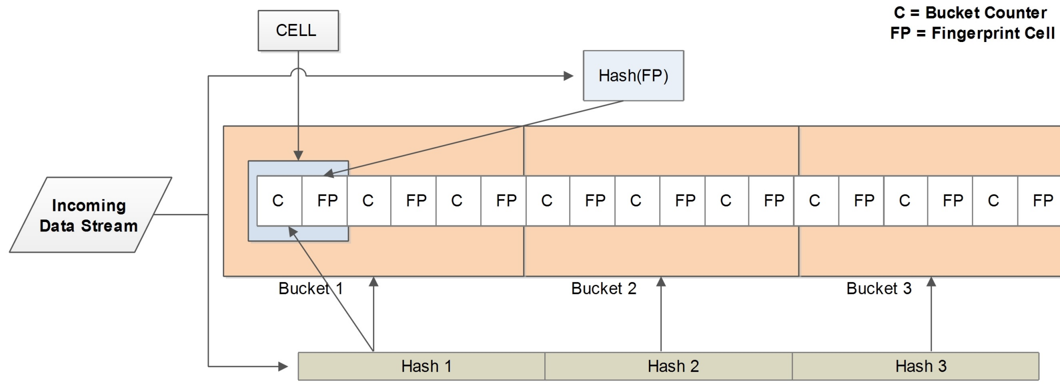

Figure 4.1: Structure of proposed Finger Print based Stable Bloom Filter

Input : FP-SBF($FP-SBF[], S, H_k, H_{fp}$)
Output: Detect duplicate item in stream.

```
1  for ∀x_i ∈ S do
2  |  if Detection(x_i) == TRUE then
3  |  |  Element appeared previously in the filter, no insertion required
4  |  else
5  |  |  if Random(ξ) > Threshold then
6  |  |  |  Delete()
7  |  |  end
8  |  |  Insert(x_i)
9  |  end
10 end
```

Algorithm 4.1: FP-SBF: Duplicate detection in streams

4.2.1 Detection

To detect an element x_i in FP-SBF, $k+1$ hash functions H_1^k and H_{fp} are used. Corresponding to k hash functions, indexes are generated, *i.e.*, $h_1^k \leftarrow H_1^k$ and following checks are performed (Fig. 4.2):

i. If any of the bucket corresponding to h_1^k is set to zero, duplicate detection mechanism returns false, *i.e.*, $x_i \notin (x_1...x_{i-1})$.

ii. If all buckets are non-zero, $H_{fp}(x_i)$ is computed for x_i and all fingerprint cells of indexes h_1^k are checked. If $H_{fp}(x_i)$ is not found in any FPC, duplicate detection mecha-

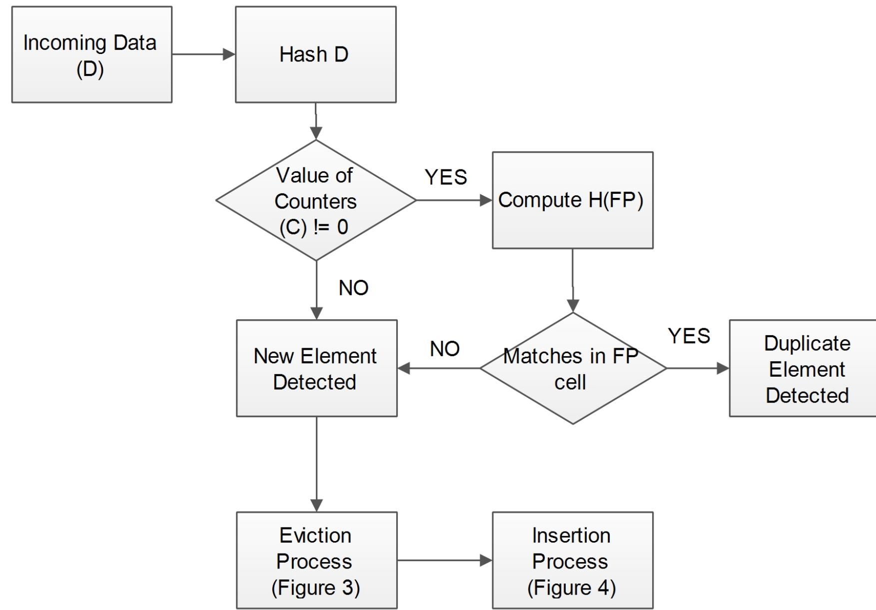

Figure 4.2: FP-SBF: Flowchart of detection process

nism returns FALSE, *i.e.*, $x_i \in (x_1 ... x_{i-1})$.

iii. If step i and ii (mentioned above) are false then x_i is duplicate, *i.e.*, it is previously seen in the stream $(x_1 ... x_{i-1})$, hence no need to perform further operations.

Deletion and insertion operations are invoked when detection process fails, *i.e.*, element is not found in the stored stream. False positives may occur in the detection process due to hash function collision of two different elements, *i.e.*, bucket set high by x_i return detection results as *true*, resulting in identification of a distinct element as duplicate element (Algorithm 4.2).

A false negative in duplicate detection on streamed data occurs when a duplicate element (x_i) is wrongly reported as distinct element. Eviction process in FP-SBF leads to decrement the value of buckets to zero, *i.e.*, for an element which is present in stream, the value of bucket is decremented to zero and this element is regarded as distinct element although it is a duplicate element.

Input : Detection($FP-SBF[], x_j, H_k, H_{fp}$)
Output: Check whether x_i element is present in the stream or not

```
1  ∀i|(1 < i < k) Calculate hash h^k_{i=1}(x_j) ← H^k_{i=1}(x_j)
2  if (∀i|(1 < i < k)), [h^k_{i=1}(x_j) > 0] then
3      h_{fp}(x_j) ← H_{fp}(x_j)
4      if (∀i|(1 < i < k)), [FPC_i = h_{fp}(x_j)] then
5          Return TRUE
6      else
7          Return FALSE
8      end
9  else
10     Return FALSE
11 end
```

Algorithm 4.2: FP-SBF: Detection process

4.2.2 Deletion or Eviction of Data

To accommodate unbound data in constant memory, it is necessary to evict the data at regular intervals. FP-SBF applies an optimized deletion approach which evicts the data quickly and decreases false positives (Fig. 4.3).

k indexes are selected randomly with probability p and decremented by a value z, *s.t.* $z \in (1,2...,2^c-1)$. Deletion process is invoked after i iterations where i is selcted randomly. This process is controlled by Evicting Rate (ER) parameter ξ, $\left(\Re_\xi \rightarrow \{0,1\}\right)$. Steps followed to accomodate in-coming data are :

i. In every iteration, check if $Random(\xi)$ returns a value more then defined threshold, where $Random(\xi)$ is a function which generates random values as per the value of ξ provided to it. If $Random(\xi)$ is greater than defined threshold then deletion is performed else step ii to iv are skipped.

ii. If step i is false, *i.e.*, threshold is not reached, select k index, *i.e.*, $h_1^k(D)$ for deletion where probability of any cell being selected is $(\frac{p}{m})$.

iii. Select a random value of z for each selected index $h_1^k(D)$ from a given range $\Re_z \rightarrow \{1,2^c-1\}$.

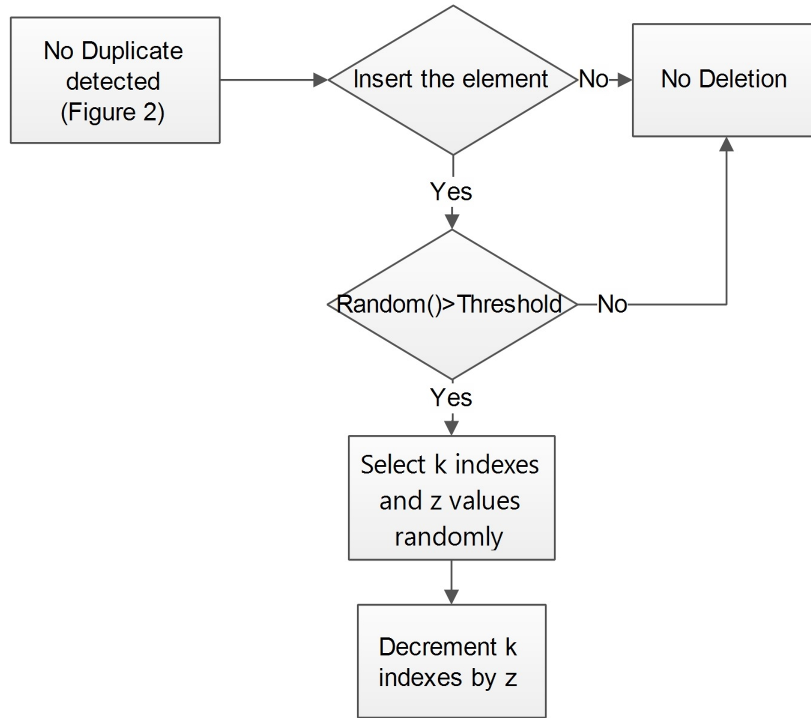

Figure 4.3: FP-SBF: Flowchart of eviction process

iv. For each index selected in Step iii decrement the value of bucket by z.

Deletion frequency can be increased by increasing value of ξ. Fingerprint cell of selected indexes remains unaffected in the deletion process (Algorithm 4.3).

Input : Delete($FP - SBF[], p_d, \Re_z$)
Output: Decrement k random cells by z value

1. Select k cells from m with probability p
2. $L^k_{i=1}$ are the selected cells for decrement operation.
3. Select a value to be decremented from each bucket
4. $z \leftarrow Random(\Re_z)$
5. $(\forall i | (1 < i < k))\ Bucket[L_i] = (Bucket[L_i] - z)$

Algorithm 4.3: FP-SBF: Deletion process

4.2.3 Insertion

To insert an element x_i, $k+1$ hash functions are used. Fig. 4.4 demonstrates the steps followed for insertion of an element in FP-SBF (Algorithm 4.4).

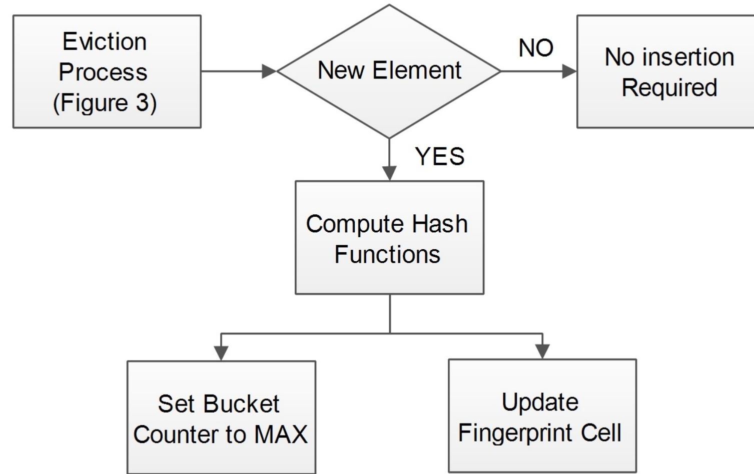

Figure 4.4: FP-SBF: Flowchart of insertion process

i. Hashing is done to get the indexes *i.e.* $h_1^k(x_i) \leftarrow H_1^k(x_i)$.

ii. Buckets are updated:

$Set(h_1^k(x_i)) = Max$, where $Max = (2^c - 1)$.

iii. Fingerprint cell is updated:

$h_{fp}(x_i) \leftarrow H_{fp}(x_i)$

$h_{fp}(x_i)$ is used to updated the corresponding FPC of selected indexes.

$FPC_i \leftarrow (FPC_i)$ XOR $(h_{fp}(x_i))$

Input : Insert($FP - SBF[], x_j, H_k, H_{fp}$)
Output: Update indexes to max according to hash indexes

1 **for** *(i=1,i$\leq$ k,i++)* **do**
2   Calculate hash $h_{i=1}^k(x_j) \leftarrow H_{i=1}^k(x_j)$
3     $Set(Bucket[h_i] \leftarrow Max)$
4     $h_{fp}(x_i) \leftarrow h_{fp}(x_j)$
5     $FPC_i \leftarrow (FPC_i)$ XOR $h_{fp}(x_j)$
6 **end**

Algorithm 4.4: FP-SBF: Insertion process

4.2.4 Stable Property

Stable Property (SP) assures that after ℓ iterations, the fraction of zeros in stable BF/FP-SBF will be fixed, *i.e.*, they will not depend on any parameter. In FP-SBF, number of iterations, ℓ required to achieve SP is depended on eviction rate ξ. It has been proved theoretically and experimentally that SP plays an important role in determining false positive rate.

Theorem 4.2.1. *In FP-SBF with m cells, each cell is updated to Max with a probability* $p_i = (\frac{k}{m})$ *and each cell is decremented by a value* $z \in (1, ..2^c - 1)$ *with a probability* $p_d = (\frac{p}{m})$. *The probability that cells become zero after N iterations as* $(N \rightarrow \infty)$ *is constant, i.e.,*

$$\lim_{N \to \infty} \sum_{i=1}^{N} Pr(ASBF_i = 0) \Rightarrow \ Constant \tag{4.1}$$

where $ASBF_N$ *is value of cell at the end of N iterations.*

Proof. For each element of stream, three possible operations are detection, deletion and insertion. Only deletion and insertion will effect the values of buckets. After N operations, a bucket can be set to $Max \leq N$ times and decremented with certain value $< N$ times . Let p_i is probability of a bucket to be selected for insertion given by $(p_i = \frac{k}{m})$; p_d is probability of a bucket to be selected for deletion *i.e.* $(p_d = \frac{p}{m})$ and ξ is evicting rate for FP-SBF. A_l denotes that no insertion has been performed in the bucket in recent l iterations ($l < N$); probability of event A_l is given by:

$$Pr(A_l) = (1 - p_i)^l p_i \tag{4.2}$$

A_N denotes that no insertion is performed in any bucket in N iterations, its probability is:

$$Pr(A_N) = (1 - p_i)^N \tag{4.3}$$

Since no insertion operation is performed in a particular bucket for N iterations, probability of having value zero in that bucket following event A_N is:

$$Pr(ASBF_N = 0|A_N) = 1 \tag{4.4}$$

P_0, probability that after deletion operation value of bucket is zero is :

$$P_0 = \xi \times p_d \times Pr(P_i = 0|(FP - SBF[i] = x)) \tag{4.5}$$

where value of $Pr(P_i = 0|(FP - SBF[i] = x))$ is dependent on the current value in the bucket and z value selected from Deletion Value Set(DVS) $(1, 2^c - 1)$ is given by:

$$Pr(P_i = 0|(FP - SBF[i] = x)) = \frac{\Delta[(z \geq x)\&(z \in DVS)]}{\Delta(z \in DVS)} \tag{4.6}$$

$\Delta()$ is function which counts the number of values of $z \in DVS$, to calculate the sample space and favorable events for deletion.

When a bucket follow event A_l, the probability that after N iterations where $N > l$, the decrement operation will reset the bucket value to zero is given by:

$$Pr(ASBF_N = 0|A_l) = \sum_{j=Max}^{l} \binom{l}{j} P_0^j (1 - P_0)^{l-j} \tag{4.7}$$

Thus, for a random element, probability $Pr(ASBF_N = 0)$ that the bucket is zero after N iterations is given by:

$$Pr(ASBF_N = 0) = \sum_{l=Max}^{N-1} [Pr(ASBF_N = 0|A_l)Pr(A_l)] + Pr(ASBF_N = 0|A_N)Pr(A_N) \tag{4.8}$$

In FP-SBF, if bucket is not set to Max in l iterations, more then one operations are required to decrement its value to zero, *i.e.*, for $l \leq Max$; cell can be decreased to zero and for $l = N$, A_N event occurs. So from Eq. (4.8), it is proved that $\lim_{N \to \infty} Pr(ASBF_N = 0)$. Hence, proposed FP-SBF follows stable property principle of stable BF efficiently and with less computational complexity. □

Definition: *Convergence Rate (CR):* From the stable point property, each bucket has fixed probability of being set to Max and constant probability of being reset to zero after certain iterations. P_0, probability that a cell becomes zero is same for all buckets. Thus, expected number of zeros in FP-SBF converges exponentially and Convergence Rate (CR) can be derived using Eq.(4.8), as follows:

$$CR = Pr(ASBF_N = 0) - Pr(ASBF_{N-1} = 0) \tag{4.9}$$

Definition: *Stable point:* It is the expected fraction of zeros in stable BF, when data is unbounded. Using Theorem 4.2.1 probability of values in bucket being set to zero is constant *i.e.*

$$Pr(ASBF_N = 0) = \sum_{l=Max}^{N-1} [Pr(ASBF_N = 0|A_l)Pr(A_l)] + Pr(ASBF_N = 0|A_N)Pr(A_N) \tag{4.10}$$

Let H be the number of cells that are set to Max and L be the number of cells that are decremented to zero after certain number of iterations (from Theorem 1), then expected number of zeros in FP-SBF *i.e.* stable point property of FP-SBF, is given by (Z_{SP}):

$$Z_{SP} = \left(\frac{1}{1 + \frac{1}{L(\frac{1}{H} - \frac{1}{m})}} \right)^{Max} \tag{4.11}$$

The optimization in decrement operation will help to achieve stable point in FP-SBF in less number of iterations with less computational complexity, which will further help to detect parameters like false positives and false negatives at earlier stage.

4.2.5 False Positives Analysis

Theorem 4.2.2. *When FP-SBF reaches a stable point, the False Positives (FPs) are given by:*

$$FP = F_{FP-SBF} - RF_{FPC}$$

where F_{FP-SBF} is error due to collision in buckets of FP-SBF and RF_{FPC} denotes the reduction factor in FPs due to fingerprint cells.

Proof. FPs are generated when distinct element in the stream is wrongly reported as duplicate. FPs are directly dependent on number of zeros at particular point in FP-SBF, *i.e.*, when stable point is reached FPs can be estimated. Change in fingerprint cell of each bucket helps to improve FPs by providing a second level check. In case of collision in the buckets, F_{FP-SBF} is dependent on the number zeros in the filter. More the number of zeros, less the collisions and less FPs. When stable point is reached, number of zeros become constant and after certain iterations, H buckets are set to Max and L are decremented to zero; F_{FP-SBF} is given:

$$F_{FP-SBF} = \Big(\frac{1}{1+\frac{1}{L(\frac{1}{H}-\frac{1}{m})}}\Big)^{Max} \tag{4.12}$$

RF_{FPC} is the reduction factor in detection procedure due to fingerprint cells when all buckets corresponding to hash indexes are high. It acts like a second level check on these f bits. Since the possibility of FPs is one out of 2^f cases in fingerprint cells, it helps in reducing the FPs by a fixed factor. Assuming that insertion operation is performed I times on k indexes (equal to number of hash functions) and fingerprint cells are updated by *xor* operation; RF_{FPC}, the probability of not having collision in f bits in m FPC's is:

$$RF_{FPC} = \Big[\Big(\frac{2^f-1}{2^f}\Big)\Big(\frac{1}{m}\Big)\binom{I}{1}\Big(1-\frac{1}{f}\Big)^{I-1}\Big]^k \tag{4.13}$$

From Eq. (4.12) and (4.13), FPs in FP-SBF are given by:

$$FPs = \Big(\frac{1}{1+\frac{1}{L(\frac{1}{H}-\frac{1}{m})}}\Big)^{Max} - \Big[\Big(\frac{2^f-1}{2^f}\Big)\Big(\frac{1}{m}\Big)\binom{I}{1}\Big(1-\frac{1}{f}\Big)^{I-1}\Big]^k \tag{4.14}$$

Thus, use of fingerprint cell helps to reduce the FPs by a significant factor and improve the accuracy of the duplicate detection system.

□

4.2.6 False Negatives Analysis

Theorem 4.2.3. *At stable point False Negatives (FNs) for FP-SBF are given by:*

$$FNs = FR_{FP-SBF} + E_{FPC}$$

where F_{FP-SBF} is error due to deletion operation on buckets of FP-SBF and E_{FPC} denotes the error in fingerprint cell due to the collision of H_{fp} function.

Proof. A FN in duplicate detection on streamed data occurs when a duplicate element (x_i) is wrongly reported as distinct element. This happens during decrement operation when some buckets associated with hashed indexes of x_i are decremented to zero, before appearing in the stream or there is a mismatch in fingerprint cells when all the buckets have high value corresponding to x_i.

Suppose x_i appears second time in the stream after δ iterations and $h_{j=1}^{j=k}(x_i)$ denotes the corresponding hash indexes and p_{ij} is probability that a particular cell C_j is set to Max. In δ iterations, some bucket from $h_{j=1}^{j=k}(x_i)$ are reduced to zero. The probability of error due to buckets resetting (FR_{FP-SBF}) is same as in Eq.(4.8); given by:

$$FR_{FP-SBF} = 1 - \prod_{j=1}^{k}(1 - Pr(ASBF_\delta = 0)) \tag{4.15}$$

and probability that after δ iterations a particular bit of FP-SBF is zero, *i.e.*, $Pr(ASBF_\delta = 0)$ is given by:

$$Pr(ASBF_\delta = 0) = \sum_{l=Max}^{\delta-1}[Pr(ASBF_N = 0|A_l)Pr(A_l)] + Pr(ASBF_N = 0|A_N)Pr(A_N) \tag{4.16}$$

So FN is function of δ and p_{ij} given by:

$$FR_{FP-SBF} = 1 - \prod_{j=1}^{k}(1 - Pr(\delta_j, p_{ij})) \tag{4.17}$$

E_{FPC}, chance of false negatives due to error in H_{fp} hash function in fingerprint cell is due to the collision in changing bits because of *xor* operation in δ iterations. In m size array, selected k FPCs are checked and mismatch in any of them leads to failure in detection process, *i.e.*, a duplicate is reported as distinct element. After f iterations, chances that erroneously reported elements are 1 out of 2^f for δ iterations using k hash function is given by:

$$E_{FPC} = \left[\left(\frac{1}{2^f}\right)\left(\frac{1}{m}\right)\binom{\delta}{1}\left(1 - \frac{1}{f}\right)^{\delta-1}\right]^k \tag{4.18}$$

FNs for FP-SBF is given by:

$$FNs = \left(1 - \prod_{j=1}^{k}(1 - Pr(ASBF_{\delta} = 0))\right) + \left[\left(\frac{1}{2^f}\right)\left(\frac{1}{m}\right)\binom{\delta}{1}\left(1 - \frac{1}{f}\right)^{\delta-1}\right]^k \tag{4.19}$$

Use of FPCs shows great improvement in decreasing the FPs of FP-SBF in all cases.

□

Theorem 4.2.4. *For given inputs k, Max, f, FPs and H_{fp}; processing each data item of the stream requires $O(k+1)$ time which is independent of the size of BF and the in-coming data stream.*

Proof. In duplicate detection, primary goal is minimization of error rates while using constant space, and time complexity in detection process should be independent of the nature and size of data stream, *i.e.*, there should be constant processing time for each element. First step in duplicate detection is to check whether an element is seen previously in the stream or not. For this first k hash functions are computed for buckets and then one hash function is calculated for fingerprint cell. For given parameters k, Max, f, FPs with constant values, there is no effect of element and detection process is also independent of the size of BF(m).

From Algorithm 4.1 and analysis performed above it is concluded that processing time in duplicate detection is only dependent on number of hash functions, *i.e.*, $O(k+1)$.

□

Experiment and analysis of FP-SBF are discussed in detail in Section 4.3.

4.3 Observation and Analysis

Following sections elaborates the result evaluation cum comparative analysis for the proposed BF.

4.3.1 Theoretical Analysis

The theoretical analysis has been provided in following section for some important parameters, *i.e.*, Max, H, L, m, ξ, FN and FP; where H is number of cells set to Max in insertion operation, *i.e.*, (equal to number of hash functions), L denotes the number of cells selected for decrement operation and m is fixed amount of memory used for BF.

FPs can be bounded according to user specified requirements. Since the FNs in the fixed amount of memory are depended on deletion operation they cannot be bounded in specified limits. Two parameters, desired FP rate and size of BF (m) are taken as input from user and other parameters like Max,H,L are selected in such a way that FNs are minimimal.The parameter ξ is also user defined and helps to control the frequency of deletion operation.

For user defined FPs, m and for constant values of Max and H; L is defined as:

$$L = \left(\frac{1}{\left(\frac{1}{\left(1-FPs^{1/H}\right)^{1/Max}} - 1 \right) \left(1/H - 1/m \right)} \right) \tag{4.20}$$

Eq. (4.20) helps to find value of L *i.e.* number of cells selected for decrement operation. From the value of L (calculated in Eq. (4.20)) p_d, the probability of selecting cells for decrement and P_0, probability that after certain decrement operations value of cell is zero, can be

derived.

Parameter H is equal to the number of hash functions used (k). $E(FN)$ denotes the expected number of false negatives in the stream. Optimal value of H should be selected to minimize the FNs. With $\tilde{N}$ as the number of false negatives in a stream of N elements, $E(FN)$ is:

$$E(FN) = \sum_{i=1}^{\tilde{N}} \Big(Pr(FNR_i) \Big) \tag{4.21}$$

$$E(FN) = \sum_{i=1}^{\tilde{N}} \left(\left(1 - \prod_{j=1}^{k} (1 - Pr(ASBF_\delta = 0)) \right) + \left[\left(\frac{1}{2^f} \right) \left(\frac{1}{m} \right) \binom{\delta}{1} \left(1 - \frac{1}{f} \right)^{\delta - 1} \right]^k \right) \tag{4.22}$$

The value of Max is depended on size of input data and number of hash functions used. Optimal value of Max can be derived from Eq. (4.22) by minimizing FNs. For efficient memory utilization Max should be set $2^c - 1$. The remaining bits that are not allocated to counter are used as fingerprint bits. For fixed amount of cells when value of Max is increased, effectiveness of fingerprint cell is reduced.

The optimization in deletion process is controlled by user defined parameter ξ and $Rand(\xi)$ function is used to set the frequency of deletion operation; larger the value of ξ more frequently the deletion operation is performed and vise versa.

4.3.2 Experiment Evaluation

To check the accuracy of proposed FP-SBF a data set of 100k elements with 70% distinct entires and 30% duplicate entires has been generated using $R-studio$. All the experiments have been performed on $i7-3612QM$ CPU @ 2.10 GHz with 8 GB of RAM. To maintain the uniformity in the results $CityHash$ 64 bit library [177] is used to compute the hash functions for PDS. Comparative analysis has been performed between stable BF [92], Reservoir Sampling based BF (RSBF) [115] and FP-SBF. Various experiments conducted indicate that

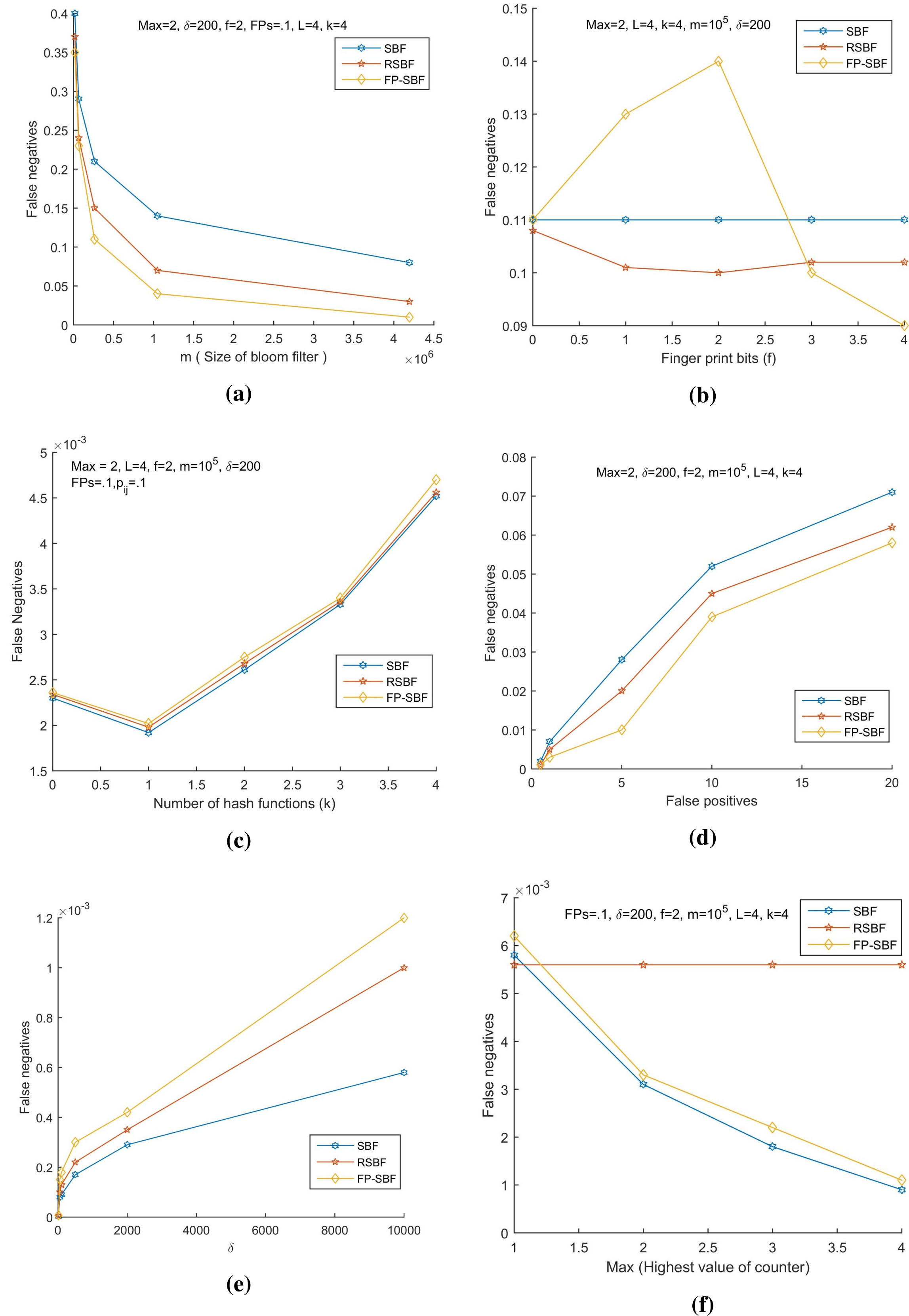

Figure 4.5: Variation in false negatives with different parameters

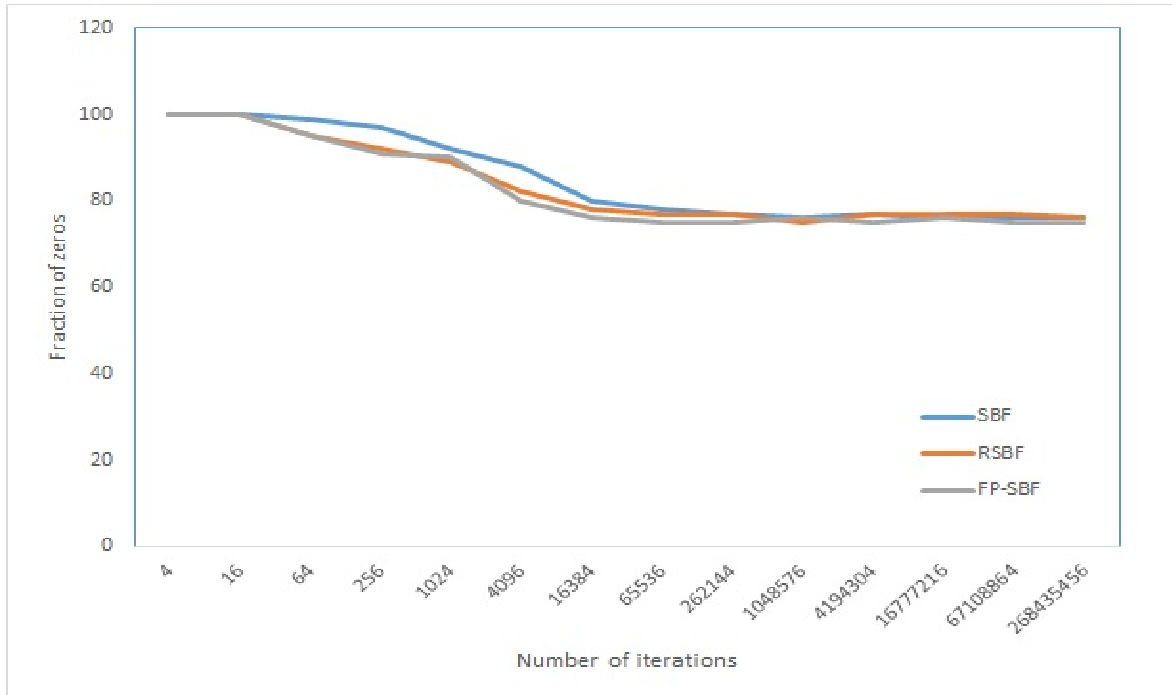

Figure 4.6: Stable point

the proposed approach outperforms stable BF and RSBF; both the approaches used for duplicate detection in the streamed data using BFs. In the following experiments SBF stands for Stable Bloom Filter.

Fig. 4.5 shows the impact on false negatives with variations in size of BF(m), size of counter in bucket(Max), number of fingerprint bits(f), number of hash functions(k) and false positives(FP). All results have been evaluated using fixed values for the required parameters.

As shown in fig. 4.5(a), false negatives decrease with the increase in size of BF; more the space, less the effect of deletion operation and less the false negatives. Fig. 4.5(b) indicates the change in false negatives with respect to fingerprint bits in FP-SBF; since the result of stable BF and RSBF are not effected by value of f so false negatives remain the same. In FP-SBF first false negative increases when f is small; for $f = 3$, accuracy is same as in the RSBF but as the value of f increases, acurracy of the proposed scheme increases. Fig. 4.5(c) indicates that as the number of hash functions increase more positions need to be checked, increasing the chances of FNs. With the change in predefined FPs, the change

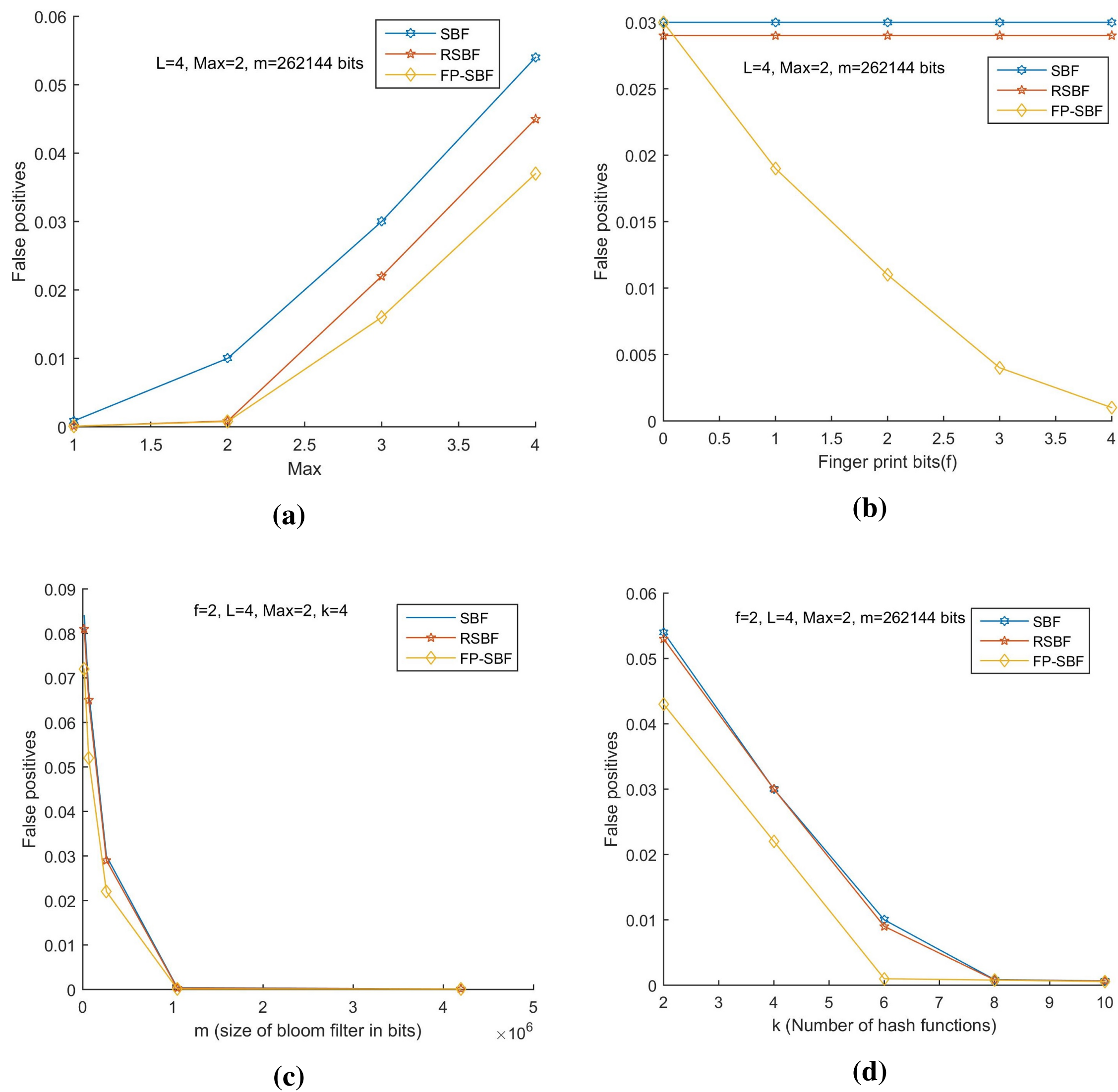

Figure 4.7: Variation in false positives with different parameters

in false negatives are indicated by fig. 4.5(d) showing that more the FPs, more error is allowed; less is the deletion operation less will be the number of FNs. δ denotes the average number of iterations between two similar items in the stream, effect of δ on FN are shown in fig. 4.5(e) which clearly shows that as the value of δ increases duplicates are detected after more iterations hence more deletion operations are performed between two similar elements so chances are high that a duplicate is detected as distinct element. Fig. 4.5(f) shows the change in FN with respect to bucket size Max; larger is the bucket size, more are the operations required to reset it to zero and less will be the false negatives.

Fig. 4.6 depicts the number of iterations required to achieve stable point, *i.e.*, constant fraction of zeros in the BF. Initially, the filter is empty so number of zero is 100%. As the number of iterations increase, more insertion operations are performed and number of zeros are reduced; after some time deletion operation is also performed and later number of zeros become constant. Both RSBF and FP-SBF perform efficiently in achieving the stable point but FP-SBF has the advantage of controlling the deletion operation by using optimized deletion process which reduces the number of iterations required to reach stable point.

Fig. 4.7 provides the false positives analysis which shows that with the use of finger print bits false positives are drastically reduced in the FP-SBF as compared to RSBF and stable BF. In fig. 4.7(a) false positives increase with the size of counter in bucket, but the use of finger print bits in the FP-SBF helps to reduce it to a great extent. Fig. 4.7(b) shows that more the number of fingerprint bits used, less will be the number of false positives in duplicate detection task. Fig. 4.7(c) indicates that large the size of BF used, less will be false positives. Number of hash functions are always a critical factor in BF, fig. 4.7(d) shows that more the number of hash functions used, less will the false positives.

4.3.3 Application Domains

Duplicate item identification is a common problem faced by social networking sites like Twitter, Facebook, Instagram, *etc.* where multiple copies of same event (tweet or post) are

generated for same input by multiple users, which keep on appearing continuously from different sources on the user's screen. In such scenarios, the primary need is to identify duplicate events and group them together to improve the user's experience [112]. Another important event related to duplicate detection is URL crawling. Search engines regularly crawl the Web to enlarge their collections of Web pages. While scanning a URL, search engine's task is to identify the new web pages and add them to its repository. So, the basic task in this scenario is comparing each scanned URL with all existing URLs in its database to identify duplicate URLs [179, 180]. In network monitoring applications, selecting distinct IP addresses is a task associated with duplicate detection. In networks, a particular server is checked for unique hits. This analytics facilitates in understanding the pattern of traffic which helps in efficient allocation of network resources [92]. Web advertising is easy and most effective way to publicize a product where advertisers pay web site publishers for number of clicks on their advertisements. Fake clicks may be generated (by using scripts) to increase the profit of the publisher. To detect the duplicate users in clicks is thus associated with duplicate detection task [113].

4.3.4 Discussion

FP-SBF uses fingerprint bits to improve the accuracy of the task and there are cases when false negatives increase with the reduction in false positives.

- Few use cases are discussed below to analyse the output in various scenarios. Parameter tunning in FP-SBF which can decrease the false positives to minimum level is:

$$MFP((Max > 3), (f > 3), (m > 5 \times 10^5), (k > 5)) \tag{4.23}$$

  Where FP-SBF can be used successfully include:

  - IoT data streams, where data is coming from number of sensors, FP-SBF can be

used to check the identity of the sensors.

- To detect first time user in real time data streams of online shopping platforms for promotional strategies.
- To detect the active users in social networking websites like twitter, facebook, *etc.*, from the number of posts registered in the given time span t.

- Parameter tunning in FP-SBF which can decrease the false negatives to minimum level is:

$$MFN((\delta < 1000), (Max > 4), (f > 4), (m > 4 \times 10^6), (2 < k < 5)) \tag{4.24}$$

Coming chapter (Chapter 5) discusses a technique proposed for Spam Detection in Social IoT through the ensemble of PDS along with the experiment analysis.

Chapter 5

Eb-SDF: Ensemble based Spam Detection Framework

In this chapter an Ensemble based Spam Detection Framework (Eb-SDF) has been proposed which tries to identify majority of spam tweets (on Twitter dataset) by using different classifiers at varied levels and update the databases efficiently.

5.1 Spam Detection

Twitter, one of the most popular microblogging based social network with 271 million monthly active users, was established in 2006. It allows 'tweets' comprising of 140 characters textual information to be shared about various topics. As per the available statistics, around 500 million tweets are posted per day on this social media platform [181]. Along with sharing tweet, twitter also provide some more features to users: allowing tweets posted by one user to be accessed by group of users, re-tweeting by sharing tweet of some other users, *etc*. Hashtags are the keywords that may be used to associate a tweet with a particular topic and used for trending topics.

Popularity of Tweeter rests on its two basic foundations: *dynamic platform and no restriction on number of tweets posted*. Twitter's active user base and its high click through

rate makes it an attractive forum for like minded people and spammers. People with varied opinions feel comfortable by expressing or sharing their views on the trending topics [181].

Twitter allows users and applications to post messages on user's behalf using Twitter API. These APIs can be efficiently used as the channel for sensors to communicate which will lead to quick deployment of IoT application. Twitter will help the devices in one IoT to communicate with each other and other devices which are in different networks along with human beings, leading to increase in the power of the entire IoT [182]. But with such a wide platform of users and their data, Twitter attracts the spammers or malicious users too, who use this media to promote their malicious activities or try to mislead followers and affect sentiments of particular social groups [145, 183]. Detecting spam accounts is an important issue which definitely needs lot of attention as nature of spammers and amount of damage they can cause is a point of major concern.

Some common approaches followed by spammers on twitter are: sharing malicious links (*e.g.* malware sites), aggressively following behavior (affecting the mass follower of a user), cyber bulling (abusive and unwanted messages to users), creating multiple accounts (either manually or using social bots), posting repeatedly on trending topics to grab attention, posting duplicate updates repeatedly, *etc.* [150, 184]. As the spam detection methodologies evolve, the spammers too update their techniques to meet their primary goal of spreading malicious content [185].

Twitter management group has taken significantly measures to stop spammer by suspending accounts of users which are identified as part of such malicious activity. Twitter allows its user to report spammers to the official @spam account. To identify spam automatically various machine learning techniques have been used where multilevel checks are performed to stop such irrelevant inputs [186].

Twitter produces huge volumes of data rapidly; handling such voluminous data and scanning for spam is complex and time consuming task. The main concern is to detect the spammer as soon as possible in streaming data [184]. As the size of incoming data increases,

analysis of such massive data sets requires fast computation techniques which can respond quickly. Approximation and probabilistic algorithms are effective tools which can be used to reduce the computation time and space complexity of analytics on large datasets.

5.2 Spam Detection in Social IoT

Consider a data set $D = \{x_1, x_2, ..., x_n\}$ of n instances with set of attributes $A = \{a_1, a_2, ..., a_t\}$ and their corresponding labels $L = \{l_1, l_2, ..., l_n\}$, where $\forall_{i=1}(l_i \in \{Spam, Ham\})$. The task is to build an ensemble based framework consisting of k classifiers ($C = \{C_1, C_2, ..., C_k\}$) which map data into appropriate labels, where l_j^F denotes the final output label for instance $x_j \in D$. Optimization objective of the proposed framework is to use Hybrid Sampling (HS) technique to efficiently sample the huge dataset by:

i. Using advance techniques in classification framework to minimize deviation ($\delta = l_i - l_j^F$) in result,

ii. Minimizing the computational complexity (ζ) for optimized output.

iii. Minimizing delay (τ) in decision making.

iv. Adapt dynamically (φ) according to changing nature of data stream.

Mathematically, the objective function of ensemble based classifier ($(E_C)^O$) can be defined as:

$$Data^{[D,A,L]}\{x_i\} \xmapsto[Input]{HS} [E_C] \underset{Dynamic\ Updates^{(\varphi)}}{\overset{Classifier^{(\delta,\tau,\zeta)}}{\rightleftharpoons}} [l_{x_i}^F] \begin{cases} Ham \\ \\ Spam \end{cases} \tag{5.1}$$

$$l_{x_i}^F \leftarrow (\max_{l_k}[\forall_{i=1}^k(\phi(l_k, w_k) > thres)]$$

where l_k is output label for instance x_j generated by the classifier C_k, w_k is the weight assigned to the classifier according to its participation in framework and $\phi(l, w)$ is decision

function based on weighted output. If output by function $\phi()$ exceeds the defined output then label having maximum votes in k classifier is assigned to final output $l_{x_i}^F$.

Spam detection in the high velocity streaming data is a tedious task especially when large number of attributes are associated with it. Using single model for detection will not be an effective mechanism as the attackers keep on changing their attack patterns dynamically. This proposed semi-supervised technique for spam detection in Twitter using ensemble based framework which will help in future integration of social media with IoT. Major focus of the Eb-SDF is to minimize error in classifier's output by optimizing computational effort and reducing delay in decision making. A hybrid sampling model has been proposed which trains the important instances using probabilistic approaches (Section 5.2.1). Quotient filter, an approximate membership query PDS, has been used for fast analysis and locality sensitive hashing have been used for quick and efficient similarity search in huge data base (Section 5.3). Use of these PDS reduce the computation time, storage requirement and search time significantly. Framework has been made adaptive by using Markov based updation model to update the databases (Section 5.3.5). Fig. 5.1 presents the overview of Eb-SDF.

5.2.1 Hybrid Sampling

Two most frequently used techniques in ensemble based models to generate dataset for classifiers are *bagging* and *boosting*. In bagging, new datasets are generated from original data by drawing samples at random. Boosting uses adaptive re-sampling (re-weighted) approach such that data-points which were misclassified are given more weight. One of the major issues in boosting is that maintaining weight for each instance, especially in large datasets, is a tedious task.

In the Eb-SDF hybrid of bagging and boosting has been used where D data set is divided into b buckets, *i.e.*, $\left((_{i=1}^{b} d_i \subset D) \text{ and } (d_1 \cup d_2 \cup ... \cup d_b = D)\right)$. In initial iterations, d_i is provided as input to all classifiers and probability Pr_{d_i} is calculated on the basis of classifier's

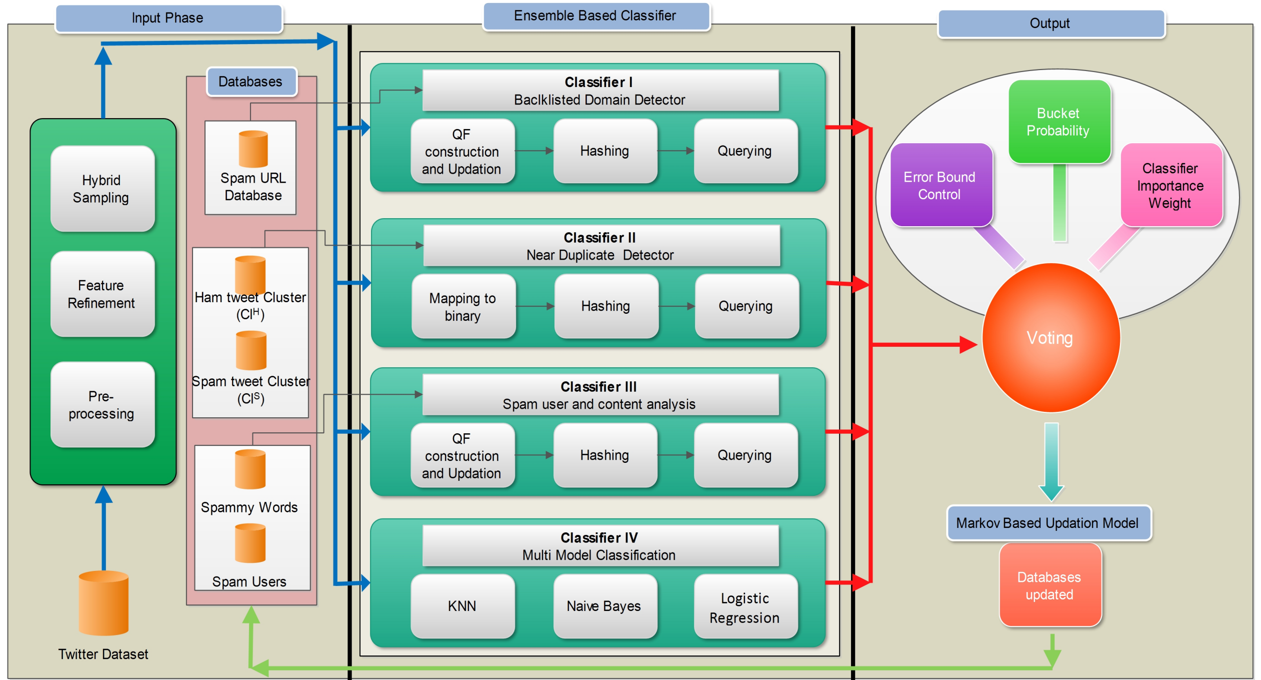

Figure 5.1: Framework for Ensemble based Spam Detection (Eb-SBF)

output. After few iterations, when output from each bucket becomes stable, dataset of size s is randomly drawn from all buckets based on the Pr_{d_i}. For each iteration Pr_{d_i} is calculated as:

$$Pr_{d_i}^{j} \leftarrow \frac{Pr_{d_i}^{j-1} e^{-f\gamma()}}{\sum_{z=1}^{k} Pr_{d_i}^{j-1} e^{-\forall(z) f\gamma(z)}} \tag{5.2}$$

$$where\ f\gamma() = [if(l_{x_j}^{o} == l_{x_j})] \begin{cases} TRUE & +1 \\ FALSE & -1 \end{cases} \tag{5.3}$$

Here $Pr_{d_i}^{j-1}$ is the probability associated with previous iterations, l_{x_j} is original label of data and $l_{x_j}^{o}$ is output label for instance x_j, given by:

$$l_j^{o} \leftarrow \alpha_{C_i} C_i(x_j) \tag{5.4}$$

Here α_{C_i} is importance weight associated with each classifier, given by:

$$\alpha_i = \frac{1}{2} ln(\frac{(1-\varepsilon_i)}{\varepsilon_i}) \tag{5.5}$$

ε_i is the error rate associated with each classifier, given by:

$$\varepsilon_i = \forall(y \in d_i) Pr_{d_i}^{j-1}(C_i(l_y \neq l_y^o)) \tag{5.6}$$

Initial probability of drawing sample is set to $\left(Pr_{d_i}^1 = \frac{1}{b}\right)$ and buckets with higher probability signifies that they need more iterations to improve the model.

5.3 Classifiers Considered in Eb-SDF

Four classifiers have been used to efficiently label a tweet as a spam or Ham in the proposed semi-supervised approach. A dataset D is sampled and passed to all four classifier; based on the output of each classifier, final decision is made by majority voting algorithm (Algorithm 5.1).

5.3.1 Classifier I: Blacklisted Domains Detector or URL checker

One of the basic approaches used by malicious users to promote their activities is by sharing the URLs. List of URLs classified as spam URLs are taken as input from the database and if any new URL is identified as *spam* by any classifier, database is updated accordingly. To check whether URL shared in tweet is valid or not, membership query is performed through the use of quotient filter. If the tweet being considered has no URL then it is marked as *ham* by classifier-I. Steps followed in classifier I are:

Blacklisted domains database creation

- All the spam URLs are stored in QF of blacklisted domains QF^{BD} using quotienting

hashing technique. Number of buckets (size) of quotient filter for m inputs is:

$$Size(QF^{BD}) \rightarrow (2^q = \lceil m \rceil) \tag{5.7}$$

- All URLs are hashed to equivalent p bits binary input denoted as fingerprint of URL(fp_{URL}) using a function $\varpi()$, *i.e.*

$$\varpi(URL_i) \mapsto (fp_{URL_i} \in \{0,1\}) \tag{5.8}$$

- To store fp_{URL_i} in QF^{BD}, reminder bits r are selected to calculate f_r and f_q, where f_q provides the bucket number in QF and f_r contains the content stored in bucket. Hashing process used is:

$$\begin{aligned} f_r^{URL_i} &= ((fp_{URL_i}) mod\ 2^r) \\ f_q^{URL_i} &= \lfloor (fp_{URL_i})/2^r \rfloor \\ QF^{BD}[f_q^{URL_i}] &\leftarrow f_r^{URL_i} \end{aligned} \tag{5.9}$$

- Before performing insertion operation in QF_{BD} for URL_i, $f_q^{URL_i}$ is calculated and start of *run* corresponding to $f_q^{URL_i}$ is identified. In QF, suitable position (sp) to insert the reminder is at the end of *run* of bucket and insertion is performed.

Blacklisted domains database updation process

- In classifier I, labeling is based on the blocked URLs available in the database, which needs constant updation.

- Unregistered URLs in the tweets which are marked as Ham by classifier-I but identified as *spam* in the final label are marked as tweets (tw_i) which need attention.

- In updation block, URL_{tw_i} is processed for inspection procedure. If it is identified as malicious then it is added to locally maintained quotient filter (QF_U^{BD}), which is

merged with main database (QF^{BD}) once the threshold is crossed and all databases are updated.

$$QF^{BD} \leftarrow (QF^{BD} \oplus QF_U^{BD}) \tag{5.10}$$
$$where\ \oplus\ indicates\ merging\ of\ QF^{BD}\ and\ QF_U^{BD}$$

Label identification in Classifier I: For tweet under observation say tw_o containing URL content from blacklisted domain, detector classifier tries to identify labels in QF as follows:

- In first step URL_{t_o} is mapped to $fp_{URL_{t_o}}$ using Eq.(5.8).
- Hashing is performed as mentioned in Eq.(5.9) to compute f_q and f_r associated with $fp_{URL_{t_o}}$.
- Following conditions are checked to label a tweet as *ham* or *spam*:

$$(f_q^{tw_o} \in QF^{BD}) \rightarrow \begin{cases} False & l_1^{tw_o} = 0 \\ True & \theta() \end{cases}$$
$$\theta() \rightarrow (f_r^{tw_o} \in\ cluster\ of\ f_q^{tw_o}) \begin{cases} False & l_1^{tw_o} = 0 \\ True & l_1^{tw_o} = 1 \end{cases} \tag{5.11}$$
$$where (0 = Ham, 1 = Spam)$$

Here $l_1^{tw_o}$ indicates the label given by classifier-I to tweet under observation tw_o. As mentioned earlier, if their is no match found for *spam* URL or tweet contain only text data then classifier marks it as *ham*.

5.3.2 Classifier II: Near Duplicate Detector

In second classifier, labelling of tweets is done by analysing the similarity of the tweet with predefined clusters. Two clusters are generated from the labelled database: Ham tweets and spam tweets represented by Cl_H and Cl_S respectively. Every tweet tw_o is checked for similarity with both the clusters, *i.e.*, Cl_H and Cl_S. Based upon the output achieved, tweet is assigned the label of cluster to which it is significantly close. To make near duplicate detection task fast and accurate LSH with hamming distance is applied in classifier-II.

When a tweet tw_o comes to classifier-II, steps followed are:

- Mapping of tw_o with d dimensions (attributes) to hamming space of p bits is done by $\varpi()$

$$\varphi(tw_o \in \Re^d) \mapsto (tw_o^h \in \{0,1\}^p) \tag{5.12}$$

  Using $\varphi()$ function dimensionality reduction is performed by representing each attribute of a tweet as a binary string of p bits.

- Randomized approach $(Random)$ is used to calculate the similarity of set R_S and R_H belonging to cluster C_s and C_h respectively. Random() function gives r randomly selected tweets $(tw_{1...r})$ from selected cluster. For comparison, these sets are also hashed to their binary equivalent using Eq. (5.12).

$$(Random)_{i=1}^{r}(tw_i \in Cl_{S/H}) \rightarrow R_{S/H} \mapsto (R_{S/H}^h \in \{0,1\}^p) \tag{5.13}$$

- Tweet under observation tw_o^h is compared with R_S^h and R_H^h using LSH. Hash functions for comparison on hamming space are constructed by selecting k bits randomly from the binary string of p bits of each tweet. ℓ hash functions are calculated, given by:

$$\forall_{i=1}^{\ell}(H_i \leftarrow Random_k(tw_o)^h) \tag{5.14}$$

  Each hash function represents a single bit or set of bits, selected at random from ham-

ming space representation of p bits.

- Final label is assigned by performing comparison of tw_o with R_S and R_H on the basis of hash functions as follows:

$$l_2^{tw_o} \leftarrow Max[(\forall H_i)(tw_o == (tw_j \in R_{H/S}))] \tag{5.15}$$

where $l_2^{tw_o}$ indicates the label given by classifier-II to tweet under observation (tw_o) on the basis of nearest cluster which contains maximum number of matches with tw_o.

Use of LSH reduces the search space by a huge factor and improves the accuracy of the system significantly. Instead of comparing each binary string with other, only hash values are compared and element having maximum similarity in hash values are considered as nearest neighbor.

5.3.3 Classifier III: Reliable Ham Tweet Detector

Third classifier uses content analysis, *i.e.*, decision is taken on the basis of who posted the tweet and text in it. Classifier-III checks for spammy words in the tweet and then validates the authenticity of the user, *i.e.*, it checks for spam user instead of ham user. Two databases are maintained for this classifier: first is of spam words and second of users who are marked as *spam*. Using QF, results are achieved efficiently for the 'element not belonging to the dataset' without any error. Steps followed in classifier-III are:

- Database of spam words and spam users is maintained using two quotient filter *i.e.* QF^{SW} and QF^{SU} using Eqs. (5.7-5.9).

- To assign label to tw_o, user associated with tweet ($uid(tw_o)$) is checked for authenti-

cation in QF^{SU}. Query process in QF is same as performed in Eq (5.11).

$$\text{If}(uid(tw_o) \in QF^{SU}) \begin{cases} TRUE & (l_3^{tw_o} = 1) \\ FALSE & (l_3^{tw_o} = 0) \end{cases} \tag{5.16}$$

- If the query in above mentioned step returns 0, *i.e.*, $uid(tw_o)$ is a Ham user then second level check on spam words is performed. A set of tokens $Tkn(tw_o)$ is generated after initial refinements like stop word removal, delimiters *etc.*, *i.e.*:

$$Tkn(tw_o) \leftarrow REF(tw_o) \tag{5.17}$$

- Words in $Tkn(tw_o)$ are checked with QF^{SW}.

$$Tkn(tw_o)^* \leftarrow \forall w_i | (w_i \in QF^{SW}) \tag{5.18}$$

- For words present in spam list $tf-idf$ is calculated using function $\tau\iota()$. Final decision is made on the basis of defined range of threshold (θ^*) for spam words, described as follows:

$$\theta = \sum \tau\iota(\forall w_i | w_i \in Tkn(tw_o)^*) \tag{5.19}$$

$$\text{If}(\theta > \theta^*) \begin{cases} TRUE & (l_3^{tw_o} = 1) \\ FALSE & (l_3^{tw_o} = 0) \end{cases} \tag{5.20}$$

where $l_3^{tw_o}$ indicates the label given by classifier-III to tweet under observation (tw_o) using content analysis model based on QF.

5.3.4 Classifier IV: Multi Model Classifier

Accuracy of classifiers(I-III) depend on the availability of database. When Eb-SDF is in initial state, accuracy of system is not upto the mark. To avoid such kind of problems, standard classifier algorithms have been used in the final stage of our ensemble based model. It helps in updating the databases and cross checks the output of other classifiers.

To improve the accuracy of the classifier, three different models are used and final decision for the label of tweet is assigned on the basis of majority voting algorithm. Three in-build classifiers used are: *K-Nearest Neighbor, Naive Bayes and Logistic Regression*. All selected attributes in feature refinement process are given as input to all the three models, a tweet under observation tw_o in classifier IV is labeled as follows:

- tw_o is given as input to all three models refereed as M^{KNN}, M^{NB} and M^{LR}.
- The final label $(l_4^{tw_o})$ of classifier-IV is based on the majority voting, given by:

$$l_4^{tw_o} \leftarrow \min \begin{cases} M^{KNN}(tw_o) \\ M^{NB}(tw_o) \\ M^{LR}(tw_o) \end{cases} \tag{5.21}$$

  At least two models should have same output to label a tweet as *spam* or *ham*.

- If tweet tw_o is marked same by all three models in classifier-IV, then it is assumed that the result are accurate since the tweet is unambiguous and tweet is marked as 'feedback tweet' (ft).

$$\text{If}(\forall Mi|(l^{Mi} = l^{Mj}))then\ \{(tw_o \leftarrow ft)\} \tag{5.22}$$

$$where\ l^{Mi}\ =\ label\ by\ i^{th}\ Model$$

Input : Ensemble Based System (D, C_k, A, L)
Output: Insert $x_i \in S$ for $t_i \in T$ in $ATBF_i$ array

```
1  Step 1: Data Sampling
2  Divide D into {d_1, d_2, ..., d_b} b buckets
3  Assign probability to each bucket Pr_{d_i} (Using Eq. 8)
4  Step 2: Ensemble Model
5  for (∀tw_o ∈ d_i) do
6      for (∀i|i ∈ C_k) do
7          Set l_i^{tw_o} = 0
8      end
9      Classifier-I
10     Construct QF^{BD} (Using Eq. (5.7-5.9))
11     if (URL(tw_o) ∈ QF^{BD}(Using Eq. (5.11)) then
12         Set l_1^{tw_o} = 1
13     end
14     Classifier-II Map tw_o into binary (Using Eq. (5.12))
15     Select random tweet sets R_H and R_S ( Eq. (5.13))
16     if (tw_o == (∀t_j ∈ R_{H/S}))(Using Eq. (5.14)) then
17         Count++
18         if (Count > threshold) (Using Eq. (5.15)) then
19             Set l_2^{tw_o} = 1
20         end
21     end
22     Classifier-III Construct QF^{SU} and QF^{SW} (Using Eq. (5.7-5.9))
23     if uid(tw_o) ∈ QF^{SU} (Eq. (5.16)) then
24         Set l_3^{tw_o} = 1
25     else
26         Make set of tokens(Tkn(tw_o)) (Eq. (5.17))
27         for (∀w ∈ Tkn)|w ∈ QF^{SW} do
28             θ = τι(w) (Using Eq. (5.18-5.19))
29             if θ > θ* then
30                 Set l_3^{tw_o} = 1
31             end
32         end
33     end
34     Classifier-IV
35     for (∀M^i) do
36         Calculate l^{M^i} ← M^i(Using Eq. (5.21))
37     end
38     if max*(l^{M^i} = 1) then
39         Set l_4^{tw_o} = 1
40     end
41     if ∀l*|(l^{M^i} = 1) then
42         Set tw_o = ft
43     end
44 end
```

Algorithm 5.1: EbSDF: PDS Based Classifiers in Ensemble Framework

5.3.5 Model Updation

A dynamically updating model is required which is adaptive to the changing input patterns. The model updation phase updates the database and the entire framework which include tasks like final voting decision, updated probabilities associated with each bucket for effective sampling in next iteration, updated weights of the classifiers, *etc.* The functionality of updation model is explained below (Algorithm 5.2):

i. Error bound control: ε_{C_i} is error rate associated with each classifier, given by:

$$\varepsilon_{C_i} = \sum_{j=1}^{b} \forall(y \in d_j)\{Pr_{d_j}(C_i)[\delta(C_i)]\} \tag{5.23}$$

where $\delta(C_i) = \#(l_y \neq l_y^o)$ provide count of instances where output of classifier does not match with the true value. $Pr_{d_j}(C_i)$ denotes the weight of the sample drawn for d_j bucket using C_i classifier. Initial probability of drawing sample $\left(Pr_{d_i}^1 = \frac{1}{b}\right)$ and buckets with higher probability means that they need more iteration for better model building.

ii. Weight Assignment: In the proposed model two type of weights are assigned, first is weight of each bucket *w.r.t.* each classifier; it is the probability of drawing samples from bucket (Pr_{bucket}) and second is importance of classifier calculated on the basis of overall accuracy of classifier α_{C_i}.

(Pr_{bucket}) for a bucket d_i is given by:

$$Pr_{d_i}^{j} \leftarrow \frac{Pr_{d_i}^{j-1} e^{-\{\forall(y \in d_i)\gamma(y)\}}}{Z} \tag{5.24}$$

where $Pr_{d_i}^{j-1}$ is the probability associated with previous iteration and Z is the normalizing factor, calculated as:

$$Z = \sum_{z=1}^{k} Pr_{d_i}^{j-1} e^{-\{\forall(y \in d_i)\gamma(y)\}} \tag{5.25}$$

$\gamma(y)$ function compares each instance output (l_y^o) processed by the classifier with true value

of instance (l_y) and then provides output as follows:

$$\gamma(y) = \begin{cases} +\alpha_{C_i}, & \text{if } (l_y^o == l_y) \\ -\alpha_{C_i}, & \text{otherwise} \end{cases} \tag{5.26}$$

where α_{C_i} is importance weight associated with each classifier, given by:

$$\alpha_{C_i} = \frac{1}{2} ln(\frac{(1-\varepsilon_{C_i})}{\varepsilon_{C_i}}) \tag{5.27}$$

iii. Voting: For an incoming tweet say y, final decision l_y^F that whether instance belongs to *ham* or *spam* category is decided by majority voting approach, given as:

$$l_y^F \leftarrow arg(\Theta) \sum_{i=1}^{k} \alpha_{C_i} C_i(y) \tag{5.28}$$

A threshold (Θ) value is defined for voting mechanism. A tweet is classified as *ham or spam* when certain threshold is crossed by the aggregation of all classifiers.

iv. Databases Updation: The most important task associated with this phase is to update databases. The major issues in updating task are: frequency of updation and content to update in database. These issues are addressed as follows:

- *Updation Frequency:* Some semi supervised models use fixed time interval approach and some use random updation procedure. Major concerns with both approaches is time consumed in updation and delayed updates respectively. To resolve this problem, proposed framework uses dynamic mechanism to perform updates in databases based on markov bound. Updation is performed at twice: first is after executing data bucket for all classifiers and second is randomly. It keeps on checking the error bound of all classifier while analysing the tweets in bucket. If, at certain time, error bound is more then the defined bound then updations are performed in-between the processing of bucket. Markov bound has been used to check deviation from original result. If

deviation is within the defined bound then no updation is performed. If deviation exceeds the defined threshold then databases in the framework are updated.

$$\text{Update} = \begin{cases} Bucket(d_i) \text{ is processed} \\ Pr[error > c] \leq \frac{E(error)}{c} \end{cases} \tag{5.29}$$

where c is a constant used to define the threshold and $E(error)$ is excepted value of error of ensemble model given by:

$$E(error) = \frac{1}{k}\sum_{i=1}^{k}(\varepsilon_{C_i}) \tag{5.30}$$

- *Content for updation*: Databases like spam URL, spam words and identified spam users need to be updated periodically to improve the accuracy of classifiers for future iterations. Tweets which are marked as *spam* in final voting, but some intermediate classifier (I-III) detected them as *ham* are updated in their corresponded databases after performing a second level check (offline task). The tweets that exceed a defined true limit(T_l) are marked as 'feedback tweets'(ft).

$$\text{If}(\forall i|(\alpha_i \times l_i^{tw_o}) > T_l) \begin{cases} TRUE & (tw_o = ft) \\ FALSE & (tw_o \neq ft) \end{cases} \tag{5.31}$$

To update the model, data from all tweets marked as 'feedback tweets' is considered.

Input : Model Updation(D, L, L^*)
Output: Insert $x_i \in S$ for $t_i \in T$ in $ATBF_i$ array

1. L^* is set of labels from classifiers
2. L is set of original labels
3. **if** $d_i \neq \phi$ **then**
4.   **for** $\forall|tw_o \in d_i$ **do**
5.   Compute error bound ε_{C_i} (Eq. (5.23))
6.   **if** *(Random(Markov))* **then**
7.   Compute E(error) (Eq. (5.30))
8.   **if** $(Error > threshold)$ **then**
9.   GOTO `LABEL-A`
10.   **end**
11.   **end**
12.   Compute $l_F^{tw_o}$ (Using Eq (5.28))
13.   **if** $(\forall C_i|(l_{C_i} = 1)||(\forall i|(\alpha_i \times l_i^{tw_o}) > T_l))$ **then**
14.   Set $tw_o = ft$
15.   **end**
16.   **end**
17. **else**
18.   **LABEL-A:**
19.   Update probability of d_i (Using Eq. (5.24))
20.   Compute α_{C_i} (Importance weight for C_i) (Eq. (5.27))
21.   Update all the databases using ft data
22.   Update $QF^{BD}, QF^{SU}, QF^{SW}$
23. **end**

Algorithm 5.2: EbSDF: Model Updation Algorithm

The analysis of the experiments to evaluate the proposed scheme is done with various parameters on the simulated environment in the next section.

5.4 Experimental Analysis of Ensemble based Spam Detection in Social IoT

Following section (section 5.4.1) discuss the experiment evaluation for the Eb-SDF in detail.

5.4.1 Performance Evaluation Metrics

A comparative analysis is performed in Figures (5.2 - 5.5) on the basis of computational time required, query time, false positives generated by the QF and number of iterations required for optimized updation.

To measure the accuracy of the classifier and Eb-SDF (Figures (5.6-5.8)), standard measures like Precision, Recall, F-score are used. For the classification problem, following performance metric are considered:

- **Recall and Precision:** Standard accuracy measure given by:

$$Recall = \frac{t_p}{t_p + f_p} \tag{5.32}$$

$$Precision = \frac{t_p}{t_p + f_n} \tag{5.33}$$

- **F-Score:** A measure that combines precision and recall (harmonic mean of precision and recall) given by:

$$F - score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{5.34}$$

where true positive (t_p) represents the elements present in the QF that are successfully queried, false positive (f_p) denotes the element not present in QF but query process return *true*, and false negative (f_n) denotes the elements which are present in the set and query process return *false* while querying them. For each classifier C_i, f_p and f_n are calculated on the basis of ε_{C_i}, using Eq. (5.23).

5.4.2 Observations and Analysis

For data simulation in the Eb-SDF MATLAB has been used. During simulations, an ensemble containing four classifiers is setup where each tweet is given to all classifiers in parallel

and then final result is decided on the basis of majority voting. All experiments are performed on the dataset collected by using script available on Kaggle as "test-on-twitter" [187], where tweets are extracted for a given time span. Various parameters have been configured to analyze the performance of the proposed approach. Search complexity, error bound and accuracy parameters like precision, recall, *etc.*, have been evaluated using a query set generated from original data. All tweets in query set have the original labels to make comparative analysis for performance evaluation.

Figures(5.2- 5.5) depict the advantage of using PDS in classifiers(I-III) compared to traditional approaches on the basis of computational time and query time complexity. In fig. 5.2, comparative analysis of LSH and indexing based similarity approach is provided *w.r.t.* number of tweets randomly selected for the set $R_{S/H}$. In indexing approach, the main focus is to store an index in such a way that it optimizes the speed and performance of search result. In indexing based approach, computations required increases drastically as the number of elements for the comparison increases. LSH includes only preprocessing overhead which is independent of the input size.

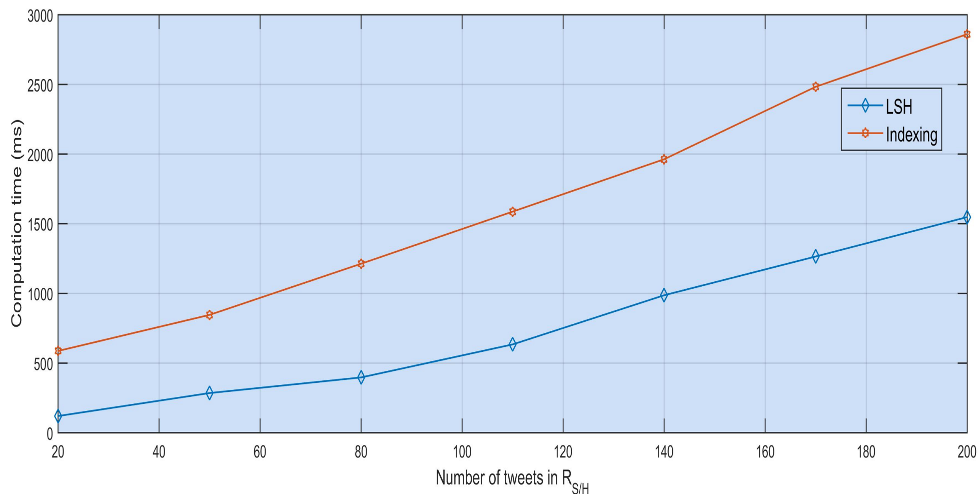

Figure 5.2: Computational time (ms) for LSH and hash indexing with increase in number of tweets

Fig. 5.3 depicts comparative analysis of approximate membership query using QF, hash tables and B+ tree. Hash table is a deterministic data structure which maps keys to values.

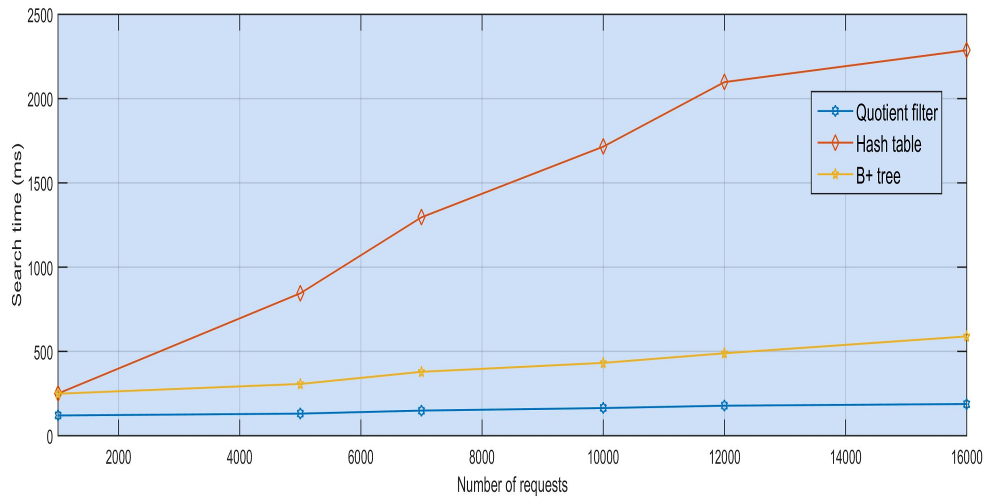

Figure 5.3: Search time (ms) for QF, hash table and B+ tree

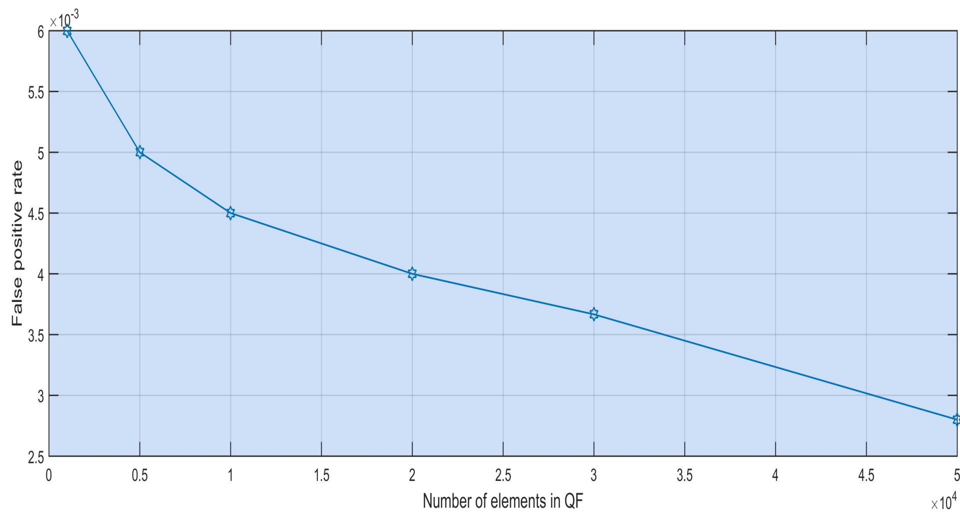

Figure 5.4: False positive rate vs. number of elements in Quotient Filter

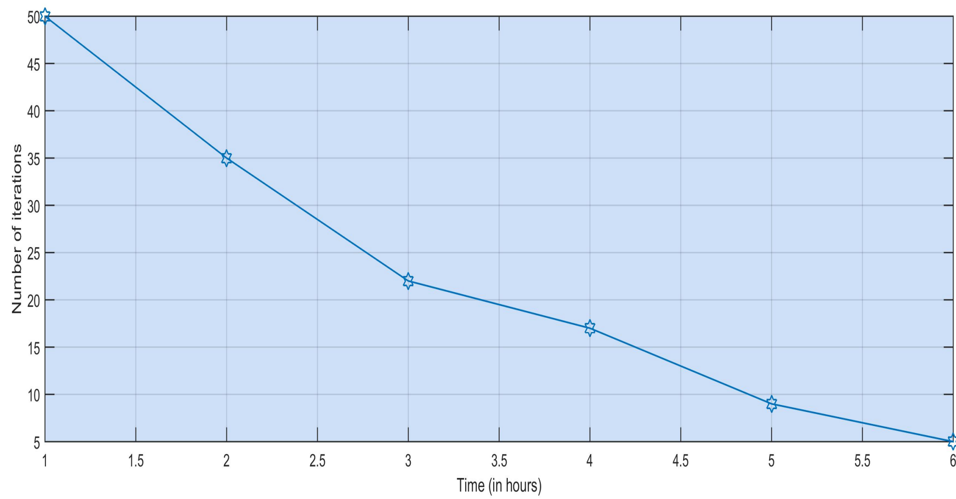

Figure 5.5: Number of iterations vs. time (in hours) required for optimized updation

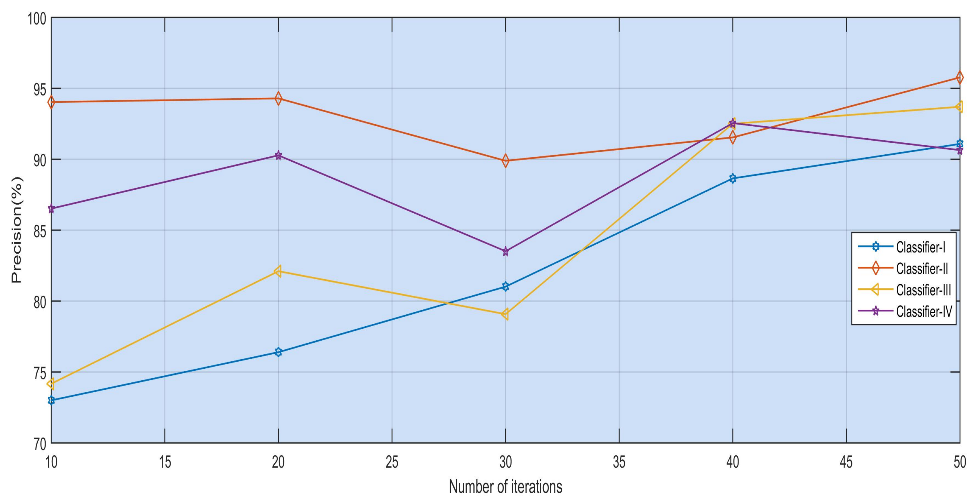

Figure 5.6: Precision of classifiers(I-IV) vs. number of iterations

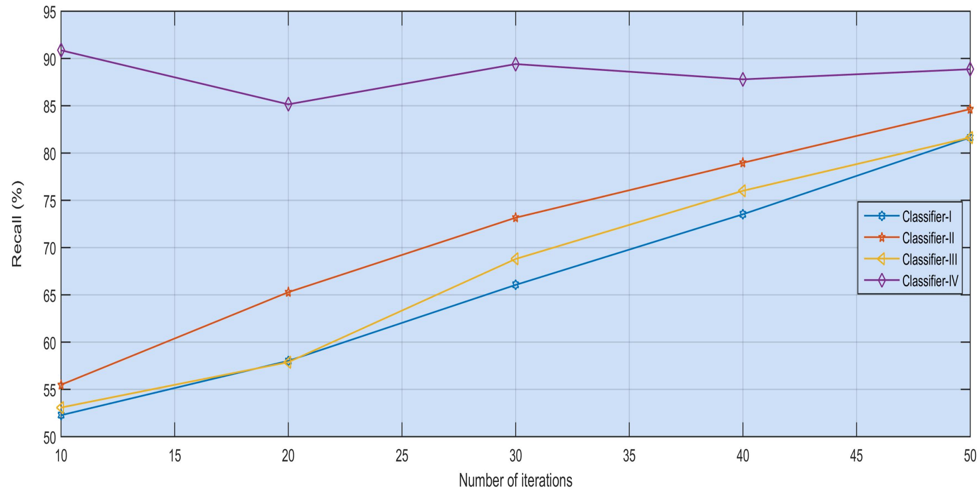

Figure 5.7: Recall of classifiers(I-IV) vs. number of iterations

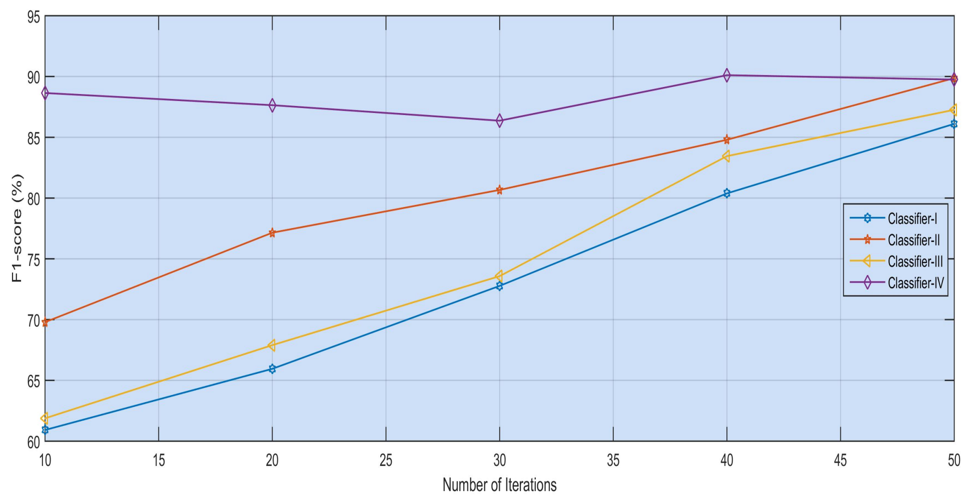

Figure 5.8: F1 score of classifiers(I-IV) vs. number of iterations

A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. The B+ tree is an n-array tree with a variable but often large number of children per node. It consists of a root, internal nodes and leaves where each node contains only keys and to store values in B+ tree an additional level is added at the bottom with linked leaves. For hash tables, search time complexity for k open address hash table is $O(\frac{n}{k}+1)$ and in B+ tree searching complexity for a tree with b keys is $O(log_b(n))$, where n is the size of input. In QF searching time is dependent only on time to hash and it is independent of the size of input.

Fig. 5.4 shows the probability of error in QF, *i.e.*, false positives varying with the number of elements stored in quotient filter. Fig. 5.5 shows the number of updations required with the increase in time span. In the initial state, databases are not updated so classifier (I-III) error bound are below the threshold and updation process is called more frequently. As the number of iterations increase, *i.e.*, database becomes more mature, number of updations required decrease.

In fig. 5.6- 5.8, results of classifiers are compared on the basis of accuracy parameters *w.r.t.* the number of iterations (updations done). Fig. 5.6, 5.7 and 5.8 show the comparative study of classifiers on the basis of precision, recall and F1-score respectively. For classifiers(I-III) false positives are high in the initial stage hence precision is less. As the number of updations in the database increase, false positives decrease and false negatives are also less in the final result. Precision, recall and F1-score for all the classifiers(I-III) increase as more and more updations are performed. Classifier IV is based on the standard techniques which shows results independent of the updation process.

5.4.3 Discussion

In above section results of PDS based Eb-SDF has been discussed, which shows significant improvement in performance and fulfills stated objectives efficiently. It uses probabilistic data structures as classifiers in association with standard classifiers which enhance

the overall accuracy of the system. Eb-SDF fulfill all objective stated in function defined in Eq. (5.1). Deviation(δ) in result is in minimized since it considers the final decision by adaptive weighted voting mechanism, based on the output generated by each classifier. Computational effort (ζ) have been reduced by the use of hybrid sampling technique, which samples the data according to the classifier. Use of quotient filter in clasifier I and III for membership query and LSH in classifier II for similarity search provide fast results (reduces τ) using less computational time (minimizes ζ). Markov bound based dynamic model with fixed error bound (ε_{C_i}) help the Eb-SDF to adapt according to the changing nature of data (φ).

- Domains where Eb-SDF can be successfully employed for detecting spam:
  - Detecting malicious user in social networking platform like Facebook, Linkdn.
  - Detect manipulated/fake reviews in promotional platforms.
  - Identifying anomalous nature of user(attacker) in a network.

Coming chapter (Chapter 6) concluded the thesis with future directions of the thesis.

Chapter 6

Conclusion and Future Scope

Big Data is everywhere and there is almost an urgent need to collect and preserve whatever data is being generated, to uncover hidden patterns, unknown correlations, market trends, customer preferences and other useful information. This thesis work provides comprehensive discussions on Big data analytic and PDS and how problem faced in the current scenarios especially in massive data sets can be solved using PDS. Section 6.1 concludes the thesis work and section 6.2 discusses the contribution of proposed techniques. Finally section 6.3 provides the future scope of extension in the proposed solutions.

6.1 Conclusion

The focus of this thesis has been on efficient storage and retrieval of massive datasets especially in stream data analysis for operations which include querying, similarity search, spam detection, *etc.*

A new variant of BF, *i.e.*, ATBF is proposed for processing streamed data in a defined time windows. To accommodate dynamic data and query the hourly information, the proposed BF uses the properties of ageing BF, *i.e.*, evicting data after fixed time interval and incremental BF properties, *i.e.*, changes size as the data increases. For duplicate detection of streaming data in one pass and to accommodate unbounded data, another variant in the

category of Aging BF called FP-SBF has been proposed. FP-SBF uses stable bloom filter with fingerprint bits to decrease error rate. An ensemble based framework, Eb-SDF, for spam detection in Twitter is proposed which uses PDS as classifiers in various stages of ensemble to mark a tweet as *Spam or Ham*. Final decision of the four stage ensemble is made on the basis of majority voting. Markov bound based updation helps in making the framework adaptive as timely updates make it more efficient.

6.2 Thesis Contributions

Major contributions of this work are:

- Three techniques: ATBF (for hourly analysis and update), FP-SBF (for duplicate detection) and Eb-SDF (for spam detection in social networks) have been proposed and results achieved validate the effectiveness of the proposed work.
- All techniques use PDS which reduces the memory requirement drastically; processing time and query complexity of PDS is quite less when compared with deterministic data structures.
- The techniques proposed in this thesis can be successfully applied in various real time applications which include web-analytics, streaming data analysis, software defined networks, IoT data streams, *etc.*

6.3 Future Scope

In this thesis work, streamed data analysis is performed for different application domains by using PDS like Bloom filter, Quotient filter, Locality Sensitive Hashing. More PDS like Count-Min Sketch for frequency estimation, Hyperloglog counter for cardinality estimation, Skip List and treap for randomized storage can also be used in Big data analytics.

ATBF proposed for streaming data analysis task like peak hour analysis, server utilization, can further be combined with machine learning algorithms to provide results at much faster rate. FP-SBF which performs efficiently for duplicate detection problem can be extended to design hash functions which consider spatial and temporal data sets. The proposed ensemble based spam detection framework has been tested for tweets generated by Twitter. In future, it can be enhanced by deploying it for other social media platforms like Facebook, Linkedln, *etc.* Further, it can be implemented in real time data from Social IoT network.

Various schemes proposed in this work use variant of scalable bloom filter and stable bloom filter which can be made more computationally efficient by using hybrid hashing with various optimization approaches.

Bibliography

[1] V. Mayer-Schnberger, *Big Data: A Revolution That Will Transform How We Live, Work and Think. Viktor Mayer-Schnberger and Kenneth Cukier*. UK: John Murray Publishers, 2013.

[2] K. Krishnan, *Data Warehousing in the Age of Big Data*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed., 2013.

[3] S. John Walker, "Big data: A revolution that will transform how we live, work, and think," 2014.

[4] A. Big Data, "new world of opportunities," *NESSI White Paper*, 2012.

[5] R. Thomson, C. Lebiere, and S. Bennati, "Human, model and machine: a complementary approach to big data," in *Proceedings of the 2014 Workshop on Human Centered Big Data Research*, p. 27, ACM, 2014.

[6] A. Snell, "Solving big data problems with private cloud storage," *Intersect360 Research, Cloud Computing in HPC: Usage and Types, June*, 2011.

[7] F. Xia, L. T. Yang, L. Wang, and A. Vinel, "Internet of things," *International Journal of Communication Systems*, vol. 25, no. 9, p. 1101, 2012.

[8] G. Lu and W. H. Zeng, "Cloud computing survey," in *Applied Mechanics and Materials*, vol. 530, pp. 650–661, Trans Tech Publ, 2014.

[9] X. Jin, B. W. Wah, X. Cheng, and Y. Wang, "Significance and challenges of big data research," *Big Data Research*, vol. 2, no. 2, pp. 59–64, 2015.

[10] W. Gentzsch, "Grid initiatives: Lessons learned and recommendations," *RENCI Report retrieved from www. renci. org/publications/reports. php*, 2007.

[11] E. Slack, "Storage infrastructures for big data workflows," *Storage Switchland, LLC, Tech. Rep*, 2012.

[12] H. Chen, R. H. Chiang, and V. C. Storey, "Business intelligence and analytics: From big data to big impact.," *MIS quarterly*, vol. 36, no. 4, pp. 1165–1188, 2012.

[13] A. Cuzzocrea, "Privacy and security of big data: current challenges and future research perspectives," in *Proceedings of the First International Workshop on Privacy and Secuirty of Big Data*, pp. 45–47, ACM, 2014.

[14] Gartner, "Big data." https://www.gartner.com/it-glossary/big-data, 2012.

[15] J.-P. Dijcks, "Oracle: Big data for the enterprise," *Oracle White Paper*, 2012.

[16] I. B. data and analytics hub, "The four v's of big data." http://www.ibmbigdatahub.com/infographic/four-vs-big-data, note =, 2013.

[17] Microsoft, "The big bang: How the big data explosion is changing the world," *Microsoft News Center*, Feb. 2012.

[18] G. C. Platform, "Google cloud big data solutions." https://cloud.google.com/what-is-big-data/, note =, 2017.

[19] Wikipedia, "Big data." https://en.wikipedia.org/wiki/Big_data, 2010. [Online; accessed Nov-2017].

[20] Casari, "Big data definition." http://dictionary.casrai.org/Big_data , 2015. [Online; accessed Nov-2017].

[21] Forbes, "12 big data definitions: What's yours?." https://www.forbes.com/sites/gilpress/2014/09/03/12-big-data-definitions-whats-yours/#464de2e113ae, 2014. [Online; accessed Nov-2017].

[22] Facebook, "Facebook: 10 years of social networking, in numbers." https://www.theguardian.com/news/datablog/2014/feb/04/facebook-in-numbers-statistics, note =, 2014.

[23] J. Bullas, "22 social media facts and statistics you should know in 2014." http://www.jeffbullas.com/20-social-media-facts-and-statistics-you-should-know-in-2014/, note =, 2014.

[24] M. Conner, "Data on big data." http://marciaconner.com/blog/data-on-big-data/, note =, 2014.

[25] YouTube, "Youtube for press." https://www.youtube.com/yt/about/press/, note =, 2017.

[26] Forbes, "Unstructured data: The other side of analytics." https://www.forbes.com/sites/steveandriole/2015/03/05/the-other-side-of-analytics/#789db5192b8d , note =, 2015.

[27] Fortune, "Linkedin lost 167 million account credentials in data breach." http://fortune.com/2016/05/18/linkedin-data-breach-email-password/ , note =, 2016.

[28] A. Whitmore, A. Agarwal, and L. Da Xu, "The internet of things?a survey of topics and trends," *Information Systems Frontiers*, vol. 17, no. 2, pp. 261–274, 2015.

[29] J. Scott, *Social network analysis*. Sage, 2017.

[30] L. Atzori, A. Iera, and G. Morabito, “From” smart objects” to” social objects”: The next evolutionary step of the internet of things,” *IEEE Communications Magazine*, vol. 52, no. 1, pp. 97–105, 2014.

[31] T. Hey, S. Tansley, K. M. Tolle, *et al.*, *The fourth paradigm: data-intensive scientific discovery*, vol. 1. Microsoft research Redmond, WA, 2009.

[32] E. Liberty and J. Nelson, “Streaming data mining,” Presented at Princeton University by Yahoo Research group, 2012.

[33] P. Zikopoulos, C. Eaton, *et al.*, *Understanding big data: Analytics for enterprise class hadoop and streaming data*. McGraw-Hill Osborne Media, 2011.

[34] A. Labrinidis and H. V. Jagadish, “Challenges and opportunities with big data,” *Proc. VLDB Endow.*, vol. 5, pp. 2032–2033, Aug. 2012.

[35] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, “Models and issues in data stream systems,” in *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS ’02, (New York, NY, USA), pp. 1–16, ACM, 2002.

[36] M. P. Singh, M. A. Hoque, and S. Tarkoma, “Analysis of systems to process massive data stream,” *CoRR*, 2016.

[37] U. N. G. Pulse, “Big data for development: challenges and opportunities.” http://www.unglobalpulse.org/projects/BigDataforDevelopment/, 2012.

[38] J. A. Ou and S. H. Penman, “Financial statement analysis and the prediction of stock returns,” *Journal of Accounting and Economics*, vol. 11, no. 4, pp. 295 – 329, 1989.

[39] J. Bobadilla, F. Ortega, A. Hernando, and A. GutiéRrez, “Recommender systems survey,” *Know.-Based Syst.*, vol. 46, pp. 109–132, July 2013.

[40] O. Ylijoki and J. Porras, “Conceptualizing big data: Analysis of case studies,” *Intelligent Systems in Accounting, Finance and Management*, 2016.

[41] “Denial of service attack.” https://en.wikipedia.org/wiki/Denial-of-service-attack. Online.

[42] L. Atzori, A. Iera, G. Morabito, and M. Nitti, “The social internet of things (siot)–when social networks meet the internet of things: Concept, architecture and network characterization,” *Computer networks*, vol. 56, no. 16, pp. 3594–3608, 2012.

[43] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, “Internet of things (iot): A vision, architectural elements, and future directions,” *Future Gener. Comput. Syst.*, vol. 29, pp. 1645–1660, Sept. 2013.

[44] A. H. Ngu, M. Gutierrez, V. Metsis, S. Nepal, and Q. Z. Sheng, “Iot middleware: A survey on issues and enabling technologies,” *IEEE Internet of Things Journal*, vol. 4, no. 1, pp. 1–20, 2017.

[45] M. R. Group *et al.*, "Internet of things (iot) & m2m communication market-advanced technologies, future cities & adoption trends, roadmaps & worldwide forecasts 2012-2017," *Electronics. ca Publications, Tech. Rep*, 2012.

[46] S. García, S. Ramírez-Gallego, J. Luengo, J. M. Benítez, and F. Herrera, "Big data preprocessing: methods and prospects," *Big Data Analytics*, vol. 1, no. 1, p. 9, 2016.

[47] U. Sivarajah, M. M. Kamal, Z. Irani, and V. Weerakkody, "Critical analysis of big data challenges and analytical methods," *Journal of Business Research*, vol. 70, pp. 263–286, 2017.

[48] H. S. Blog, "Probabilistic data structures for web analytics and data mining." http://highlyscalable.wordpress.com/2012/05/01/ probabilistic-structures-web-analytics-data-mining/, 2014.

[49] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*. Cambridge university press, 2017.

[50] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, July 1970.

[51] M. A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok, "Don't thrash: How to cache your hash on flash," *Proc. VLDB Endow.*, vol. 5, pp. 1627–1637, July 2012.

[52] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[53] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*, pp. 693–703, Springer, 2002.

[54] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *In ESA*, pp. 605–617, 2003.

[55] P. Flajolet, E. Fusy, and O. Gandouet, "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in *Proceedings of The International Conference On Analysis Of Algorithms (AOFA'07)*, 2007.

[56] T. karnezos, "Hll talk at sfpug." https://research.neustar.biz/2014/09/23/hll-talk-at-sfpug/, September 2014. [Accessed Online: Jan 2017].

[57] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *Commun. ACM*, vol. 51, pp. 117–122, Jan. 2008.

[58] O. Chum, M. Perd'och, and J. Matas, "Geometric min-hashing: Finding a thick needle in a haystack," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'09)*, pp. 17–24, IEEE, 2009.

[59] N. Chand, R. C. Joshi, and M. Misra, "Cooperative caching strategy in mobile ad hoc networks based on clusters," *Wireless Personal Communications*, vol. 43, no. 1, pp. 41–63, 2007.

[60] G. Thiriveni and M. Ramakrishnan, "Distributed clustering based energy efficient routing algorithm for heterogeneous wireless sensor networks," *Indian Journal of Science and Technology*, vol. 9, no. 3, 2016.

[61] C. Clifton, M. Kantarcioglu, J. Vaidya, X. Lin, and M. Y. Zhu, "Tools for privacy preserving distributed data mining," *ACM Sigkdd Explorations Newsletter*, vol. 4, no. 2, pp. 28–34, 2002.

[62] Y. Wang, L. Kung, W. Y. C. Wang, and C. G. Cegielski, "An integrated big data analytics-enabled transformation model: Application to health care," *Information & Management*, 2017.

[63] B. Pandey and R. Mishra, "Knowledge and intelligent computing system in medicine," *Computers in biology and medicine*, vol. 39, no. 3, pp. 215–230, 2009.

[64] A. Kaulfus, S. Alexander, S. Zhao, R. A. Oster, L. C. O'keefe, and A. Bartolucci, "The inherent challenges of using large data sets in healthcare research: Experiences of an interdisciplinary team," *CIN: Computers, Informatics, Nursing*, vol. 35, no. 5, pp. 221–225, 2017.

[65] B. Klievink, B.-J. Romijn, S. Cunningham, and H. de Bruijn, "Big data in the public sector: Uncertainties and readiness," *Information Systems Frontiers*, vol. 19, no. 2, pp. 267–283, 2017.

[66] A. N. Toosi, R. K. Thulasiram, and R. Buyya, "Financial option market model for federated cloud environments," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, pp. 3–12, IEEE Computer Society, 2012.

[67] L. Y.-F. Su, M. A. Cacciatore, X. Liang, D. Brossard, D. A. Scheufele, and M. A. Xenos, "Analyzing public sentiments online: combining human-and computer-based content analysis," *Information, Communication & Society*, vol. 20, no. 3, pp. 406–427, 2017.

[68] H.-C. Chu, D.-J. Deng, and J. H. Park, "Live data mining concerning social networking forensics based on a facebook session through aggregation of social data," *IEEE Journal on Selected Areas in Communications*, vol. 29, no. 7, pp. 1368–1376, 2011.

[69] P. Mitra and K. Baid, "Targeted advertising for online social networks," in *Networked Digital Technologies, 2009. NDT'09. First International Conference on*, pp. 366–372, IEEE, 2009.

[70] O. Marjanovic, B. Dinter, and T. Ariyachandra, "Introduction to organizational issues of business intelligence, business analytics and big data minitrack," in *Proceedings of the 50th Hawaii International Conference on System Sciences*, 2017.

[71] A. G. Barnston and R. E. Livezey, “Classification, seasonality and persistence of low-frequency atmospheric circulation patterns,” *Monthly Weather Review*, vol. 115, no. 6, pp. 1083–1126, 1987.

[72] H. Channe, S. Kothari, and D. Kadam, “Multidisciplinary model for smart agriculture using internet-of-things (iot), sensors, cloud-computing, mobile-computing & big-data analysis,” *Int. J. Computer Technology & Applications*, vol. 6, no. 3, pp. 374–382, 2015.

[73] Y. Zhao, S. Guo, and Y. Yang, “Hermes: An optimization of hyperloglog counting in real-time data processing,” in *International Joint Conference on Neural Networks (IJCNN)*, pp. 1890–1895, IEEE, 2016.

[74] S. F. Wamba, A. Gunasekaran, S. Akter, S. J.-f. Ren, R. Dubey, and S. J. Childe, “Big data analytics and firm performance: Effects of dynamic capabilities,” *Journal of Business Research*, vol. 70, pp. 356–365, 2017.

[75] V. Deshmukh, K. Komatsu, and P. Saraswat, “System and method for storing and accessing data using a plurality of probabilistic data structures,” Oct. 2012.

[76] G. S. Lueker and M. Molodowitch, “More analysis of double hashing,” *Combinatorica*, vol. 13, no. 1, pp. 83–96, 1993.

[77] “Bloom filter by arpita korwar.” http://www.cse.iitk.ac.in/users/arpk/articles/BloomFilters.pdf . [Online; May 2010].

[78] Y. Kanizo, D. Hay, and I. Keslassy, “Optimal fast hashing,” in *INFOCOM 2009, IEEE*, pp. 2500–2508, IEEE, 2009.

[79] F. Bonomi, M. Mitzenmacher, R. Panigraphy, S. Singh, and G. Varghese, “Bloom filters via d-left hashing and dynamic bit reassignment extended abstract,” in *Forty-Fourth Annual Allerton Conf., Illinois, USA*, pp. 877–883, 2006.

[80] A. Kirsch and M. Mitzenmacher, “Less hashing, same performance: Building a better bloom filter,” *Random Struct. Algorithms*, vol. 33, pp. 187–218, Sept. 2008.

[81] R. Milner, “Towards the technical unification of consistent hashing and the partition table,” *Journal of Real-Time Theory*, vol. 38, pp. 20–24, 1998.

[82] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, “Theory and practice of bloom filters for distributed systems,” *IEEE Communications Surveys Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[83] A. Kirsch and M. Mitzenmacher, “Distance-sensitive bloom filters.,” in *Proceedings of the Meeting on Algorithm Engineering & Experimiments*, vol. 6, (Philadelphia, PA, USA), pp. 41–50, SIAM, 2006.

[84] J. Bruck, J. Gao, and A. Jiang, “Weighted bloom filter,” in *IEEE International Symposium on Information Theory*, IEEE, 2006.

[85] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, pp. 281–293, June 2000.

[86] S. Cohen and Y. Matias, "Spectral bloom filters," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, (New York, NY, USA), pp. 241–252, ACM, 2003.

[87] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese, "An improved construction for counting bloom filters," in *Proceedings of the 14th Conference on Annual European Symposium - Volume 14*, ESA'06, (London, UK, UK), pp. 684–695, Springer-Verlag, 2006.

[88] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, and M. F. Magalhaes, "The deletable bloom filter: a new member of the bloom family," *IEEE Communications Letters*, vol. 14, pp. 557–559, June 2010.

[89] D. Guo, J. Wu, H. Chen, Y. Yuan, and X. Luo, "The dynamic bloom filters," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 1, pp. 120–133, 2010.

[90] P. S. Almeida, C. Baquero, N. Preguiça, and D. Hutchison, "Scalable bloom filters," *Inf. Process. Lett.*, vol. 101, pp. 255–261, Mar. 2007.

[91] K. Xie, Y. Min, D. Zhang, J. Wen, and G. Xie, "A scalable bloom filter for membership queries," in *Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE*, pp. 543–547, Nov 2007.

[92] F. Deng and 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, (New York, NY, USA), pp. 25–36, ACM, 2006.

[93] F. Chang, W. chang Feng, and K. Li, "Approximate caches for packet classification," in *INFOCOM 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies*, vol. 4, pp. 2196–2207 vol.4, March 2004.

[94] M. Yoon, "Aging bloom filter with two active buffers for dynamic sets," *IEEE Trans. Knowl. Data Eng.*, vol. 22, no. 1, pp. 134–138, 2010.

[95] M. Mitzenmacher, "Compressed bloom filters," *IEEE/ACM Transactions on Networking*, vol. 10, no. 5, pp. 604–612, 2002.

[96] A. Kumar, J. J. Xu, L. Li, and J. Wang, "Space-code bloom filter for efficient traffic flow measurement," in *Proceedings of the 3rd ACM SIGCOMM Conference on Internet Measurement*, IMC'03, (New York, USA), pp. 167–172, ACM, 2003.

[97] E.-J. Goh, "Secure indexes.," *IACR Cryptology ePrint Archive*, vol. 2003, pp. 2–16, 2003.

[98] K. Shanmugasundaram, H. Brönnimann, and N. Memon, "Payload attribution via hierarchical bloom filters," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS'04, (New York, USA), pp. 31–41, ACM, 2004.

[99] B. Chazelle, J. Kilian, R. Rubinfeld, and 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*, SODA'04, (Philadelphia, PA, USA), pp. 30–39, Society for Industrial and Applied Mathematics, 2004.

[100] M.-Z. Xiao, Y.-F. Dai, and X.-M. Li, "Split bloom filter," *Tien Tzu Hsueh Pao/Acta Electronica Sinica*, vol. 32, pp. 241–245, 2004.

[101] F. Chang, W. chang Feng, and K. Li, "Approximate caches for packet classification," in *Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies (INFOCOM'04)*, vol. 4, pp. 2196–2207, March 2004.

[102] Y. Lu, B. Prabhakar, and F. Bonomi, "Bloom filters: Design innovations and novel applications," in *In Proc. of the Forty-Third Annual Allerton Conference*, 2005.

[103] B. Donnet, B. Baynat, and T. Friedman, "Retouched bloom filters: Allowing networked applications to trade off selected false positives against false negatives," in *Proceedings of the ACM CoNEXT Conference*, CoNEXT'06, (New York, USA), pp. 13:1–13:12, ACM, 2006.

[104] J. Bruck, J. Gao, and A. A. Jiang, "Adaptive bloom filter," *California Institute of Technology*, 2006.

[105] M. Zhong, P. Lu, K. Shen, and J. Seiferas, "Optimizing data popularity conscious bloom filters," in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC'08, (New York, NY, USA), pp. 355–364, ACM, 2008.

[106] M. Ahmadi and S. Wong, "A memory-optimized bloom filter using an additional hashing function," in *IEEE Global Telecommunications Conference (GLOBECOM'08)*, pp. 1–5, Nov 2008.

[107] A. Goel and P. Gupta, "Small subset queries and bloom filters using ternary associative memories, with applications," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 143–154, June 2010.

[108] R. P. Laufer, P. B. Velloso, and O. C. M. B. Duarte, "A generalized bloom filter to secure distributed network applications," *Comput. Netw.*, vol. 55, pp. 1804–1819, June 2011.

[109] J. L. Dautrich, Jr. and C. V. Ravishankar, "Inferential time-decaying bloom filters," in *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT'13, (New York, USA), pp. 239–250, ACM, 2013.

[110] X. Gong, W. Qian, Y. Yan, and A. Zhou, "Bloom filter-based xml packets filtering for millions of path queries," in *Data Engineering, 2005. ICDE 2005. Proceedings. 21st International Conference on*, pp. 890–901, IEEE, 2005.

[111] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 1–16, ACM, 2002.

[112] J. S. Merlin and A. V. A. Mary, "An approach for quick & efficient detection of duplicate data-survey," *International Journal of Applied Engineering Research*, vol. 11, no. 5, pp. 3430–3432, 2016.

[113] A. Metwally, D. Agrawal, and A. El Abbadi, "Duplicate detection in click streams," in *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, (New York, NY, USA), pp. 12–21, ACM, 2005.

[114] J. Wei, H. Jiang, K. Zhou, D. Feng, and H. Wang, "Detecting duplicates over sliding windows with ram-efficient detached counting bloom filter arrays," in *Networking, Architecture and Storage (NAS), 2011 6th IEEE International Conference on*, pp. 382–391, IEEE, 2011.

[115] S. Dutta, S. Bhattacherjee, and A. Narang, "Towards intelligent compression in streams: A biased reservoir sampling based bloom filter approach," in *Proceedings of the 15th International Conference on Extending Database Technology*, pp. 228–238, ACM, 2012.

[116] O. Rottenstreich, Y. Kanizo, and I. Keslassy, "The variable-increment counting bloom filter," *IEEE/ACM Transactions on Networking*, vol. 22, no. 4, pp. 1092–1105, 2014.

[117] S. Pontarelli, P. Reviriego, and J. A. Maestro, "Improving counting bloom filter performance with fingerprints," *Information Processing Letters*, vol. 116, no. 4, pp. 304 – 309, 2016.

[118] S. Geravand and M. Ahmadi, "Survey bloom filter applications in network security: A state-of-the-art survey," *Comput. Netw.*, vol. 57, pp. 4047–4064, Dec. 2013.

[119] S. Geravand and M. Ahmadi, "Bloom filter applications in network security: A state-of-the-art survey," *Computer Networks*, vol. 57, no. 18, pp. 4047–4064, 2013.

[120] K. W. Choi, D. T. Wiriaatmadja, and E. Hossain, "Discovering mobile applications in cellular device-to-device communications: Hash function and bloom filter-based approach," *IEEE Transactions on Mobile Computing*, vol. 15, no. 2, pp. 336–349, 2016.

[121] K. Verma and H. Hasbullah, "Bloom-filter based ip-chock detection scheme for denial of service attacks in vanet," *Security and Communication Networks*, vol. 8, no. 5, pp. 864–878, 2015.

[122] W. Song, B. Wang, Q. Wang, Z. Peng, W. Lou, and Y. Cui, "A privacy-preserved full-text retrieval algorithm over encrypted data for cloud storage applications," *Journal of Parallel and Distributed Computing*, vol. 99, pp. 14 – 27, 2017.

[123] Quora, "What are the best applications of bloom filters?," 2014. [Accessed Online: Feb 2017].

[124] D. E. Knuth, *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, 1998.

[125] M. Al-hisnawi and M. Ahmadi, "Deep packet inspection using quotient filter," *IEEE Communications Letters*, vol. 20, pp. 2217–2220, Nov 2016.

[126] P. Goudarzi, H. T. Malazi, and M. Ahmadi, "Khorramshahr: A scalable peer to peer architecture for port warehouse management system," *Journal of Network and Computer Applications*, vol. 76, pp. 49 – 59, 2016.

[127] S. Dutta, A. Narang, and S. K. Bera, "Streaming quotient filter: A near optimal approximate duplicate detection approach for data streams," *Proc. VLDB Endow.*, vol. 6, pp. 589–600, June 2013.

[128] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press, 2011.

[129] A. Al-Fuqaha, "Similarity analysis and distance min-hashing locality sensitive hashing." https://cs.wmich.edu/ al-fuqaha/summer14/cs6530/lectures/SimilarityAnalysis.pdf.

[130] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[131] P. Indyk and R. Motwani, "Approximate nearest neighbors: Towards removing the curse of dimensionality," in *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, (New York, NY, USA), pp. 604–613, ACM, 1998.

[132] "Nearest neighbor search." https://people.csail.mit.edu/gregory/annbook/introduction.pdf . [Online].

[133] A. Broder, "On the resemblance and containment of documents," in *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, (Washington, DC, USA), pp. 21–, IEEE Computer Society, 1997.

[134] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), pp. 518–529, Morgan Kaufmann Publishers Inc., 1999.

[135] J. Wang, H. T. Shen, J. Song, and J. Ji, "Hashing for similarity search: A survey," *CoRR*, vol. abs/1408.2927, 2014.

[136] A. Andoni and P. Indyk, "E2lsh user manual," 2005. [Online; June 21, 2005].

[137] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG'4, pp. 253–262, ACM, 2004.

[138] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC'02, (New York, USA), pp. 380–388, ACM, 2002.

[139] F. Chierichetti and R. Kumar, "Lsh-preserving functions and their applications," *Journal of the ACM (JACM)*, vol. 62, no. 5, p. 33, 2015.

[140] A. Becker, L. Ducas, N. Gama, and T. Laarhoven, "New directions in nearest neighbor searching with applications to lattice sieving," in *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 10–24, SIAM, 2016.

[141] Z. Kang, W. T. Ooi, and Q. Sun, "Hierarchical, non-uniform locality sensitive hashing and its application to video identification," in *IEEE International Conference on Multimedia and Expo (ICME'04)*, vol. 1, pp. 743–746, IEEE, 2004.

[142] C. Soh, H. B. K. Tan, Y. L. Arnatovich, and L. Wang, "Detecting clones in android applications through analyzing user interfaces," in *2015 IEEE 23rd International Conference on Program Comprehension*, pp. 163–173, May 2015.

[143] K. Berlin, S. Koren, C.-S. Chin, J. P. Drake, J. M. Landolin, and A. M. Phillippy, "Assembling large genomes with single-molecule sequencing and locality-sensitive hashing," *Nature biotechnology*, vol. 33, no. 6, pp. 623–630, 2015.

[144] N. Spirin and J. Han, "Survey on web spam detection: principles and algorithms," *ACM SIGKDD Explorations Newsletter*, vol. 13, no. 2, pp. 50–64, 2012.

[145] A. Schulz, C. Guckelsberger, and F. Janssen, "Semantic abstraction for generalization of tweet classification: An evaluation of incident-related tweets," *Semantic Web*, vol. 8, no. 3, pp. 353–372, 2017.

[146] "Spam." https://en.wikipedia.org/wiki/Spam . [Online].

[147] G. V. Cormack *et al.*, "Email spam filtering: A systematic review," *Foundations and Trends® in Information Retrieval*, vol. 1, no. 4, pp. 335–455, 2008.

[148] N. Spirin and J. Han, "Survey on web spam detection: principles and algorithms," *ACM SIGKDD Explorations Newsletter*, vol. 13, no. 2, pp. 50–64, 2012.

[149] F. Benevenuto, T. Rodrigues, V. Almeida, J. Almeida, and M. Gonçalves, "Detecting spammers and content promoters in online video social networks," in *Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, pp. 620–627, ACM, 2009.

[150] N. Jindal and B. Liu, "Review spam detection," in *Proceedings of the 16th international conference on World Wide Web*, pp. 1189–1190, ACM, 2007.

[151] G. Mishne, D. Carmel, R. Lempel, *et al.*, “Blocking blog spam with language model disagreement.,” in *AIRWeb*, vol. 5, pp. 1–6, 2005.

[152] S. Xie, G. Wang, S. Lin, and P. S. Yu, “Review spam detection via temporal pattern discovery,” in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 823–831, ACM, 2012.

[153] J. Weng, E.-P. Lim, J. Jiang, and Q. He, “Twitterrank: finding topic-sensitive influential twitterers,” in *Proceedings of the third ACM international conference on Web search and data mining*, pp. 261–270, ACM, 2010.

[154] K. Lee, J. Caverlee, and S. Webb, “The social honeypot project: protecting online communities from spammers,” in *Proceedings of the 19th international conference on World wide web*, pp. 1139–1140, ACM, 2010.

[155] K. Thomas, C. Grier, D. Song, and V. Paxson, “Suspended accounts in retrospect: an analysis of twitter spam,” in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pp. 243–258, ACM, 2011.

[156] D. Wang, S. B. Navathe, L. Liu, D. Irani, A. Tamersoy, and C. Pu, “Click traffic analysis of short url spam on twitter,” in *Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom), 2013 9th International Conference Conference on*, pp. 250–259, IEEE, 2013.

[157] C. Cao and J. Caverlee, “Detecting spam urls in social media via behavioral analysis,” in *European Conference on Information Retrieval*, pp. 703–714, Springer, 2015.

[158] S. Sedhai and A. Sun, “Hspam14: A collection of 14 million tweets for hashtag-oriented spam research,” in *Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 223–232, ACM, 2015.

[159] J. Messias, L. Schmidt, R. Oliveira, and F. Benevenuto, “You followed my bot! transforming robots into influential users in twitter,” *First Monday*, vol. 18, no. 7, 2013.

[160] K. Lee, J. Caverlee, and S. Webb, “Uncovering social spammers: social honeypots+ machine learning,” in *Proceedings of the 33rd international ACM SIGIR conference on Research and development in information retrieval*, pp. 435–442, ACM, 2010.

[161] X. Hu, J. Tang, Y. Zhang, and H. Liu, “Social spammer detection in microblogging.,” in *IJCAI*, vol. 13, pp. 2633–2639, Citeseer, 2013.

[162] E. Tan, L. Guo, S. Chen, X. Zhang, and Y. Zhao, “Unik: Unsupervised social network spam detection,” in *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*, pp. 479–488, ACM, 2013.

[163] E. Ferrara, O. Varol, C. Davis, F. Menczer, and A. Flammini, “The rise of social bots,” *Communications of the ACM*, vol. 59, no. 7, pp. 96–104, 2016.

[164] I. Santos, I. Minambres-Marcos, C. Laorden, P. Galan-Garcia, A. Santamaria-Ibirika, and P. G. Bringas, "Twitter content-based spam filtering," in *International Joint Conference SOCO2013-CISIS2013-ICEUTE2013*, pp. 449–458, Springer, 2014.

[165] J. Martinez-Romo and L. Araujo, "Detecting malicious tweets in trending topics using a statistical analysis of language," *Expert Systems with Applications*, vol. 40, no. 8, pp. 2992–3000, 2013.

[166] C. Castillo, M. Mendoza, and B. Poblete, "Information credibility on twitter," in *Proceedings of the 20th international conference on World wide web*, pp. 675–684, ACM, 2011.

[167] Y.-S. Wu, S. Bagchi, N. Singh, and R. Wita, "Spam detection in voice-over-ip calls through semi-supervised clustering," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pp. 307–316, IEEE, 2009.

[168] M. Kanakaraj and R. M. R. Guddeti, "Performance analysis of ensemble methods on twitter sentiment analysis using nlp techniques," in *Proceedings of the 2015 IEEE 9th International Conference on Semantic Computing (IEEE ICSC 2015)*, pp. 169–170, Feb 2015.

[169] A. Hassan, A. Abbasi, and D. Zeng, "Twitter sentiment analysis: A bootstrap ensemble framework," in *Social Computing (SocialCom), 2013 International Conference on*, pp. 357–364, IEEE, 2013.

[170] S. Giorgis, A. Rousas, J. Pavlopoulos, P. Malakasiotis, and I. Androutsopoulos, "aueb. twitter. sentiment at semeval-2016 task 4: A weighted ensemble of svms for twitter sentiment analysis," *Proceedings of SemEval*, pp. 96–99, 2016.

[171] A. Tsakalidis, S. Papadopoulos, and I. Kompatsiaris, "An ensemble model for cross-domain polarity classification on twitter," in *International Conference on Web Information Systems Engineering*, pp. 168–177, Springer, 2014.

[172] Z. Wang, "The evaluation of ensemble sentiment classification approach on airline services using twitter," 2017.

[173] N. Wiener, *Extrapolation, interpolation, and smoothing of stationary time series*, vol. 7. MIT press Cambridge, MA, 1949.

[174] P. Shaw, D. Greenstein, J. Lerch, L. Clasen, R. Lenroot, N. Gogtay, A. Evans, J. Rapoport, and J. Giedd, "Intellectual ability and cortical development in children and adolescents," *Nature*, vol. 440, no. 7084, pp. 676–679, 2006.

[175] R. E. Kalman, "A new approach to linear filtering and prediction problems," *Transactions of the ASME–Journal of Basic Engineering*, vol. 82, no. Series D, pp. 35–45, 1960.

[176] A. Jain, E. Y. Chang, and Y.-F. Wang, "Adaptive stream resource management using kalman filters," in *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, (New York, NY, USA), pp. 11–22, ACM, 2004.

[177] "Introducing cityhash." https://opensource.googleblog.com/2011/04/introducing-cityhash.html . [Online; April 11, 2011].

[178] FiveThirtyEight, "Uber pickups in new york city." https://www.kaggle.com/fivethirtyeight/uber-pickups-in-new-york-city, 2016.

[179] A. Kapoor and V. Arora, "Application of bloom filter for duplicate url detection in a web crawler," in *Collaboration and Internet Computing (CIC), 2016 IEEE 2nd International Conference on*, pp. 246–255, IEEE, 2016.

[180] M. Weis and F. Naumann, "Dogmatix tracks down duplicates in xml," in *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, (New York, NY, USA), pp. 431–442, ACM, 2005.

[181] "About twitter." http://articles.washingtonpost.com/2013-03-21/business/378893871tweets-jack-dorsey-twitter . [Online; 2015].

[182] A. M. Ortiz, D. Hussein, S. Park, S. N. Han, and N. Crespi, "The cluster between internet of things and social networks: Review and research challenges," *IEEE Internet of Things Journal*, vol. 1, no. 3, pp. 206–215, 2014.

[183] R. Alnashwan, A. O'Riordan, H. Sorensen, and C. Hoare, "Improving sentiment analysis through ensemble learning of meta-level features," in *KDWEB 2016: 2nd International Workshop on Knowledge Discovery on the Web*, Sun SITE Central Europe (CEUR)/RWTH Aachen University, 2016.

[184] D. Mowery, C. Bryan, and M. Conway, "Feature studies to inform the classification of depressive symptoms from twitter data for population health," *arXiv preprint arXiv:1701.08229*, Jan. 2017.

[185] A. H. Wang, "Don't follow me: Spam detection in twitter," in *Security and Cryptography (SECRYPT), Proceedings of the 2010 International Conference on*, pp. 1–10, IEEE, 2010.

[186] S. Hirve and S. Kamble, "Twitter spam detection," *International Journal of Engineering Science*, vol. 2807, 2016.

[187] "test-on-twitter." https://www.kaggle.com/alexanderb/test-on-twitter . [Kaggle; Online].

List of Publications

1) Amritpal Singh and Shalini Batra, "Streamed data analysis using adaptable bloom filter," *Computing and Informatics, Slovak Academy of Sciences*- SCIE Indexed (IF:0.504), 2017 (Published)

2) Amritpal Singh and Shalini Batra, "Ensemble based Spam Detection in Social IoT using Probabilistic Data Structures," *Future Generation Computer Systems, Elsevier*- SCIE Indexed (IF: 3.997), 2017, doi:https://doi.org/10.1016/j.future.2017.09.072.

3) Amritpal Singh and Shalini Batra, "FingerPrint Based Duplicate Detection in Streamed Data," *Computing and Informatics, Slovak Academy of Sciences*- SCIE Indexed (IF:0.504), 2017 (Accepted)