Privacy-Preserving Record Linkage for High Linkage Quality

Sirintra Vaiwsri

A thesis submitted for the degree of
Doctor of Philosophy in Engineering and Computer Science
The Australian National University

Australian
National
University

January 2023

Except where otherwise indicated, this thesis is my own original work.

Sirintra Vaiwsri
24 January 2023

In memory of my grandmothers and my father,
Arporn Mahasuwan, Chirawan Vaiwsri, and Sompong Vaiwsri,
who supported and raised me to be patient and calm, and taught me to live an adequate and simple life.

To my mother, *Yistha Vaiwsri*, who always supports and raises me to be strong, think positive, and brave to overcome obstacles.

Acknowledgments

I would like to express my appreciation to all who support me. First, I would like to express my sincere gratitude to my primary supervisor Professor Dr Peter Christen for his support and valuable guidance in both my academic and personal life. His guidance has helped me in contributing and improving my research, writing this thesis, and work-life balance.

I would like to extend my sincere gratitude to my associate supervisor Dr Thilina Ranbaduge for his support and valuable suggestions which have helped me to improve my research work. I also would like to thank my associate supervisor Professor Dr Rainer Schnell for very helpful feedback on my research. In addition, I would like to thank Associate Professor Dr Kee Siong Ng and Dr Anushka Vidanage whom I co-authored with, for interesting ideas and contributions.

I am grateful to King Mongkut's University of Technology North Bangkok for selecting me to receive a scholarship, and I am grateful to the Royal Thai Government Scholarship, Thailand, which has provided me necessary financial support to complete my study. I am also grateful to the School of Computing, the Australian National University (ANU), and all staff at the ANU for providing various sources of knowledge.

I have been fortunate to work with great people. I would like to thank my colleagues, Dr Anushka Vidanage, Dr Charini Nanayakkara, Nishadi Kirielle, and Asara Senaratne. I would like to thank you for sharing ideas, experiences, and valuable times during my study. I also would like to thank Professor Dr Inger Mewburn, Dr Cally Guerin, Dr Benjamin Kooyman, and everyone in the speaking and writing groups for sharing helpful knowledge to improve my speaking and writing skills.

Pursuing a PhD would be more difficult without the support of the supervisor of my master's degree, Associate Professor Dr Sudsanguan Ngamsuriyaroj, and the research experiences I received from Dr Anuphap Prachumwat, Dr Sissades Tongsima, Dr Jittima Piriyapongsa, and colleagues at the National Center for Genetic Engineering and Biotechnology (BIOTEC), Thailand.

Lastly, I would like to express my appreciation to my family, Yistha Vaiwsri, Vassachon Vaiwsri, Rassarin Surapanthip, Tawan Nowyenpon, and everyone in my family for the support and for always being beside me in every situation since before I started my PhD until now. Without their love and care, it would be very difficult for me to achieve my dream. I also would like to thank all of my friends and Paul James Collins for sharing their thoughts and support.

"Life is like riding a bicycle. To keep your balance, you must keep moving."

— Albert Einstein

"The best way to predict your future is to create it."

— Abraham Lincoln

Abstract

Organisations such as healthcare and financial service providers collect vast amounts of data with varying levels of data quality. Such data often need to be integrated across databases owned by different organisations to facilitate effective and efficient data analysis. Record linkage is one of the processes in data integration that aims to link records in different databases which refer to the same entities. However, much of the data collected by organisations such as research institutes, healthcare, and financial service providers is about individuals. Therefore, privacy-preserving record linkage (PPRL) is required to link records between these databases, while preserving the sensitive information of the individuals in such databases.

In the last two decades, various PPRL techniques have been developed, where the most widely used techniques are based on Bloom filter encoding. However, the Bloom filter technique is vulnerable to various privacy attacks. Therefore, hardening techniques have been proposed to improve the privacy of Bloom filter encoding. Hardening techniques result in lower linkage quality and some have higher computational complexities. Developing effective PPRL techniques, especially in the Big data era, is still an open challenge because (1) the linking of large amounts (volume), diverse data (variety), and data of different qualities (veracity) requires fast linkage processing (velocity), (2) the privacy requirements of sensitive information, and (3) the linkage process should provide high linkage quality to allow data scientists to analyse data accurately.

In this thesis, we develop PPRL techniques to provide high linkage quality where the scalability and privacy of the linking process are also of high importance. We first propose a hardened Bloom filter based PPRL technique to improve the privacy and linkage quality of the original Bloom filter encoding. We then propose a novel PPRL technique to link databases that contain missing values, because missing data can lead to low linkage quality and it has seen limited attention in the record linkage and PPRL contexts. Next, we propose two PPRL techniques that consider positional information of sub-strings to provide efficient and accurate string matching results. Finally, we propose two PPRL techniques to improve the time complexity of the linkage process while providing high linkage quality of string matching results.

We comparatively evaluated our proposed PPRL techniques with various baseline techniques on both real-world and synthetic databases. We assessed our techniques in terms of linkage quality, scalability, and privacy. The experimental results obtained illustrate that all of our proposed PPRL techniques provide higher linkage quality and privacy compared to the baselines. Our string matching techniques provide accurate linkage results and outperform some baselines with regard to scalability for linking larger databases.

List of Publications

Publications contributing for this thesis

1. **Reference values based hardening for Bloom filters based privacy-preserving record linkage**: *Sirintra Vaiwsri*, Thilina Ranbaduge, and Peter Christen. In Australasian Data Mining Conference (AusDM), 2018, Volume 996, Pages 189-202. Springer. (Full Paper) - corresponds to Chapter 4 of this thesis.

2. **Accurate and efficient suffix tree based privacy-preserving string matching**: *Sirintra Vaiwsri*, Thilina Ranbaduge, Peter Christen, and Kee Siong Ng. In arXiv, 2021, doi:10.48550/ARXIV.2104.03018.

3. **Accurate privacy-preserving record linkage for databases with missing values**: *Sirintra Vaiwsri*, Thilina Ranbaduge, Peter Christen, and Rainer Schnell. In Information Systems, 2022, Volume 106, Article 101959. (Journal) - corresponds to Chapter 5 of this thesis.

4. **Accurate and efficient privacy-preserving string matching**: *Sirintra Vaiwsri*, Thilina Ranbaduge, and Peter Christen. In International Journal of Data Science and Analytics, 2022, Pages 1-25. (Journal) - corresponds to Chapter 6 of this thesis.

Minor contributions

1. **Evaluating hardening techniques against cryptanalysis attacks on Bloom filter encodings for record linkage**: Thilina Ranbaduge, Anushka Vidanage, *Sirintra Vaiwsri*, Rainer Schnell, and Peter Christen. In International Journal of Population Data Science, 2018, Volume 3, Number 4.

2. **Attacking and hardening Bloom filter encoding**: Peter Christen, Thilina Ranbaduge, Rainer Schnell, Anushka Vidanage, *Sirintra Vaiwsri* (co-authored book chapter). In Linking Sensitive Data - Methods and Techniques for Practical Privacy-Preserving Information Sharing, Springer, 2020, Chapter 9, Pages 221-251.

3. **Empirical evaluation**: Peter Christen, Thilina Ranbaduge, Rainer Schnell, Anushka Vidanage, *Sirintra Vaiwsri* (co-authored book chapter). In Linking Sensitive Data - Methods and Techniques for Practical Privacy-Preserving Information Sharing, Springer, 2020, Chapter 12, Pages 323-344.

Contents

Acknowledgments vii

Abstract xi

List of contributions xiii

Notation and Terminology xxv

1 Introduction 1
- 1.1 Research Problem . . . 1
- 1.2 Aim of Research . . . 4
- 1.3 Contributions . . . 5
- 1.4 Research Methodology . . . 7
- 1.5 Thesis Outline . . . 8

2 Background 11
- 2.1 Record Linkage . . . 11
- 2.2 Privacy-Preserving Record Linkage (PPRL) . . . 16
  - 2.2.1 Protocols . . . 17
  - 2.2.2 Adversary Models . . . 18
  - 2.2.3 PPRL Techniques . . . 19
    - 2.2.3.1 Secure Multi-Party Computation . . . 19
    - 2.2.3.2 Perturbation . . . 21
- 2.3 Bloom Filter Encoding in PPRL . . . 23
  - 2.3.1 Hashing Technique for Bloom Filters . . . 23
    - 2.3.1.1 Double Hashing . . . 24
    - 2.3.1.2 Random Hashing . . . 24
  - 2.3.2 Bloom Filters Encoding Techniques . . . 24
    - 2.3.2.1 Attribute-level Bloom Filter (ABF) . . . 25
    - 2.3.2.2 Cryptographic Longterm Key (CLK) . . . 25
    - 2.3.2.3 Record-level Bloom Filter (RBF) . . . 26
    - 2.3.2.4 Hybrid Encoding (CLKRBF) . . . 28
  - 2.3.3 Attacking Techniques for Bloom Filters . . . 28

2.3.4 Hardening Techniques . . . 29
2.3.4.1 Salting . . . 29
2.3.4.2 XOR-folding . . . 29
2.3.4.3 Balancing . . . 30
2.3.4.4 BLoom and FlIP . . . 30
2.3.4.5 Rule 90 . . . 31
2.3.4.6 Markov Chaining . . . 31
2.3.4.7 Windowing based XORing . . . 32
2.3.4.8 Re-sampling based XORing . . . 33
2.4 The Relationship of Big Data with Record Linkage and PPRL . . . 33
2.5 Evaluation . . . 35
2.5.1 Linkage Quality . . . 35
2.5.2 Scalability . . . 36
2.5.3 Privacy . . . 37
2.6 Experimental Setup . . . 39
2.7 Chapter Summary . . . 40

3 Related Work **41**
3.1 PPRL for String Matching . . . 41
3.2 Record Linkage and PPRL for Sub-string Matching . . . 43
3.2.1 Suffix Tree and Array based Approaches . . . 43
3.2.2 Homomorphic Encryption based Approaches . . . 44
3.3 Record Linkage and PPRL for Fast String and Sub-string Matchings . . . 46
3.3.1 Blocking (and Clustering) Techniques for Fast String Matching . . . 46
3.3.2 Tree based Approaches for Fast String Matching . . . 48
3.3.3 Q-gram based Approaches for Fast String Matching . . . 49
3.4 PPRL for Matching Numerical Values . . . 50
3.5 Reference Values used in PPRL Approaches . . . 52
3.6 Record Linkage and PPRL for Linking Databases with Missing Values . . . 54
3.7 Chapter Summary . . . 57

4 Reference Values based Hardening for Bloom Filter based PPRL **59**
4.1 Introduction . . . 59
4.2 BLoom and FlIP (BLIP) . . . 61
4.3 Reference based BLIP BF Hardening (RBBF) Protocol . . . 61
4.3.1 Selecting Reference Values . . . 62
4.3.2 Reference Values based BLIP BF Hardening . . . 63
4.4 Analysis . . . 65
4.4.1 Linkage Quality Analysis . . . 65

4.4.2 Scalability Analysis . . . 65
4.4.3 Privacy Analysis . . . 66
4.5 Experimental Evaluation . . . 66
4.5.1 Datasets and Experimental Setup . . . 67
4.5.2 Linkage Quality Results . . . 67
4.5.3 Scalability Results . . . 71
4.5.4 Privacy Results . . . 72
4.6 Chapter Summary . . . 77

5 Accurate PPRL for Databases with Missing Values 79
5.1 Introduction . . . 79
5.2 Missing Data . . . 81
5.3 Lattice Structure to Represent Missingness Patterns . . . 82
5.4 Protocol Overview . . . 83
5.5 Encoding and Generating Missing Pattern . . . 85
5.5.1 Attribute-level BF Generation . . . 86
5.5.2 Missing Pattern Table Generation . . . 86
5.5.3 Lattice Structure and Common Pattern Generations . . . 87
5.5.4 Grouping Missing Patterns into Partitions . . . 89
5.6 Linkage Process . . . 92
5.6.1 Iterative Linkage . . . 92
5.6.2 Batch Linkage . . . 94
5.6.3 Record Pair Comparison by the Linkage Unit . . . 96
5.7 Analysis . . . 96
5.7.1 Linkage Quality Analysis . . . 97
5.7.2 Scalability Analysis . . . 98
5.7.3 Privacy Analysis . . . 99
5.8 Experimental Evaluation . . . 101
5.8.1 Generating Datasets with Missing Values . . . 101
5.8.2 Experimental Setup . . . 102
5.8.3 Linkage Quality Results . . . 104
5.8.4 Scalability Results . . . 107
5.8.5 Privacy Results . . . 109
5.9 Chapter Summary . . . 111

6 Accurate and Efficient Privacy-Preserving String Matching 113
6.1 Introduction . . . 113
6.2 Protocol Overview . . . 116
6.3 Shifting Hashed Q-grams based Approach . . . 118

6.3.1 Q-grams Generation . . . 118
6.3.2 Hashing of Q-grams and Shifting Hash Encoded Q-gram Lists . 119
6.3.3 Comparison of Shifted Hash Encoded Q-gram Lists . . . 120
6.3.3.1 Basic Shifted Hash Encoded Q-grams Comparison . . . 120
6.3.3.2 Fast Shifted Hash Encoded Q-grams Comparison . . . 121
6.4 Bit Array Based Approach . . . 123
6.4.1 Generating Bit Arrays for Strings . . . 123
6.4.2 Comparison of Bit Arrays . . . 127
6.4.2.1 Basic Bit Arrays Comparison . . . 127
6.4.2.2 Fast Bit Arrays Comparison . . . 128
6.5 LCS Length Calculation . . . 130
6.6 Scalability Aspect . . . 130
6.7 Analysis . . . 131
6.7.1 Linkage Quality Analysis . . . 132
6.7.2 Scalability Analysis . . . 133
6.7.3 Privacy Analysis . . . 134
6.8 Experimental Evaluation . . . 135
6.8.1 Datasets . . . 135
6.8.2 Experimental Setup . . . 136
6.8.3 Linkage Quality Results . . . 140
6.8.4 Scalability Results . . . 141
6.8.5 Privacy Results . . . 144
6.9 Chapter Summary . . . 149

7 PPRL for Fast and High Linkage Quality 151
7.1 Introduction . . . 151
7.2 Comparison of Ciphertexts . . . 153
7.2.1 Comparison of Special 1- and 0-Encodings . . . 153
7.2.2 Comparison of Packs of Encryptions . . . 154
7.3 Protocol Overview . . . 155
7.4 Data Generation . . . 160
7.5 One-to-One Approach . . . 163
7.5.1 One-to-One Encryption . . . 163
7.5.2 One-to-One Comparison . . . 166
7.6 Many-to-One Approach . . . 167
7.6.1 Many-to-One Encryption . . . 167
7.6.2 Many-to-One Comparison . . . 168
7.7 LCS Length Selection . . . 169

7.8 Analysis . . . 169
7.8.1 Linkage Quality Analysis . . . 170
7.8.2 Scalability Analysis . . . 171
7.8.3 Privacy Analysis . . . 172
7.9 Experimental Evaluation . . . 175
7.9.1 Datasets and Experimental Setup . . . 175
7.9.2 Linkage Quality Results . . . 176
7.9.3 Scalability Results . . . 178
7.9.4 Privacy Results . . . 183
7.10 Chapter Summary . . . 185

8 Conclusion and Future Research Directions 187
8.1 Summary of Research Problems . . . 187
8.2 Summary of Contributions . . . 188
8.3 Future Research Directions . . . 190
8.4 Conclusion . . . 192

References 193

List of Figures

1.1 The methodology of this thesis. 7

2.1 The building blocks of record linkage and PPRL. 12
2.2 Dice-coefficient similarity calculation between two Bloom filters. 23
2.3 Example of cryptographic longterm key Bloom filter of a record. 25
2.4 Example of record-level Bloom filter of a record. 27

3.1 Example of k-nearest neighbour based Bloom filters comparison. 56

4.1 Overview of our Reference value based BLIP Bloom Filter Hardening. . 62
4.2 Classification outcomes of a Bloom filter pair. 68
4.3 Re-identification results of RBBF with $s_t = 0.4$ and $k = 1$, and baselines. 73
4.4 Re-identification results of RBBF with $s_t = 0.4$ and $k = 3$, and baselines. 74
4.5 Re-identification results of RBBF with $s_t = 0.8$ and $k = 1$, and baselines. 75
4.6 Re-identification results of RBBF with $s_t = 0.8$ and $k = 3$, and baselines. 76

5.1 Example database with missing values and its lattice structure. 82
5.2 Overview of our proposed PPRL for databases with missing values. . . 83
5.3 Record Bloom filter generation for records with missing values. 85
5.4 Lattice structures with at least one common pattern between databases. 88
5.5 Lattice structures with missing layer in the common lattice structure. . . 88
5.6 Precision-recall plots of the iterative and batch linkage methods. 105
5.7 Precision-recall plots for our MVBF approach compared to baselines. . 106
5.8 Re-identification results for our MVBF approach compared to baselines. 110

6.1 Overview of our privacy-preserving string matching approaches. 116
6.2 Two example bit arrays with different random bits padded. 124
6.3 The basic bit arrays comparison. 128
6.4 The fast bit arrays comparison. 129
6.5 Similarity plots for real-world dataset pairs of data type letter. 138
6.6 Similarity plots for real-world dataset pairs of data type digit. 139
6.7 Similarity plots for synthetic dataset pairs of data type digit and mixed. 140
6.8 Runtimes of the processes by a DO of our string matchings and baselines. 142

6.9 Runtimes of comparisons by a LU of our string matchings and baselines. 143
6.10 Heatmap plots of different privacy levels of a PPRL approach. 144
6.11 Heatmap plots of real-world datasets of data type letter. 146
6.12 Heatmap plots of real-world datasets of data type digit. 147
6.13 Heatmap plots of synthetic datasets of data type digit and mixed. . . . 148

7.1 Example of the length of LCS calculation between strings. 156
7.2 Overview of our PPRL for fast and high linkage quality. 157
7.3 Example of comparisons based on references and lengths of sub-strings. 159
7.4 Runtimes per sub-string/pair of one-to-one, many-to-one, and baselines. 180
7.5 Runtimes of the whole dataset of one-to-one, many-to-one, and baselines. 181

List of Tables

1.1 Relationship between the challenges of PPRL and Big data. 2
1.2 Relationship between our research aims and the challenges of PPRL. . . 4

2.1 Transformation table of Soundex encoding. 13
2.2 Bits transformation of Rule 90. 31
2.3 Example of frequent co-occurrence q-grams in a large set **g**. 32

3.1 Example table of the encryptions of special 1-encodings. 51

4.1 Notation and terminology used in Chapter 4. 60
4.2 Precision and recall of RBBFs and baselines for attribute first name. . . 68
4.3 Precision and recall of RBBFs and baselines for attribute last name. . . . 69
4.4 Precision and recall of RBBFs and baselines for attribute street address. 70
4.5 Precision and recall of RBBFs and baselines for attribute city. 71
4.6 Numbers of reference values to be used in our RBBF approach. 72
4.7 Average runtimes for BLIPs and RBBFs. 72

5.1 Notation and terminology used in Chapter 5. 80
5.2 Example of records with patterns $m_C = 1111$ and $m_N = 1011$. 100
5.3 Numbers of records with missing values in different attributes. 102
5.4 Missingness patterns and their numbers of records in the dataset pairs. 103
5.5 Numbers of records of the MCAR datasets in the batch linkage method. 108
5.6 Average runtimes and standard deviation of our MVBF and baselines. . 108

6.1 Existing privacy-preserving string matching techniques. 114
6.2 Example string pairs and their edit distance similarities. 114
6.3 Notation and terminology used in Chapter 6. 115
6.4 Example strings and their corresponding randomly shifted q-gram lists. 119
6.5 Minimum and estimated lengths of bit arrays. 124
6.6 Lengths of the longest q-gram lists and bit arrays of different datasets. . 136
6.7 Numbers of bits used for generating blocks for different datasets. 137
6.8 Numbers of comparisons of our string matchings and baselines. 142

7.1 Notation and terminology used in Chapter 7. 152

7.2 Example of comparison results based on our one-to-one approach. . . . 165
7.3 Example of comparison results based on our many-to-one approach. . . 168
7.4 Percentages of errors in one-to-one and many-to-one approaches. 176
7.5 Precision and recall for blocks generated using Equation 7.10. 177
7.6 Precision and recall for blocks generated using Christen et al. 178
7.7 Numbers of blocks generated using Equation 7.10 and Christen et al. . 179
7.8 Numbers of comparisons for one-to-one, many-to-one, and baselines. . 179
7.9 Privacy measures based on blocks generated using Equation 7.10. . . . 183
7.10 Privacy measures based on blocks generated using Christen et al. 184

Notation and Terminology

Notation

$\mathbf{D}, \mathbf{D}_A, \mathbf{D}_B$	Database, database A, and database B to be encoded, respectively
x, y	Value, string, or number
$\mathbf{E}$	Encoded version of a database $\mathbf{D}$
$\mathbf{G}$	Global database or set
$\mathbf{q}$	List or set of q-grams
q	Length of each q-gram
b	Bloom filter, bit array, or binary string
l	Length of Bloom filter, bit array, or bit string
$\|...\|$	Length of list or set, or size of inverted index, e.g. $\|\mathbf{D}\|$ is the size of a database $\mathbf{D}$.
n_c	Number of common q-grams or bits
k, n	Number of a given value
m	Minimum number or length
i, j	Index position
r	Random or record value
t	Threshold
s_t	Similarity threshold
w	Weight
w_a	Weight of agreement
w_d	Weight of disagreement
bkv	Blocking key value

Terminology

PPRL	Privacy-preserving record linkage
SMC	Secure multi-pary computation
DO	Database owner
LU	Linkage unit
BF	Bloom filter
ABF	Attribute-level Bloom filter
RBF	Record-level Bloom filter
BLIP	BLoom and flIP hardening Bloom filter
LCS	Longest common sub-string

Chapter 1

Introduction

In this chapter, we provide an introduction to the work presented in this thesis. We describe different privacy-preserving record linkage (PPRL) techniques to address research problems and challenges in PPRL with the aim of providing high linkage quality. In Section 1.1, we introduce the research problems and challenges in the PPRL context that need to be addressed. In Section 1.2, we define the research aims of this thesis. In Section 1.3, we describe the research contributions related to our aims of research, and in Section 1.4 we describe the research methodology that we use in our contributions. Lastly, in Section 1.5, we provide an outline of the thesis.

1.1 Research Problem

Many organisations, such as businesses, industries, and research institutes, collect a large amount of data into their databases [35, 198], where the amount of accumulated data increases every day [150]. *Big data* refers to a collection of large volumes of data that cannot be efficiently processed by traditional database methods and tools [90]. To facilitate effective and efficient Big data analysis, data characteristics and appropriate technologies need to be considered [90, 198]. Big data often need to be integrated across different organisations [90], where much of the data that are collected by each organisation are about individuals, such as customers, patients, and taxpayers [185]. Therefore, privacy is often of concern when processing these sensitive data, such as during the data integration process [33, 38].

The traditional data integration process includes three main components, which are schema alignment, *record linkage*, and data fusion [33, 55]. Schema alignment considers sets of different data schemas that are in the same domain [33, 55]. Record linkage aims to link records in different databases that refer to the same entities [33]. Data fusion processes records that are classified as matches into consistent and accurate information [33] by merging data into a single and clean data representation using methods such as data join and union [15].

Big data integration (BDI) [55] differs from traditional data integration due to the consideration of the four main characteristics of Big data, known as the four Vs, which are *volume* (large amounts of data), *velocity* (the speed of data processing), *va-*

Table 1.1: The relationship between the three major challenges of PPRL and the four Vs of Big data, where $***$ means a certain V is highly relevant, $**$ means a V is relevant, and $*$ means a V is somewhat relevant to a given challenge.

	Linkage quality	Scalability	Privacy
Volume	$**$	$***$	$***$
Velocity	$*$	$***$	$***$
Variety	$***$	$*$	$***$
Veracity	$***$	$*$	$***$

riety (heterogeneous data), and *veracity* (diverse data qualities) [35, 54, 55]. However, both traditional data integration and Big data integration require high quality record linkage techniques [38, 55].

This thesis focuses on the *record linkage* component of data integration with the consideration of privacy, known as *privacy-preserving record linkage* (PPRL) [33, 184]. PPRL aims to link records of individuals in different databases without revealing sensitive information of the individuals represented by these records [33, 184].

Many PPRL techniques have been developed in recent decades [38, 57, 184]. PPRL techniques generally encode or encrypt a set of attribute values in each record in a database and then link these records based on their encoded or encrypted attribute values [184]. These techniques can be categorised into two categories. The first category is secure multi-party computation (SMC) [115] based techniques which perform matching of encrypted records, but they often have high computational costs. The second category is perturbation based techniques [115, 185] which encode the attribute values by using an appropriate encoding technique.

There are three major challenges in PPRL, which are *linkage quality*, *scalability*, and *privacy* [185]. These three challenges are also related to the four Vs of Big data, as illustrated in Table 1.1. Therefore, we need to consider the three challenges that are the basis of developing efficient PPRL techniques which should be applicable to large volumes of data. In this thesis, we mainly focus on the *linkage quality* aspect of the three challenges of PPRL, where *scalability* and *privacy* are also considered.

1. **Linkage quality**
   Linkage quality relates to volume, variety, and veracity in the four Vs of Big data. It refers to the effectiveness of a process that links records between two or more databases [38]. The linking process is achieved by similarity functions which can be based on exact or approximate comparisons [33, 36]. These functions provide weight vectors that contain numerical values of the compared record pairs to be classified as matches, non-matches, or potential matches [33, 36, 184]. The potential matches often need a clerical review process which is processed by a domain expert to classify these potential matches record pairs as matches or non-matches [38].

It is challenging to achieve high linkage quality in the PPRL context because the databases to be linked possibly contain missing attribute values, heterogeneous data, and different data types such as string and numerical data [38, 76, 181]. Therefore, a developed PPRL technique should provide high linkage quality even when the databases contain incomplete data, such as missing values. Consequently, it is important to develop PPRL techniques to provide accurate linkage results.

2. **Scalability**
   Scalability is challenging in the PPRL context. This is because many organisations require efficient PPRL techniques to manage a large amount of data (volume) and provide high computing performance to speed up computation time (velocity), where some organisations require real-time data analysis [96].

   Various techniques have been proposed for improving the efficiency of PPRL which can be categorised into three types. The first type is blocking techniques [185] which aim to reduce the comparison space by grouping similar records into blocks and comparing only those records in the same blocks. The second type is filtering techniques [185] which utilise different approaches to eliminate sets of records that cannot meet a constraint such as a similarity threshold (of being classified as matches) to reduce the number of record pairs to be compared. The third type is parallelisation (on multiple processors) based techniques [37, 185] which aim to improve the computation time by concurrently comparing record pairs on different processors.

   However, the PPRL techniques that require high computation costs, such as SMC based techniques [115], are currently not suitable for large volumes of data due to the computation complexities involved in encryption and decryption processes [38]. Therefore, efficient PPRL techniques are still required.

3. **Privacy**
   Privacy is the most challenging aspect in the PPRL context compared to linkage quality and scalability because sensitive information must not be revealed and a PPRL method must be resilient to attacks [33, 184]. Privacy often creates a trade-off in the linkage process [33] as any PPRL technique that provides high degrees of privacy can result in low linkage quality, and it possibly has high computational requirements [185] which slow down the computation time. Privacy is also highly relevant to all of the four Vs of Big data because (1) linking large amounts of data (volume) from multiple databases (variety) often requires blocking techniques (velocity) where privacy of the individual records must be preserved in the blocking process, and (2) encoding techniques can be more complex and complicate when linking databases that have different data quality (veracity). Therefore, developing PPRL techniques for improving privacy without substantial additional computational requirements and providing accurate linkage quality is challenging.

Table 1.2: The relationship between the aims of our research and the three challenges of PPRL, where $***$ means a certain challenge is highly relevant, $**$ means a challenge is relevant, and $*$ means a challenge is somewhat relevant to our aims.

	Linkage quality	Scalability	Privacy
PPRL hardening for Bloom filters	$***$	$**$	$**$
PPRL for databases with missing values	$***$	$*$	$**$
PPRL for accurate sub-string matching	$***$	$*$	$***$
PPRL for fast string matching	$***$	$***$	$***$

1.2 Aim of Research

We aim to develop PPRL techniques to provide high linkage quality where the scalability and privacy challenges of PPRL are also considered. Table 1.2 shows the relationship between the aims of our research and the three challenges of PPRL.

1. **Developing a PPRL technique for improving the privacy and linkage quality of Bloom filter (BF) encoding**
   Bloom filter encoding [155] is one of the perturbation based techniques in the PPRL context. A BF is a bit vector generated by encoding attribute values using a set of hash functions, as we describe in more detail in Chapter 2. BF encoding is widely used in PPRL because it is easy to implement, supports approximate string matching, and does not have high computational costs. However, several researchers have shown that BF encoding is vulnerable to cryptanalysis attacks [40, 42, 108, 109], and therefore different hardening techniques have been proposed to improve the degrees of privacy of BF encoding [5, 157, 158].

   However, some hardening techniques can be attacked [40] by identifying the frequency distribution of 1-bit positions in the hardened BFs, and some techniques lower the linkage quality of BF based PPRL [5, 157]. Therefore, we aim to develop a hardening technique for PPRL to improve the degrees of privacy of BF encoding and provide higher linkage quality compared to the existing hardening techniques [5, 157].

2. **Developing a PPRL technique for databases that contain missing values**
   Most real-world data collected by organisations include errors, such as missing and mistyped values [33]. These errors in data can lead to low linkage quality when such data are used in the linkage process [33, 184]. Therefore, efficient PPRL techniques are required to provide high linkage quality when the databases involved in the linkage process contain missing values.

   In the bioinformatics domain, issues with missing values in gene expression can be addressed by imputation using singular value decomposition (SVD) or imputation using the *k*-nearest neighbours (k-NNs) based method [84, 173]. The k-NN based method has also been proposed in the PPRL context by apply-

ing the method on attribute-level BFs (ABFs) [30], as we describe in Chapter 2. However, ABFs have been shown to be vulnerable to several cryptanalysis attacks [39, 109, 129]. Therefore, we aim to develop a PPRL technique to provide accurate linkage results while privacy is still preserved.

3. **Developing PPRL techniques for accurate sub-string matching**
   The matching sequence order of characters in a pair of strings (sub-string matching) is required in some application domains, such as healthcare and financial service providers [38]. For example, the matching of credit card numbers between financial service providers requires a matching sequence of numbers on a credit card[1], where each sub-sequence of numbers identifies the card type, card company, account number, and verification number.

   Most PPRL techniques that are developed for string matching, such as BFs [155], calculate an overall similarity between two strings where the sequence of information is lost. Therefore, unless the two strings in a pair are exactly matched, BFs [155] cannot identify if, for example, two credit card numbers are registered by the same bank or the credit card company. Hence, we aim to develop PPRL techniques to provide accurate string matching results where no sensitive information is revealed to any party participating in the linkage process.

4. **Developing PPRL techniques for fast and high linkage quality of string matching**
   Some organisations require fast or real-time data analysis [96]. For example, the linkage of crime, immigration, and airline databases for reporting criminal activities needs a fast response for police to take action in real-time.

   Some techniques, such as tree-based and blocking techniques, have been proposed for speeding up the computation time of PPRL [96, 97, 138]. However, these techniques possibly reveal sensitive information and can result in low linkage quality between databases involved in the linking process. Therefore, we aim to develop PPRL techniques to provide fast and high linkage quality while privacy is still preserved.

1.3 Contributions

According to the aims of our research, we propose PPRL techniques as following:

1. **A PPRL hardening technique for Bloom filters**
   As discussed in Section 1.2, the BF encoding approach for PPRL is vulnerable to cryptanalysis attacks [40, 42, 108, 109] and various hardening techniques have been proposed to improve the degrees of privacy of BF encoding [5, 157, 158]. However, some of the proposed hardening techniques are still vulnerable to privacy attacks while some techniques lower the linkage quality [5, 157].

[1] https://www.creditcards.com/credit-card-news/credit-card-appearance-1268/

We propose a novel technique to improve the degrees of privacy and linkage quality of hardened BFs by adapting the hardening technique called BLoom and flIP (BLIP) [5, 157] to BFs. The BLIP hardening technique [5, 157] randomly flips bits in BFs with a given flip probability, as we discuss in Chapters 2 and 4. However, the BLIP technique [5, 157] can lower the linkage quality of BF. Therefore, our proposed technique also improves the linkage quality of the BLIP hardening technique by using reference values from a publicly available database [132]. We discuss our proposed technique in detail in Chapter 4.

2. **A PPRL technique for databases with missing values**
   The databases to be linked can contain missing values. These missing values can cause false positive and false negative matching results, which lead to low linkage quality [76]. We propose a novel PPRL technique to link records that contain missing values by applying a lattice structure [82] (as we describe in Chapter 5) and record-level BFs (RBF) [59] (as we explain in detail in Chapter 2).

   Our proposed technique generates a lattice structure based on the missing value patterns that occur in databases to be linked, then uses a secure set intersection protocol [49, 53] to find common and not common missingness patterns between these databases. After that, one BF of each record in each database is generated. The generated BFs corresponding to not common missing patterns are grouped into BFs corresponding to common missing patterns based on a weight calculation. Each group of BFs of the databases to be linked are then compared to find matches using a linkage unit which is a third party that can conduct the linkage between different databases, as we describe in Chapter 2. We describe our proposed technique in detail in Chapter 5.

3. **PPRL techniques for accurate sub-string matching**
   Some domains, such as financial and bioinformatics, require the positions of sub-strings that occur in strings to find the longest common sub-strings between databases to be linked in the linkage process [79, 80]. However, most PPRL techniques, such as BF encoding [155], often provide approximate similarity calculations between strings or textual values where the position of sub-strings is not taken into account.

   We propose two novel PPRL techniques to find the longest common sub-strings between linked databases, where each technique provides accurate linkage results without revealing sensitive information to any party involved in the linkage process. The first technique generates a list of q-grams (sub-strings of length q characters) for each string, and then randomly shifts the q-grams in these generated lists to hide the actual positions of the q-grams in a string before the lists of q-grams are being compared. The second technique extends the first technique by generating bit arrays and hiding the actual positions of bits by randomly padding the generated bit arrays with random bits, resulting in final bit arrays of fixed length which improves the privacy of these generated bit arrays. We discuss our proposed techniques in detail in Chapter 6.

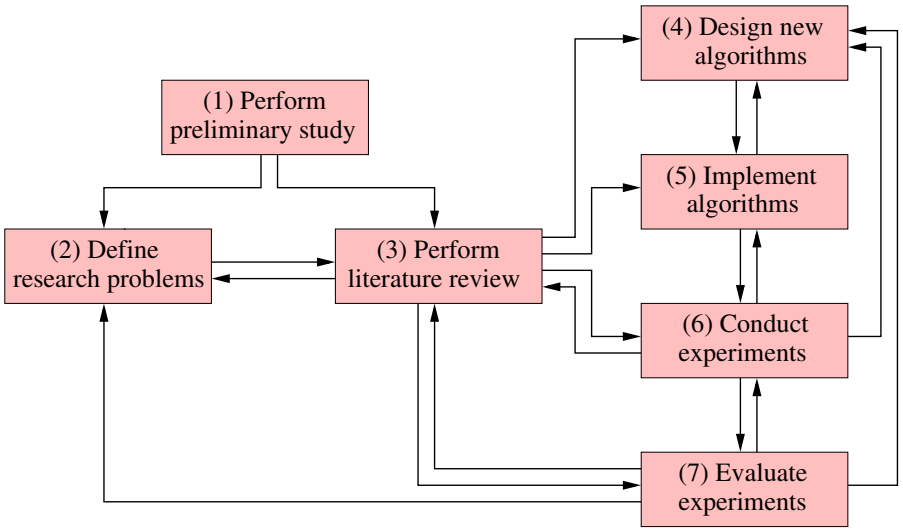

Figure 1.1: The methodology of this thesis.

4. **PPRL techniques for fast and high linkage quality of string matching**
   As described in Section 1.2, some organisations require PPRL techniques that can provide fast string matching results [96]. We propose two novel PPRL techniques that preserve privacy and provide fast and high linkage quality of string matching results based on the length of the longest common sub-string between strings in a pair. In our proposed techniques, we first generate blocks and references based on sub-strings that occur in strings in a database to be linked and a global database. We then calculate the length of sub-strings in strings. These calculated lengths will be encrypted into ciphertexts and then compared.

   The first technique compares lists of hash values of calculated lengths that correspond to the same sub-string, where each list of hash values is generated based on special 1- or 0-encodings [114]. We then encrypt each calculated length into a ciphertext by using a Paillier homomorphic encryption scheme [131]. The second technique generates a list of the calculated lengths and encrypts it into one ciphertext by using a packing method [26]. The ciphertexts of the two databases are then compared. We discuss our proposed techniques in Chapter 7.

1.4 Research Methodology

The methodology for designing and developing our proposed PPRL techniques in this thesis involves seven steps as illustrated in Figure 1.1 and described below:

1. **Perform preliminary study**
   The first step is to study the background and related concepts of PPRL. These concepts include record linkage, existing PPRL techniques, sub-string matching

for PPRL, PPRL for databases that contain missing values, PPRL for fast string matching, privacy attacks on PPRL, and other relevant concepts.

2. **Define research problems**
   The second step of the methodology is to define the research problems and set the boundaries of the defined problems.

3. **Conduct literature review**
   We study the literature related to PPRL techniques that can address the defined research problems. The literature includes Bloom filters, hash functions, hardening techniques, approximate matching, missing data matching, longest common sub-string matching, blocking techniques, and ciphertexts comparison. We use some of the existing PPRL approaches as baselines for the proposed PPRL techniques in this thesis.

4. **Design new PPRL techniques and algorithms**
   The fourth step is to design novel PPRL techniques to solve the problems defined in step 2. The designed techniques are mainly aimed to improve the linkage quality with the consideration of the privacy and scalability challenges of PPRL, as we described in Sections 1.2 and 1.3.

5. **Implement the designed algorithms**
   We implement the designed PPRL techniques in this step. Furthermore, we also implement the baseline techniques which are used to evaluate the performance compared to our designed PPRL techniques.

6. **Conduct experiments**
   This step involves collecting real datasets and generating synthetic datasets that are suitable for evaluating our PPRL techniques. These datasets are then used with the implemented algorithms to evaluate our PPRL techniques against different baseline techniques. In this step, we also fix any errors that could occur in our implementation, and we collect the experimental results.

7. **Evaluate the experimental results**
   Once all experimental results are collected, we evaluate the proposed PPRL techniques and the baselines by measuring the linkage quality, scalability, and privacy performance.

1.5 Thesis Outline

In Chapter 2, we describe the background and related concepts that are relevant to the research proposed in this thesis. In Chapter 3, we then discuss related research works that are relevant to this thesis. In Chapter 4, we describe and evaluate our proposed hardening technique for Bloom filter based PPRL, and in Chapter 5 we then discuss and evaluate our proposed PPRL technique for linking databases that contain missing values. In Chapter 6, we describe the protocol and discuss the evaluation

results of our proposed PPRL techniques for accurate and efficient PPRL sub-string matching. In Chapter 7, we then describe our proposed PPRL techniques for fast and high linkage quality of string matching. Finally, in Chapter 8, we summarise all of our contributions and discuss directions for future work.

Chapter 2

Background

This chapter describes background knowledge that relates to our research problems. In Section 2.1, we describe record linkage and its steps. In Section 2.2, we then discuss privacy-preserving record linkage (PPRL) including protocols, adversary models, and techniques. In Section 2.3, we describe Bloom filter based approaches as they have been widely used in PPRL. In Section 2.4, we discuss Big data and its challenges that relate to the three challenges of PPRL on which we focus in this thesis. In Section 2.5, we discuss the descriptions of evaluation techniques that are used in the record linkage and PPRL contexts. In Section 2.6, we then discuss the experimental setup including datasets and implementation environment that are used for evaluating our proposed PPRL techniques in this thesis. Finally, in Section 2.7, we provide a summary of this chapter.

2.1 Record Linkage

The term *record linkage* was first introduced by Dunn in 1946 [56] to describe an assembly of events for each individual in their lifetime, called a *book of life*. Such a book consists of the principal events of life such as birth, health, marriage, and death records of a person. These pieces of information can be used to improve national statistics, healthcare services, or the identification of individuals. However, the data quality of such records has been recognised by Dunn [56] as a challenging issue.

In 1959, Newcombe et al. [128] proposed an automated record linkage method by using computers to link marriage records and birth records. They applied phonetic encoding [33] to the attribute last name in records to tackle variation of last name values, and used probability calculations to classify record pairs as matches or non-matches. Later, Fellegi and Sunter [67] formalised the idea of probabilistic record linkage and proved that using the assumption of conditional independence between attributes can lead to an optimal decision function. Their approach is still the basis of many record linkage techniques and is widely used today. For example, in 2021 Xu et al. [197] proposed a record linkage technique that applied the Fellegi and Sunter [67] approach to adjust the frequency of agreeing attributes for classifying record pairs.

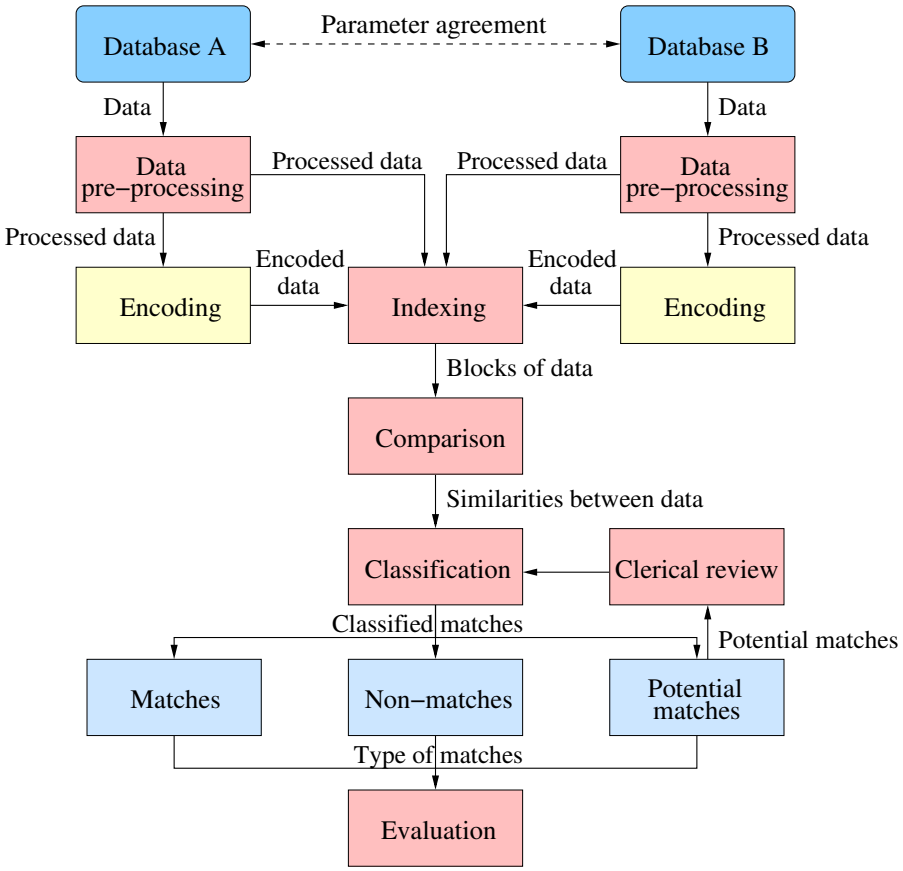

Figure 2.1: The building blocks of record linkage and PPRL (adapted from Christen [33]), where the red boxes are common steps while the yellow boxes are only applicable in the PPRL process. The blue boxes are the two databases being linked and the light blue boxes are classified matches, non-matches, or potential matches.

The record linkage process includes data cleaning and standardisation (data preprocessing), indexing, comparison, classification, and evaluation [33, 184], as we illustrated in Figure 2.1, where the encoding step is only processed in the PPRL linkage process, as we discuss in Section 2.2.

1. **Data cleaning and standardisation**
   Records in databases often contain data errors such as missing and mistyped attribute values [33, 184]. Such data errors can lead to low linkage quality in the linkage process [38]. Therefore, appropriate data cleaning and standardisation techniques are required [33, 184]. These techniques are, for example, (1) removing unwanted attribute values because some of them will not be used in the linkage process, (2) converting attribute values into standardised forms, and (3) verifying the correctness of attribute values [33]. For details we refer the reader to the articles by Rahm and Do [137], and Randall et al. [149].

Table 2.1: Transformation table of Soundex encoding [33].

Character	Transformed character
a, e, h, i, o, u, w, y	0
b, f, p, v	1
c, g, j, k, q, s, x, z	2
d, t	3
l	4
m, n	5
r	6

2. **Indexing**
The naive process of comparing all record pairs between two databases, $\mathbf{D}_A$ and $\mathbf{D}_B$, requires $|\mathbf{D}_A| \times |\mathbf{D}_B|$ comparisons [184], where $|...|$ denotes the number of records in each database. Indexing, also known as *blocking*, in the linkage process aims to reduce the number of record pair comparisons [33, 38]. The basic indexing technique, called standard blocking, uses the concept of a blocking key value, *bkv*, to group similar records into the same blocks, such that records that are in different blocks will not be compared [33, 184]. However, it is a naive technique due to errors and variations causes matching record pairs to be inserted into different blocks, resulting in records that refer to the same entity not being compared [184]. Other indexing techniques are, for example, phonetic encoding functions, sorted neighbourhood indexing, and q-gram based indexing [33].

   Phonetic encoding functions group similar records into the same blocks based on the pronunciation of values in an attribute [33]. A popular phonetic encoding function is Soundex [33, 184]. It encodes a string of an attribute value by keeping the first letter and converting the remaining letters to characters of numbers according to a transformation table (as illustrated in Table 2.1), where all zero digits and repetitions in the transformed code will be removed. The final code is one letter followed by three digits, where the code will be extended if it has less than three digits and truncated if it has more than three digits. For example, a string of an attribute first name is "*albert*". It is first transformed to "*a*41063" which is converted to "*a*4163" and then truncated to "*a*416", while the string "*mary*" is transformed to "*m*060" which is converted to "*m*6" and then expanded to "*m*600".

   Sorted neighbourhood indexing [32, 85] generates keys of records in databases to be compared, where these keys can be values of an attribute or concatenated values of multiple attributes in records. The records in databases are then sorted based on the generated keys and a sliding window of fixed size is then applied to group records into windows. Finally, the records in the same window are compared.

Q-gram based indexing technique [31, 32, 33] generates blocks of records based on lists of q-grams, where each q-gram is a sub-string of length q characters. For each record in a database, an attribute value of a record is converted into a list of q-grams, $\mathbf{q}$, which is then used to generate sub-lists by using a sliding window, where each sub-list contains at least $n = max(1, \lfloor |\mathbf{q}| \times t \rfloor)$ q-grams, where t is a given threshold, $|\mathbf{q}|$ denotes the length of $\mathbf{q}$, and $\lfloor ... \rfloor$ denotes rounding to the next lower integer [31, 32]. The q-grams in each sub-list are then concatenated into a string and this string is being used as an index key. Therefore, each record is inserted into several blocks depended upon its generated index keys and different records that correspond to the same index keys (sharing the same sub-lists of q-grams) are inserted into the same blocks [33]. The records in the same blocks are then used as candidate record pairs for comparison.

To conduct this indexing (blocking) step in PPRL, the privacy of individuals represented by the records are needed to be ensured. There are many other indexing techniques that have been proposed. For details we refer the reader to the survey by Li et al. [113].

3. **Comparison**
Comparing candidate record pairs can be either at the attribute or the record level [184]. Attribute level is the comparison of values in an individual attribute between records in a pair, while record level compares multiple selected attribute values between records in a pair. Comparison techniques can either rely on exact or approximate matching [33, 184]. Exact comparison returns a similarity value of 1 when the two strings of records in a pair are the same character by character (in PPRL, a pair of records have the same encoded value), otherwise, it returns a similarity value of 0. Therefore, when records in a pair contain errors and variations, exact comparison can result in false negatives [33]. To overcome this problem, an approximate comparison can be used [38].

Approximate comparison measures how similar the values of records in a pair are [33, 184]. It returns a similarity value of 0 when two record values are totally different, 1 when two record values are exactly the same, and a similarity value between 0 and 1 when two record values contain some common characters [38, 184]. Approximate comparison techniques include, for example, Levenshtein edit-distance [127] and Dice-coefficient [33]. Levenshtein edit-distance calculates the number of character changes that are required (insertion, deletion, and substitution) between strings in a pair and finds the smallest number of changes required to convert one string to the other [127].

The Dice-coefficient calculates the similarity value between strings of records in a pair based on their common q-grams of strings in the two records [33]. The Dice-coefficient, sim_D, between strings x and y is calculated as:

$$sim_D(x, y) = \frac{2 \times n_c}{|\mathbf{q}_x| + |\mathbf{q}_y|}, \tag{2.1}$$

where $|...|$ denotes the cardinality of a set, n_c is a number of q-grams in common between strings x and y, and $\mathbf{q}_x$ and $\mathbf{q}_y$ are q-gram sets of strings x and y, respectively. For example, we assume the length of q-grams is $q = 2$, and two strings are $x =$ "*marry*" and $y =$ "*mary*". The strings x and y are first converted to the sets of q-grams $\mathbf{q}_x = \{ma, ar, rr, ry\}$ with $|\mathbf{q}_x| = 4$ and $\mathbf{q}_y = \{ma, ar, ry\}$ with $|\mathbf{q}_y| = 3$, respectively. The number of common q-grams between $\mathbf{q}_x$ and $\mathbf{q}_y$ is $n_c = 3$, where the common q-grams are ma, ar, and ry. Therefore, the Dice-coefficient between these strings x and y is $(2 \times 3)/(4 + 3) = 0.86$. For PPRL, we describe the Dice-coefficient for comparing two encoded values (Bloom filters [155]) in Section 2.3.

4. **Classification**
   Candidate record pairs can be classified as matches, non-matches, or potential matches, based on the similarity values between them [33]. Matches are record pairs that refer to the same real-world entity. Non-matches are record pairs that refer to different real-world entities [33, 67]. Potential matches are record pairs that are unclear to be classified as a match or non-match with regard to a certain classification threshold [33]. The simplest classification technique is a threshold based classification which uses one to two similarity thresholds to classify record pairs as matches, non-matches, or potential matches [33, 184].

   For example, we assume similarity values between pairs of records are calculated by using Dice-coefficient (Equation 2.1 and as we describe in Section 2.3 for PPRL), resulting in similarity values between 0 and 1. By using one similarity threshold, for example, $s_t = 0.5$, record pairs with similarity $sim_D \geq 0.5$ are classified as matches and record pairs with $sim_D < 0.5$ are classified as non-matches. When using two similarity thresholds, for example, $s_{t1} = 0.8$ and $s_{t2} = 0.5$, record pairs with $sim_D \geq 0.8$ are classified as matches, with $sim_D \leq 0.5$ are classified as non-matches, and with $0.5 < sim_D < 0.8$ are classified as potential matches. The potential matches need to be classified manually in a clerical review process which is time consuming and expensive due to the requirement of manual processing by domain experts [33] (as we illustrated in Figure 2.1).

   The probabilistic classification, also known as *probabilistic record linkage* [33], which was formalised by Fellegi and Sunter [67], is a widely used classification technique. In their approach, record pairs are classified as matches, non-matches, or potential matches based on match weights and two cutoff thresholds [33, 38]. Each record pair is first assigned into either a set of true matches, **M**, where the two records in a pair refer to the same entity, or a set of true non-matches, **U**, where the two records in a pair refer to different entities [33, 193]. Then, the conditional probabilities of a true match, P_m, and true non-match, P_u, of each attribute i of a record pair is calculated, where $1 \leq i \leq n$ and n is the number of compared attributes [33, 193].

The P_m and P_u probabilities of an attribute i are calculated as:

$$P_m^i = P([r_a^i = r_b^i, r_a \in \mathbf{D}_A, r_b \in \mathbf{D}_B] | (r_a, r_b) \in \mathbf{M}), \tag{2.2}$$

$$P_u^i = P([r_a^i = r_b^i, r_a \in \mathbf{D}_A, r_b \in \mathbf{D}_B] | (r_a, r_b) \in \mathbf{U}), \tag{2.3}$$

where r_a^i and r_b^i are attribute values of records r_a and r_b, respectively. The probabilities P_m^i and P_u^i are then used to calculate the weight of agreement, w_a, or disagreement, w_d, of an attribute i of a record pair (r_a, r_b) [33, 38] as:

$$w_i = \begin{cases} w_a = log_2\left(\frac{P_m^i}{P_u^i}\right) & if\ r_a^i = r_b^i, \\ w_d = log_2\left(\frac{(1-P_m^i)}{(1-P_u^i)}\right) & if\ r_a^i \neq r_a^i. \end{cases} \tag{2.4}$$

The weights w_i of all compared attributes in a record pair (r_a, r_b) are then summed into one weight, w. This weight w and the cutoff thresholds are then used for classifying the record pair as a match, non-match, or potential match.

However, there are many other classification techniques that have been proposed such as clustering [123], supervised [82], cost-based [186], and rule-based [33, 126] classification techniques.

5. **Evaluation**
   Evaluating the performance of a record linkage process can be achieved by measuring linkage quality and scalability [33, 184]. Linkage quality can be measured by calculating the accuracy of the linkage process [33, 182], while scalability can be measured by using reduction ratio, pairs completeness, pairs quality [33, 38, 182], and time complexity. For PPRL, in addition to measuring linkage quality and scalability, the measuring of privacy is also required. We describe these evaluation measures in detail in Section 2.5.

2.2 Privacy-Preserving Record Linkage (PPRL)

Many databases contain information about individuals, such as customers, patients, or taxpayers, where they often include sensitive information, such as names, addresses, and dates of birth [185]. *Privacy-preserving record linkage* (PPRL) [33] links records between databases without revealing such sensitive information. PPRL differs from traditional or conventional record linkage such that the records in databases that are to be linked must be encoded or encrypted to preserve the privacy of sensitive information contained in them [38]. We formally define PPRL in Definition 2.1 [38, 184].

Definition 2.1. ***Privacy-preserving record linkage (PPRL)***
Assume $DO_A, DO_B, DO_C, ...$ are n database owners with their respective databases $\mathbf{D}_A, \mathbf{D}_B, \mathbf{D}_C, ...$. The linkage process classifies candidate record pairs $(r_a, r_b), (r_b, r_c), (r_a, r_c), ...$, where $r_a \in \mathbf{D}_A, r_b \in \mathbf{D}_B, r_c \in \mathbf{D}_C, ...$, into a set of either $\mathbf{M}$ (set of matches) or $\mathbf{U}$ (set of non-matches) by using a decision model $\mathcal{C}$. At the end of the linkage process, no sensitive information is being revealed to any DOs or any external party, and the DOs only learn records which are in common between databases (set $\mathbf{M}$).

In the following subsections, we discuss different aspects of PPRL which are protocols, adversary models, and techniques.

2.2.1 Protocols

In the PPRL context, organisations such as financial, criminal, and government organisations are considered as *database owners* (DOs) when they provide their data for linking between databases [38]. A PPRL protocol can be a two-party, three-party, or multi-party protocol [38].

1. **Two-party protocol**
   The two DOs that participate in a PPRL protocol directly communicate with each other to conduct the linkage process [33, 38, 184]. The DOs individually process data cleaning and standardisation on their records, and then encode or encrypt their records before they conduct the linkage process, which includes the indexing, comparison, classification, and evaluation steps (as illustrated in Figure 2.1). Two-party protocols are more secure than the three-party protocol (which we will describe next) because the records that are being linked do not reveal to any external party [38]. However, they are more computationally and communicatively expensive than the three-party protocol because most of two-party protocols use secure multi-party computation (SMC) techniques (as we describe in Section 2.2.2) and iterative linkage approaches which are more complex and require many messages to be communicated between DOs [43, 116, 178].

2. **Three-party protocol**
   A three-party protocol uses a third party, called a *linkage unit* (LU), to perform the comparison and classification steps [33, 184]. A LU is a party that participates in the linkage process. In general, a LU does not have any data itself but it conducts the linkage between data sent to it by the DOs. The two DOs that participate in a protocol first agree on the parameters to be used for encoding or encrypting their records and to be used in the linkage protocol. Their records are encoded or encrypted and then sent to the LU to conduct the comparison and classification [38, 184]. However, the LU may aim to learn sensitive information from the encoded or encrypted records it receives from the DOs, thus, the LU is considered as a semi-trusted third party [38], as we describe in Section 2.2.2.

3. **Multi-party protocol**
   A multi-party protocol is similar to the two-party protocol when a LU is not involved and similar to the three-party protocol when a LU is involved in a linkage process [38]. However, with the increasing number of parties participating in a PPRL protocol, a multi-party protocol requires higher complexity and has computational costs to improve the degrees of privacy protection of the protocol [38]. This is because there is a higher risk that any parties involved in a linkage process can collude with other parties with the aim to re-identify the sensitive information of parties that are not colluding, and therefore more complex, secure, and efficient techniques are required [38].

2.2.2 Adversary Models

A PPRL protocol is considered as a secure protocol when no parties can infer any sensitive information of any other parties, except record pairs that are classified as matches [38]. To design a secure protocol for PPRL, the trustworthiness of each party participating in a protocol is considered. A party can be considered as a fully-trusted (honest), semi-trusted (honest-but-curious adversary model), or untrusted (malicious adversary model) [38, 116].

1. **Honest (fully-trusted party)**
   An honest party follows the steps of a PPRL protocol without trying to learn sensitive information of any other parties participating in a protocol [38]. An honest party also does not collude with any other parties (including external parties) to share information with the aim to learn sensitive information of other parties participating in a protocol [38]. However, this scenario is not always possible in real-world applications.

2. **Honest-but-curious adversary model (semi-trusted party)**
   Most protocols in the PPRL context follow the honest-but-curious (HBC) adversary model [38]. This model is also known as *passive* adversary model [116]. A semi-trusted party follows the steps of a PPRL protocol but it tries to learn sensitive information of the other parties that participate in a linkage process by storing data it receives from other parties and then uses these data to infer sensitive information of the other parties [33, 116].

3. **Malicious model (untrusted party)**
   An untrusted party may not follow the protocol steps and behave maliciously [33, 38, 116] by sending fake or invalid records to any other parties that participate in a PPRL protocol with the aim to learn sensitive information of the other parties [33, 38]. Most PPRL protocols that are developed to prevent a malicious adversary model are based on secure multi-party computation (SMC), as we describe next.

In this thesis, we aim to develop three-party PPRL protocols where we assume all parties follow the honest-but-curious (HBC) adversary model and the DOs do not collude with the LU.

2.2.3 PPRL Techniques

PPRL techniques can be categorised into two categories which are secure multi-party computation (SMC) based and perturbation based techniques [182].

2.2.3.1 Secure Multi-Party Computation

The concept of secure computation for two parties was first introduced by Yao [199] in 1986. The approach was originally proposed for generating and exchanging secret values between two parties where both parties could not learn any sensitive information of each other, except the final result of the computation. Later, the approach was extended for multiple parties, known as *secure multi-party computation* (SMC), by Goldreich et al. [75] in 1987. The techniques used in PPRL under the SMC category include homomorphic encryption, secure summation, secure set union, and secure set intersection [38, 43].

1. **Homomorphic encryption**
   Homomorphic encryption uses the concept of encryption (using a public key) and decryption (using a private key) [106]. Homomorphic encryption allows arithmetic operations such as addition and multiplication to be conducted over encrypted values (ciphertexts), where the decryption of the calculated results (addition, multiplication, or both) are the same as the results of conducting the same operation over plaintext (unencrypted) values [38].

   Homomorphic encryption can be grouped into three categories which are partially, somewhat, and fully homomorphic encryptions [1, 72]. Partially homomorphic encryption allows addition and multiplication (multiply only by a constant value) to be conducted over ciphertexts. Partially homomorphic encryptions include for example El-Gamal [64], Rivest–Shamir–Adleman (RSA) [151], Benaloh [10, 11], and Paillier [131].

   Somewhat homomorphic encryption such as Boneh-Goh-Nissim [17] allows both addition and multiplication to operate over ciphertexts. However, the encryption scheme can generate noise in the ciphertexts, and therefore in somewhat homomorphic encryption the number of arithmetic operations has to be limited to ensure the decryption results are correct.

   Fully homomorphic encryption was first proposed by Gentry [73] to overcome the problem of a somewhat homomorphic encryption scheme by allowing an unlimited number of operations of both additive and multiplicative to be conducted over the ciphertexts. However, fully homomorphic encryption is currently not efficient and feasible in real-world applications because of its complex encryption and decryption processes [192]. Various fully homomorphic encryptions have been proposed to improve the efficiency of Gentry's proposed approach [1]. For details of different homomorphic encryption schemes we refer the reader to the surveys by Acar et al. [1], and Fun and Samsudin [72].

2. **Secure summation**
   Secure summation allows more than two parties to compute the sum operation over their individual values where no party can learn any sensitive information related to the numerical value of the other parties that participate in the protocol [38]. We assume all parties have individual numerical values. In a basic secure summation protocol [43], these parties are communicating in a circular manner, where the first party generates a random value and conducts the sum operation with its private value. It then sends the summed result to the second party. The second party conducts the sum operation between the received value and its private value and sends the summation result to the next party to do the same. Once all parties have conducted the sum operation on their values, the last party sends the final summed result back to the first party.

   The first party then subtracts the random value from the final summed result, resulting in the summation between the actual values of all parties. Therefore, all parties participating in the protocol receive the summation of their values without knowing the actual values of the other parties [43].

   There are other secure summation protocols, for example, encrypted, salted, randomly shared, and homomorphically shared secure summation protocols. For details we refer the reader to the survey by Ranbaduge et al. [143].

3. **Secure set union**
   Secure set union uses the concept of commutative encryption and finds union between sets that belong to different parties participating in the protocol [43]. The union set (global set), $\mathbf{G}$, contains pairs of encrypted values and binary vectors that are computed by all parties, where each binary vector is with the length of a number of parties participating in the protocol. The binary vector is used by the parties to check if they have already encrypted each value in the set $\mathbf{G}$, where if a party has already encrypted a value, its position in the binary vector is set to 1. For example, if we assume the first party in the three-party protocol has the first position in the binary vector, thus, it sets the first position in the binary vector to 1 as $\langle 1, 0, 0 \rangle$.

   Let us assume three parties participate in a secure set union protocol where the first and third parties have the same value x, and the second party has the value y. The parties individually encrypt their values by using their public keys and then generate binary vectors of these encrypted values. They generate pairs of encrypted values and binary vectors, and insert them into a union set, $\mathbf{G}$. For example, the pairs $(\langle 1, 0, 0 \rangle, E_A(x))$, $(\langle 0, 1, 0 \rangle, E_B(y))$, and $\langle 0, 0, 1 \rangle, E_C(x))$ are generated by the first, second, and third party, respectively. The parties then encrypt any values in $\mathbf{G}$ that they have not encrypted, resulting in the set $\mathbf{G} = \{(\langle 1, 1, 1 \rangle, E_C(E_B(E_A(x)))), (\langle 1, 1, 1 \rangle, E_C(E_A(E_B(y)))), (\langle 1, 1, 1 \rangle, E_B(E_A(E_C(x))))\}$. After that, the parties remove any duplicate pairs from $\mathbf{G}$. As the secure set union protocol uses commutative encryption, the encryption $E_C(E_B(E_A(x)))$ is equivalent to $E_B(E_A(E_C(x)))$. Thus, after removing a duplicate pair, the set $\mathbf{G}$ is $\mathbf{G} = \{(\langle 1, 1, 1 \rangle, E_C(E_B(E_A(x)))), (\langle 1, 1, 1 \rangle, E_C(E_A(E_B(y))))\}$.

The parties then decrypt the encrypted values in **G** by using their private keys and then permute values in **G**. At the end of this protocol, the parties know the union values of all of them without knowing the owner of each value in **G** [43].

4. **Secure set intersection**
   Secure set intersection protocol is used to find common values between different sets owned by different parties, where no party can learn sensitive information about the other parties [38, 43]. In other words, at the end of the protocol, all parties can only learn a common set of values between databases. This protocol can use either commutative or homomorphic encryption to encrypt the values of each party that participates in the protocol [38]. However, the secure set intersection requires high computation costs due to the complex encryption process and communications between parties [38].

Although SMC based techniques are efficient, they are not widely used in the PPRL context because these techniques have high computational costs which are not suitable to process large amounts of data [185].

2.2.3.2 Perturbation

Perturbation based techniques encode sensitive plaintext values into encoded values. The aims of perturbation techniques are to make the encoded values and their distributions different from the plaintext values and to make it more difficult to be re-identified by an adversary [2]. However, perturbation based techniques often result in a trade-off between linkage quality, scalability, and privacy [182]. We describe some perturbation techniques below.

1. **Noise addition**
   The noise addition technique adds random values to a database, where each random value can be added to an attribute value of a record or added as an extra record in a database [38, 184]. This technique can be used to prevent a frequency attack [38] because it conceals the original frequency distribution of each record value by random values. Although the privacy of the linkage process is improved, these random values can increase the size of the database and lower the similarity between records, leading to lower linkage quality [38, 184].

2. **Differential privacy**
   Differential privacy was proposed as an alternative to the noise addition technique which provides more controlled noise addition with mathematical guarantees [38]. In this technique, a random value is added into a statistical query result based on the privacy parameter, ϵ, to change the frequency distribution of the result [38, 62]. Formally, a randomised algorithm, $\mathcal{R}$, is applied to a query result, where it provides ϵ-differential privacy if the two databases, $\mathbf{D}_A$ and $\mathbf{D}_B$, contain one different record [61]. The probability of $\mathcal{R}$ providing ϵ-differential

privacy is calculated as:

$$P(\mathcal{R}(\mathbf{D}_A) \in \mathbf{S}) \leq e^{\epsilon} \times P(\mathcal{R}(\mathbf{D}_B) \in \mathbf{S}), \tag{2.5}$$

where $\mathbf{S}$ is the set of all possible subsets of the output from $\mathcal{R}$ and e^{ϵ} is an exponential of ϵ [38, 61].

Differential privacy makes it more difficult for an adversary to learn actual counts or sums of individual records, and learn if an individual value exists in a database or not, thus, providing provable privacy guarantees [38, 61, 62].

3. **Embedding**
An embedding technique [98, 153] maps sensitive values into a multidimensional space such as Euclidean space or Hamming space. In this technique, random values are first generated. These values are then used to generate an embedding space. After that, the sensitive values in a database are mapped to the generated embedding space by computing distances between sensitive values and random values in the generated embedding space, resulting in embedded vectors. These vectors are then compared and classified as matches or non-matches.

4. **Hash encoding**
A hash function encodes a sensitive value into a different value (hash code) which is irreversible and difficult for an adversary to conduct a dictionary attack [136]. One-way hashing techniques such as the Message Digest (MD) and Secure Hash Algorithms (SHA) are widely used [38, 184], where the most popular techniques are MD5, SHA1, and SHA2. However, SHA is slower than the MD technique because the SHA encodes a value into a longer bit length [136], where SHA1 encodes a value into 160 bits and SHA2 encodes a value into 224 or 512 bits, while MD encodes a value into 128 bits [38, 136]. To improve the privacy protection of these hashing techniques against a dictionary attack, a key-hash message authentication code (HMAC) can be used [107]. HMAC uses a secret key combined with a one-way hashing (MD or SHA) to encode a sensitive value. However, HMAC values can still be vulnerable to a frequency attack [38, 184].

   Locality sensitive hashing (LSH) [58] is a hashing technique that uses an independent set of hash functions to map values into partitions leading to similar values being mapped into the same partition that have high similarity [169]. The techniques that were developed based on LSH are for example Euclidean LSH [48] and Hamming LSH [58].

   Other techniques that have been proposed to use hash encoding are Bloom filter (BF) encoding [155], tabulation based hashing (TMH) [170], and two-step hashing [139] techniques, where BF encoding is the main aspect of this thesis. We describe BF encoding in the next section (Section 2.3) and discuss the TMH and two-step hashing techniques in Chapter 3.

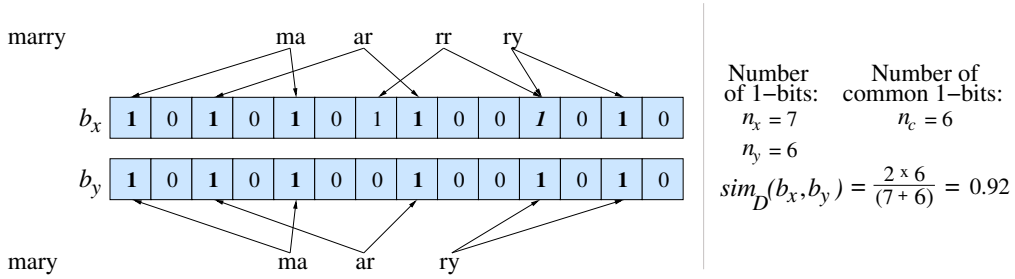

Figure 2.2: Dice-coefficient similarity calculation between two BFs [33, 155], b_x and b_y that are encoded of the string values "*marry*" and "*mary*", respectively. The q-grams *rr* and *ry* are encoded into the same bit position (a collision) as shown in italic in b_x. The common 1-bits between these BFs are shown in bold.

2.3 Bloom Filter Encoding in PPRL

Bloom Filter (BF) encoding for PPRL was first proposed by Schnell et al. [155] in 2009 and from there onward BF encoding has been the most widely used perturbation technique in PPRL. This is because the BF encoding allows approximate matching between strings [155, 185], numerical values [95, 181], hierarchical codes (such as of occupation and diseases) [161, 162], and geographical locations [163].

A BF is a bit vector that is initially set to all zeroes [155]. To encode a string value to a BF, a string value is first converted into a set of q-grams, $\mathbf{q}$, where each q-gram is a sub-string of length q characters. Each q-gram in $\mathbf{q}$ is then encoded by using a set of hash functions and a hash method, where the bit positions that correspond to the encoded q-grams are set to 1 [155], as we describe in detail in Section 2.3.1.

The similarity value between a pair of BFs can be measured by using the Dice-coefficient [33] which is calculated as:

$$sim_D(b_x, b_y) = \frac{2 \times n_c}{n_x + n_y}, \tag{2.6}$$

where n_c is a number of common 1-bits, and n_x and n_y are the numbers of 1-bits in the two BFs, b_x and b_y, of the two string values x and y, respectively [155]. We illustrated an example of two BFs and their similarity calculation in Figure 2.2.

Although BF encoding is widely used and different BF encoding techniques have been proposed [58, 59, 156], as we describe in Section 2.3.2, several research works have shown that BF encoding is vulnerable to privacy attacks by re-identifying the actual string values that are encoded in them [40, 42, 108, 109], as we describe in Section 2.3.3. Therefore, hardening techniques have been developed to improve the degrees of privacy of BFs [157, 175]. We discuss different hardening techniques in Section 2.3.4.

2.3.1 Hashing Technique for Bloom Filters

As we described above, a set of hash functions and a hash method are required in a BF encoding process. In this thesis, we focus on double hashing and ran-

dom hashing methods because double hashing is widely used in BF approaches for PPRL [58, 59, 156], while we use random hashing in our proposed works, as we describe in Chapters 4 and 5. However, there are other hashing methods that have been proposed. For details we refer the reader to the book by Christen et al. [38].

2.3.1.1 Double Hashing

Dillinger and Manolios [52] proposed to use two (double) to three (triple) hash functions to generate BFs to improve performance and reduce overhead in the BF encoding process. Later, Kirsch and Mitzenmacher [105] proposed the idea that two hash functions can simulate more than two (k) hash functions without asymptotic false positive probability increased. They generate k hash values of a value x as:

$$h_i(x) = (\mathcal{H}_1(x) + i \times \mathcal{H}_2(x)) \; mod \; l, \tag{2.7}$$

where $\mathcal{H}_1()$ and $\mathcal{H}_2()$ are two independent hash functions, i is the range from 0 to $k-1$, l is the length of a BF, and *mod* is the modulo operation. The *mod* l is used to ensure that the computed index value is valid in the ranges from 0 to $l-1$ for a BF of length l.

2.3.1.2 Random Hashing

Niedermeyer et al. [129] demonstrated that BFs that encode values using double hashing can be easily attacked based on the sequential hash encoding patterns due to the composition of the two hash functions [105]. Therefore, they suggested a random hashing method to minimise information that can reveal to an adversary to conduct the pattern based attack. Schnell and Borgs [157] conducted the pattern based attack on BFs encoded using random hashing and concluded that the random hashing can prevent the pattern based attack proposed by Niedermeyer et al. [129].

To generate a BF by using random hashing, they first generate a single value to be used as a random seed to ensure the same values are encoded into the same hash values. This single value can be generated by using the pseudo-random number generator (PRNG) [38]. They then generate a set of k random numbers for each value to be encoded, where $0 \leq k \leq l-1$ and l is the length of a BF. Finally, for each random number in a set, they use it as a bit position in a BF and set the bit at this position to 1. As a result, it is more difficult for an adversary to conduct a pattern based attack on the BFs because random bits in a BF are set to 1.

2.3.2 Bloom Filters Encoding Techniques

There are different types of BF encodings that have been proposed which are the attribute-level BF (ABF) [155], cryptographic longterm key (CLK) [156], record-level BF (RBF) [58, 59], and hybrid BF (CLKRBF) [182].

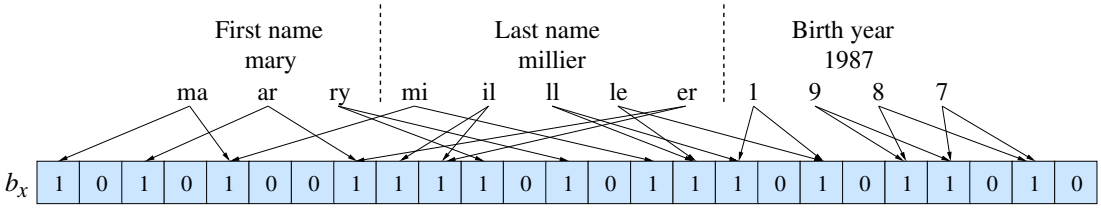

Figure 2.3: Cryptographic longterm key BF [156], b_x, of the record x that is generated from the attributes first name, last name, and birth year, where no padding character is added.

2.3.2.1 Attribute-level Bloom Filter (ABF)

Schnell et al. [155] proposed the attribute-level BF (ABF) encoding technique, also known as *field-level* BF. In their approach, an attribute value is first converted into a string and padded at the beginning and end with a special character that is not contained in the databases to be linked [38]. The padded string is then converted into a set of sub-strings, called *q-grams*, where each q-gram in a set has the length q.

For example, an attribute first name contains the string $x =$ "*mary*". This x is padded with the special character "_", resulting in the string $x' =$ "*_mary_*". The string x' is then converted into a q-gram set, where each q-gram has the length $q = 2$, resulting in a q-gram set $\mathbf{q}_x = \{\_m, ma, ar, ry, y\_\}$. However, the padding character can reveal the frequency of q-grams that are at the beginning and end of strings [155]. Therefore, unless the linkage process needs an emphasis on the beginning and end of strings, the padding character is not required [38]. The k hash functions are then used to hash encode and map each q-gram in a set of q-gram, $\mathbf{q}$, into the BF, where the mapped bit positions in the BF are set to 1.

In their approach, they apply double hashing (Equation 2.7) [105], as we described in Section 2.3.1, for mapping q-grams in $\mathbf{q}$ to k bit positions. When two or more q-grams are mapped to the same bit positions in a BF, called a *collision*, it leads to false matches that can lower the linkage quality of a PPRL protocol. However, collisions improve the degrees of privacy of the linkage process because they make it more difficult for an adversary to re-identify the individual q-grams that map to bit positions [38, 155]. We illustrated an example of two BFs (ABFs) with no padding character added and where a collision occurs in a BF in Figure 2.2.

2.3.2.2 Cryptographic Longterm Key (CLK)

Schnell et al. [156] proposed the cryptographic longterm key (CLK) encoding technique to allow error-tolerant comparisons while the privacy of sensitive information is preserved. The idea of this approach is to encode multiple attributes of a record into a single Bloom filter (BF) [155], where each attribute value is encoded into the BF independently.

In this approach, each attribute value is first converted to a string. Each string is then used to generate a set of q-grams, resulting in a number of sets of q-grams that equals the number of attributes to be used in the linkage process. The BF with the length l is then initialised and set to all zeroes. Each q-gram in each q-gram set is then mapped into the BF.

As illustrated in Figure 2.3, three attributes of the record x are being used in the linkage process. These attributes are first name, last name, and birth year, with the values "*mary*", "*miller*", and "1987", respectively. We assume a padding character is not added to these values. The sets of q-grams of attributes first and last names are generated with the length of q-gram $q = 2$, and the attribute birth year is generated with the length of q-gram $q = 1$. Therefore, three sets of q-grams are generated which are $\{ma, ar, ry\}$, $\{mi, il, ll, le, er\}$, and $\{1, 9, 8, 7\}$, respectively. These sets are then encoded by using double hash encoding [105] (Equation 2.7) and mapped into a BF of length $l = 25$.

2.3.2.3 Record-level Bloom Filter (RBF)

Durham et al. [58, 59] proposed the record-level Bloom filter (RBF) encoding technique. Similar to the CLK approach [156], RBF encoding was proposed to improve the privacy protection level of BFs [155] and multiple attributes of a record are encoded into a single BF. However, the steps to generate a RBF are different from generating a CLK. These steps are (1) ABF [155] generation, (2) eligible bit identification, (3) attribute weighting, (4) RBF generation, and (5) RBF permutation.

1. **ABF [155] generation**
   Each attribute value to be used in the linkage process is first encoded into an ABF [155], as we described on page 25. However, in the RBF approach, the length of each ABF, l_{ABF}, can be either static or dynamic.

   Static length means every attribute value to be linked is encoded into ABFs of the same length l_{ABF}, while dynamic length means the length of an ABF, l_{ABF}, of each attribute value depends upon the attribute. For example, let us assume padding is not applied and the length of q-gram is $q = 2$. The attribute birth year contains a string value with the length of 4 characters, resulting in the length of q-gram set $|\mathbf{q}| = 3$, while the attribute telephone number contains a string value with the length of 10 characters, resulting in the length of its q-gram set $|\mathbf{q}| = 9$. Therefore, the l_{ABF} for the attribute birth year is shorter than the l_{ABF} for the attribute telephone number.

   The dynamic length of an attribute can be calculated as [59]:

   $$l_{ABF} = \frac{1}{1 - \sqrt[k \times |\mathbf{q}|]{P}}, \tag{2.8}$$

   where k is a constant number, $|\mathbf{q}|$ is the length of a q-gram set, and P is the probability of bits that are unset to 1. For example, we assume $k = 5$ and $P = 0.5$, the l_{ABF} of the attribute birth year with the length of a q-gram set

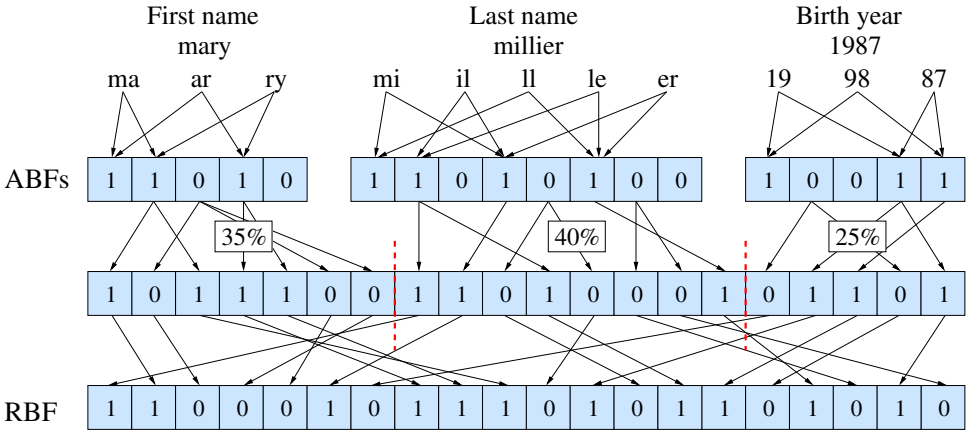

Figure 2.4: Record-level BF [58, 59] of a record that is generated from three attributes, which are first name, last name, and birth year, where no padding character is added.

$|\mathbf{q}| = 3$ results in $l_{ABF} = 22$ bits, while the l_{ABF} of the attribute telephone number with a length of the q-gram set $|\mathbf{q}| = 9$ results in the $l_{ABF} = 65$ bits.

2. **Eligible bit identification**
   The bits in the generated ABFs are then selected for generating a RBF if and only if the bits at certain positions can be mapped to at least a certain (security constraint) number of attributes. In other words, only bits that an adversary cannot clearly re-identify the original q-grams are being selected for generating the RBF [38]. However, the higher this security constraint of number of attributes is, the higher are the computational costs for generating the final RBF.

3. **Attribute weighting**
   The number of bits to be sampled from each ABF to generate a RBF of a record can be calculated based on the discriminatory power of an attribute or on the Fellegi and Sunter (FS) [67] model. The FS method calculates weights of agreement, w_a, and disagreement, w_d, of each attribute based on the expectation-maximisation (EM) [50] algorithm. The weight, w, of an attribute is calculated as:
   $$w[i] = \frac{w_a[i] - w_d[i]}{\sum_{i=1}^{|\mathbf{a}|} (w_a[i] - w_d[i])}, \tag{2.9}$$
   where $\mathbf{a}$ is a list of attributes and i is the index of an attribute in $\mathbf{a}$. The weight w is then converted into a percentage form by multiplying it with 100, $w \times 100$.

4. **RBF generation**
   In this step, the possible length of a RBF, $l_{p_{RBF}}$, is first calculated as:
   $$l_{p_{RBF}} = \left(\frac{l_{ABF}}{w}\right) \times 100, \tag{2.10}$$

where l_{ABF} is the length of ABF of an attribute and w is a weight (in percentage form) of an attribute. The maximum $l_{p_{RBF}}$ across all attributes is used as the length of the RBF, l_{RBF}, of a record. The l_{RBF} is then used to calculate the number of bits to be sampled, n, from each attribute as:

$$n = \frac{w \times l_{RBF}}{100}. \tag{2.11}$$

Then, n bits at the eligible bit positions (as we described in step 2) of each attribute are selected. The selected bits of all attributes are inserted into one BF of length l_{RBF}, resulting in the RBF of a record.

5. **RBF permutation**
   The generated RBF is randomly permuted to improve the degrees of privacy of the RBF, where the permutations are the same between two databases to be linked. This is to ensure that similar or the same record values are being encoded into similar or the same RBFs.

We illustrate the example of a RBF in Figure 2.4. The ABFs of attributes first name, last name, and birth year are generated by using the length of q-gram $q = 2$, dynamic length with constant number $k = 1$, and probability of bits set to 1 $P = 0.5$. The weights of these attributes are $w = 35\%, 40\%$, and 25%, respectively.

2.3.2.4 Hybrid Encoding (CLKRBF)

Vatsalan et al. [182] proposed a hybrid BF encoding approach that combines the CLK [156] and RBF [58, 59] encoding techniques to encode different attributes into a single BF. In their approach, different numbers of hash functions are first assigned to different attributes based on the weight of each attribute to improve the accuracy of BF comparison, as in the RBF approach [58, 59]. The q-grams of attributes are then encoded using their corresponding number of hash functions into a BF of length l, as in the CLK approach [156], to improve the privacy protection level of the linkage process. This allows hash collision to occur, and as a result, it is more difficult for an adversary to conduct a frequency attack [40].

2.3.3 Attacking Techniques for Bloom Filters

Several research works have shown that Bloom filter (BF) encoding can be attacked [40, 42, 108, 109]. In this thesis, we mainly focus on the cryptanalysis attacks proposed by Christen et al. [39, 40] as we use these attacks to evaluate the privacy of our approaches, as we describe in Chapters 4 and 5. However, there are many other attacks for BFs that have been proposed. For details we refer the reader to the book by Christen et al. [38].

The ideas of cryptanalysis attacks proposed by Christen et al. [39, 40] are that the adversary maps frequent BFs of the linked databases to frequent attribute values

in a publicly available database, resulting in pairs of BFs and attribute values. The adversary then ranks these pairs based on their frequencies. After that, the adversary initialises sets of possible and not-possible q-grams. For each bit position in a BF in a pair, if the bit is set to 1, the adversary inserts all q-grams of an attribute value in the pair into the set of possible q-grams. If the bit is set to 0, the adversary inserts all q-grams into the set of not-possible q-grams. Based on these sets of possible and not-possible q-grams, the adversary then tries to re-identify the original attribute values in the linked databases.

However, while the attack proposed by Christen et al. [40] in 2017 mainly uses the set of possible q-grams to re-identify the original attribute values in the databases to be linked, the attack that they proposed in 2018 [39] improves the precision by using the set of not-possible q-grams.

2.3.4 Hardening Techniques

In this section, we describe different hardening techniques that have been proposed to improve the degrees of privacy of BFs [157, 175].

2.3.4.1 Salting

Niedermeyer et al. [129] proposed the salting hardening technique to avoid an adversary to correctly re-identify original strings from BFs [155]. In their approach, an extra string value, called *salt*, is concatenated to each q-gram of a string before it is encoded into a BF, where salt is specific for each individual record in a database and does not change over time [38]. For example, let us assume we use the birth year as a salting value and two records have the same string value, "*mary*", while their corresponding strings of birth years are different, for example, "1987" and "1990". Therefore, the sets of q-grams of these two records are $\mathbf{q}_1 = \{ma1987, ar1987, ry1987\}$ and $\mathbf{q}_2 = \{ma1990, ar1990, ry1990\}$. Q-grams in the sets $\mathbf{q}_1$ and $\mathbf{q}_2$ are then encoded into two BFs, resulting in two BFs to contain different bit patterns. With no salting technique applied to the sets of q-grams, the two records with the same string value will result in two BFs that contain the same bit patterns. Therefore, the salting hardening technique improves the privacy of BFs by reducing the frequency of bits that are encoded of the same q-grams in a database [38].

2.3.4.2 XOR-folding

Schnell and Borgs [158] proposed a hardening technique to improve the privacy level of BF encoding [155] by using a vector folding and bit-wise XOR operation. In their approach, a BF of length l is divided into two bit vectors, b_1 and b_2, with the same length, $|b_1| = |b_2| = l/2$. These bit vectors, b_1 and b_2, are then combined by using a bit-wise XOR operation, $\oplus$, where the XOR results of bits $0 \oplus 1$ and $1 \oplus 0$ are 1 and the XOR results of bits $0 \oplus 0$ and $1 \oplus 1$ are 0. For example, we assume a BF with length $l = 12$ is $b = 110101010001$. This b is then divided into two bit vectors,

$b_1 = 110101$ and $b_2 = 010001$. These b_1 and b_2 are then combined using the bit-wise XOR operation, resulting in the final BF $b' = 100100$.

However, this hardening technique can lower linkage quality because each bit position in the final BF, b', is combined from bits that possibly encode multiple q-grams. Furthermore, the recently proposed cryptanalysis attack by Christen et al. [40] (as we described in Section 2.3.3) has shown that the XOR-folding technique can be attacked by identifying the frequency distribution of 1-bit positions in BFs.

2.3.4.3 Balancing

Schnell and Borgs [157] proposed the balancing BF hardening technique. In their approach, a BF of length l is concatenated with its negated copy (all bits are flipped), resulting in a new BF of length $2 \times l$ with half of the bits in the BF set to 1. Then, the new BF is randomly permuted, resulting in bits that are encoded of each q-gram being located at random positions in the BF, thus, the privacy of a new BF is improved. For example, we assume a BF of length $l = 6$ is $b = 100110$. This b is concatenated with its negated copy which is 011001, resulting in a new BF, $b' = 100110011001$. This b' is then randomly permuted, resulting in a balanced BF, $b'' = 110010101010$. Although this technique improves the privacy of BFs, Christen et al. [40] (as we described in Section 2.3.3) have shown that they can successfully re-identify the original q-grams from the balanced BFs.

2.3.4.4 BLoom and FlIP

The BLoom and FlIP (BLIP) method randomly flips bit values in certain bit positions of a BF [157]. It was originally proposed by Alaggan et al. [5] in the context of privacy-preserving comparing of user profiles in social networks. Recently, Alaggan et al. [4] used the BLIP method for Wi-Fi analytics by applying pan-privacy which maintains the privacy of the internal states although the entire internal state is visible to an adversary. Their method was called pan-private BLIP. The method is based on differential privacy [62] and the Bernoulli distribution [152]. The Bernoulli distribution includes μ_0 which is the chance of flipping a bit to 0 and μ_1 which is the chance of flipping a bit to 1. These μ_0 and μ_1 are defined as $\mu_0 = 1/2 - \eta/2$ and $\mu_1 = 1/2 + \eta/2$, where η is a parameter in the range 0 to 1. This method improves privacy against legal and illegal intrusions into the systems [4].

The BLIP method was first used as a hardening technique for BF encoding [155] by Schnell and Borgs [157]. They adapted the BLIP method to improve the privacy protection level of BFs based on the idea proposed by Erlingsson et al. [65] called RAPPOR. In contrast to the approaches proposed by Alaggan et al. [4, 5], the RAPPOR method randomly selects bits to be set to 0 or 1 with equal probability.

When different BLIP methods and different flip probabilities are used, the flipped BF results in different numbers of 1-bits [38]. The similarity value between two flipped BFs can be increased or decreased depending upon the randomly flipping

Table 2.2: Bits transformation of Rule 90.

Bit patterns	111	110	101	100	011	010	001	000
Transformed bits	0	1	0	1	1	0	1	0

bits in these BFs [38]. This leads to false matches, thus, it can lower linkage quality. We propose an approach for improving the linkage quality of the BLIP methods, as we describe in Chapter 4.

2.3.4.5 Rule 90

Rule 90 is one of 256 automata transformation rules that were proposed by Wolfram [195], where Schnell and Borgs [159] were the first who proposed to use this rule as a hardening technique for BFs. The idea behind this hardening technique is that the bits in a BF are transformed to different bit values based on Rule 90 [159], as we illustrate in Table 2.2.

In this approach, a BF, b, of length l contains bit positions $\{p_0, p_1, ..., p_{l-1}\}$. The bit at the position p_0 is added to the right of the bit at the position p_{l-1} and the bit at the position p_{l-1} is added to the left of the bit at the position p_0, resulting in the length of a BF, b, to become $l' = l + 2$. A sliding window of size 3 is then applied to b. After that, Rule 90 is applied to each window to generate a new bit, resulting in b containing l bits. In other words, the bit-wise XOR operation, $\oplus$, is applied between bits at the positions p_{i-1} and p_{i+1} to generate a new bit at the position p_i, where $0 \leq i \leq l' - 1$ and $l' = l + 2$ [38]:

$$b[p_i] = b[(p_{i-1})] \oplus b[(p_{i+1})]. \tag{2.12}$$

For example, we assume a BF, b, of length $l = 6$ bits is $b = 110010$. We add the bits at positions p_0 and p_{l-1} (as we described above) to b, resulting in $b = 01100101$ with the length $l' = l + 2 = 8$. We then use a sliding window of size 3 and check the bits in each window with Rule 90 in Table 2.2 (or using Equation 2.12) to generate the new bits. Therefore, the final results of b after applying Rule 90 is $b = 111100$ with the length $l = 6$. The hardening technique based on Rule 90 improves the privacy of BF because it is non-reversible, thus, it can be used to against bit pattern based attacks [159].

2.3.4.6 Markov Chaining

Schnell and Borgs [160] proposed a Markov chaining based hardening technique for BFs (MCBF) with the aim to avoid frequency attacks where the frequent bits in BFs can be re-identified. In their approach, a large set of q-grams, **g**, are first generated based on either a probabilistic language model [118] or a large public database in the same domain as the databases being linked [38].

Table 2.3: Example of frequent co-occurrence q-grams in a large set **g**.

Q-gram	Co-occurrence q-gram	Frequency
ar	*re*	152
ar	*ri*	206
ar	*rr*	198
ma	*ae*	105
ma	*an*	195
ma	*as*	287
ry	*yy*	165
ry	*y−*	199
ry	*y_*	187

For each string value in a database to be linked, a set of q-grams, **q**, is generated. Then, for each q-gram in **q**, the n extra q-grams are selected from the generated set **g** based on a transition probability that a unique q-gram occurs after a certain q-gram [118]. These selected extra q-grams are inserted into **q** and all q-grams in **q** are then encoded to a BF. For example, we assume the string value is $x =$ "*mary*". The set of q-grams of this string is $\mathbf{q} = \{ma, ar, ry\}$. For each q-gram in **q**, we randomly select $n = 2$ extra q-grams from the set **g**, as we illustrate in Table 2.3, based on a transition probability. As a result, the string x has its corresponding set of q-grams $\mathbf{q} = \{ma, as, an, ar, ri, rr, ry, y-, y\_\}$. This set **q** is then used to generate a BF. The MCBF hardening technique improves the privacy of BF by making it more difficult for a frequency attack because the addition of extra q-grams distorts the frequency distribution of q-grams in the encoded database [38].

2.3.4.7 Windowing based XORing

Ranbaduge and Schnell [141] recently proposed a hardening technique based on a sliding window to improve the privacy protection level of BFs. In their approach, they first generate a BF of length l. They then generate two windows, $\mathbf{w}_1$ and $\mathbf{w}_2$, of the same size, $|\mathbf{w}| = |\mathbf{w}_1| = |\mathbf{w}_2|$. These windows are iteratively moved over the generated BF, where the first position of $\mathbf{w}_1$ on the BF is p, the first position of $\mathbf{w}_2$ is $p + 1$, and $0 \leq p \leq l - |\mathbf{w}|$. For each iteration, they conduct XOR operation, $\oplus$, between bits in the two windows resulting in new $|\mathbf{w}|$ bits. For example, we assume $\mathbf{w}_1$ contains bits $[1, 1, 0, 0, 1]$ and $\mathbf{w}_2$ contains bits $[1, 0, 1, 1, 0]$. The new bits XORed from these windows are $\mathbf{w}_1 \oplus \mathbf{w}_2 = [0, 1, 1, 1, 1]$. The iteration is terminated when all bits in the original BF are processed. However, this approach provides a trade-off between the scalability and linkage quality of PPRL, where the smaller size of window uses more runtime but provides higher linkage quality than the larger size of the window.

2.3.4.8 Re-sampling based XORing

Re-sampling based XORing hardening technique was recently proposed by Ranbaduge and Schnell [141]. In this approach, they first generate a BF of length l. They then generate a new bit (harden) for each bit position, p, where $0 \leq p \leq l-1$, in the generated BF by randomly selecting bits (sampling) from two bit positions in the BF, p_i and p_j, where $0 \leq p_i, p_j \leq l-1$. The two bits at these p_i and p_j positions are then XORed, $\oplus$, resulting in a new bit. They then replace a bit at the position p in the BF with this new bit. They repeat conducting these steps for l times until all bits in the original BF are replaced by the new (XORed) bits. For example, we assume a BF is $b = 100110$ and the randomly selected bit positions p_i and p_j for the first bit in b are 2 and 5, respectively. The bits at these positions are then XORed, resulting in a new bit $p_i \oplus p_j = 0 \oplus 0 = 0$. This new bit is replaced to the first bit position in b, resulting in a BF $b = 000110$. These sampling (randomly select bits) and replacement steps are repeated until all original bits are replaced by the new generated bits.

2.4 The Relationship of Big Data with Record Linkage and PPRL

The term *Big data* was first used in 1980 by Tilly [172] in a social history paper, where the history of Big data began in 1944 when Rider [150] raised the problem of data growth in the American research library that could not be ignored due to the doubling of data size around every sixteen years. The increasing volume of data stored in databases leads to the need of efficient management for storing, processing, and analysing such large data collections [90]. In 1990, Denning [51] proposed that it was possible to build a fast machine to predict patterns from large volumes of data.

In computer science, the term *Big data* was first used by Cox and Ellsworth [45] to describe the requirement of resources to be able to fit and process large volumes of data. However, Cox and Ellsworth [45] were not credited as the first who used the term *Big data*, rather Mashey has been credited because he used this term in his various speeches [119].

Big data is used in many organisations and application domains, such as businesses, research institutes, healthcare service providers, and criminal detection organisations [185]. Some organisations need to integrate data from multiple sources to facilitate data mining [90]. However, Big data integration [54, 55] is challenging because of the four main characteristics of Big data which are volume, velocity, variety, and veracity (known as the four Vs) [90, 185]. These four characteristics of Big data relate to the three challenges of PPRL which are linkage quality, scalability, and privacy, as we described in Chapter 1.

1. **Volume**
   Volume refers to a large amount of data and possibly a large number of data sources that can be from either the same or different domains [55]. Volume is challenging in the Big data context because it requires high performance

techniques, large data storage, and high computing resources, which are used for storing, processing, and analysing large amounts of data [90, 198]. Volume is also challenging in the contexts of record linkage and PPRL. This is because the comparison of all pairs of records in large databases is not feasible [185]. Therefore, blocking and filtering techniques are needed in record linkage and PPRL because these techniques improve the efficiency of linking records by reducing the candidate record comparison space in the linkage process [3, 78, 142, 144, 183, 185].

2. **Velocity**
   Velocity refers to the speed of data creation, processing, and analysis [90]. Velocity relates to the volume characteristic of Big data and it is challenging in the contexts of record linkage and PPRL. This is because high performance techniques and computing resources are required to process data and compare all record pairs in the linkage process [185, 198]. Furthermore, some financial institutes and criminal detection organisations require near real-time record comparison results to act quickly on the decision making processes [96]. Therefore, speeding up the data processing and linkage process is needed. Speeding up these processes can be achieved by using blocking, filtering, and parallelisation techniques [71, 78, 164, 185].

3. **Variety**
   Variety refers to the heterogeneous nature of data, such as different data formats and inconsistent data semantics [55, 90]. Linking heterogeneous data is challenging in record linkage and PPRL contexts because it can result in low linkage quality and incorrect data analysis [90]. Therefore, techniques for linking databases that can handle heterogeneous data are required [92, 185]. One PPRL technique for linking heterogeneous databases was proposed by Karakasidis and Verykios [92]. In their approach, they used multiple reference databases with different data types to generate blocks of encoded records that are being compared in the linkage process.

4. **Veracity**
   Veracity refers to different levels of data quality, where data might be missing and/or contain erroneous values [185]. Veracity is challenging because removing records that contain missing values can lower the quality of record linkage and PPRL since these records possibly contain important information which should be used in data analysis [33, 185]. Imputation technique such as *k*-nearest neighbours (k-NNs) can be used for linking databases that contain missing values by imputing values from *k* similar records of a record that contain missing values (in an attribute). These imputed values are then used in the linkage process [30]. However, the k-NNs imputation technique is suitable for databases that contain groups of similar records, as we describe in detail in Chapter 3. Therefore, techniques for linking databases with missing values are still required.

As described above, many organisations have to process and link databases every day, while the four Vs of Big data are challenging and related to the three challenges of PPRL. Therefore, to achieve the aim of this thesis which is to develop efficient PPRL techniques to provide high linkage quality, where the privacy and scalability challenges of PPRL are of concern (as we described in Chapter 1), we also need to consider the four Vs of Big data.

2.5 Evaluation

Evaluation of record linkage and PPRL protocols are similar because both of them measure linkage quality and scalability, while privacy is only measured in the PPRL protocol [33, 184].

2.5.1 Linkage Quality

When true matches and true non-matches are available in ground truth data, the linkage quality can be measured by calculating precision, recall, F-measure, as well as many other evaluation measures [33]. To calculate these measures, we use the three counts which are: (1) true positives (TP) which is the number of true matches correctly classified as matches, (2) false positives (FP) which is the number of non-matches incorrectly classified as matches, and (3) false negatives (FN) which is the number of true matches incorrectly classified as non-matches [182, 184].

1. **Precision**
   Precision (*prec*) measures the proportion of correctly classified true matches over all classified matches [33, 182]. Precision is calculated as:

$$prec = \frac{TP}{TP + FP}. \tag{2.13}$$

2. **Recall**
   Recall (*rec*) measures the proportion of true matches that are correctly classified as matches over all true matches [33, 182]. Recall is calculated as:

$$rec = \frac{TP}{TP + FN}. \tag{2.14}$$

3. **F-measure**
   F-measure (FM) is used to measure the harmonic mean between precision and recall [33, 126], and calculated as:

$$FM = 2 \times \left(\frac{prec \times rec}{prec + rec} \right). \tag{2.15}$$

   However, to make the F-measure comparable between different linkage methods, different similarity thresholds should be used [83]. The value of the F-

measure is high if both precision and recall values are high. If the similarity thresholds are low, the recall value generally becomes higher than the precision value, while if the similarity thresholds are high, the precision value becomes higher than the recall value [83].

2.5.2 Scalability

In general, scalability measures the performance of the indexing step in record linkage and PPRL protocols, where it can be measured by calculating the reduction ratio (RR), pairs completeness (PC), and pairs quality (PQ). In order to calculate these measures, we need four counts which are: (1) SM which is the total number of true matches generated by the indexing step, (2) SN which is the total number of true non-matches generated by the indexing step, (3) NM which is the total number of all true matches, and (4) NN which is the total number of all true non-matches [184].

However, in this thesis, we focus on the time complexity rather than the indexing (blocking) step of our proposed PPRL techniques. Thus, we measure our scalability by using Big-O notation [135], as we describe next.

1. **Reduction ratio**
   Reduction ratio (RR) measures the performance of the indexing process by considering the proportion between the number of candidate record pairs that are generated during the indexing step and all possible record pairs [33, 182]. Reduction ratio is calculated as:

$$RR = 1 - \left(\frac{SM + SN}{NM + NN} \right). \tag{2.16}$$

2. **Pairs completeness**
   Pairs completeness (PC) corresponds to recall [38]. It measures the effectiveness of the indexing process to generate true matches of candidate record pairs, where a lower PC value means many true match pairs of records were removed during the indexing process [33, 182]. Pair completeness is calculated as:

$$PC = \frac{SM}{NM}. \tag{2.17}$$

3. **Pairs quality**
   Pairs quality (PQ) corresponds to precision [38]. It measures the quality of candidate record pairs that are generated during the indexing process [33, 182]. Pairs quality is calculated as:

$$PQ = \frac{SM}{SM + SN}. \tag{2.18}$$

4. **Time complexity**
   In computer science, Big-O notation, also called Landau's symbol, is generally used to evaluate the time complexity of any process [135]. The Big-O describes asymptotic behaviour to demonstrate the time that is used in a process [38, 135]. In this thesis, we evaluate the scalability of our proposed PPRL techniques by measuring the runtime of our PPRL linkage protocols using Big-O notation.

   The examples of Big-O notation of a process with the size of n inputs are:

   - $O(1)$ describes a constant time complexity, where a process always consumes the same amount of time unrelated to the size of an input.
   - $O(n)$ describes a linear time complexity, where the runtime grows directly in proportion to n inputs.
   - $O(n^2)$ describes quadratic time complexity. A process that has nested iterations often consumes $O(n^2)$ time complexity. However, when more nested iterations are included in a process, more runtime is used. For example, a process with three nested iterations requires cubic ($O(n^3)$) time complexity.
   - $O(log(n))$ describes logarithmic time complexity. A very efficient process often consumes $O(log(n))$ time complexity which is less than linear time.
   - $O(k^n)$ describes exponential time complexity, where k is a constant number and $k > 1$. A process that consumes $O(k^n)$ time complexity is considered as not efficient and scalable.

2.5.3 Privacy

Privacy measures the information that an adversary can gain from the encoded strings and the disclosure risk that it can correctly re-identify the original plaintext values from the encoded values [38, 182].

1. **Entropy and information gain**
   Entropy and information gain (IG) measure the amount of information that an adversary can infer about a plaintext value by gaining access to its corresponding encoded value [184].

   (a) **Entropy**
   Entropy, $H()$, is a function of the probability distribution over the set of all possible random values in a discrete random variable R [38, 93], where a higher entropy of R means it is more difficult to infer the original plaintext values from their corresponding encoded values. [93]. The entropy of R is defined as:

$$H(R) = -\sum_{r} P(r)\ log_2\ P(r) = \sum_{r} P(r)\ log_2\ \frac{1}{P(r)}, \tag{2.19}$$

where each r is a possible outcomes with probability $P(r)$. The entropy of encoded string values in a database $\mathbf{D}$ that match with string values in a global database $\mathbf{G}$ [38] is defined as:

$$H(\mathbf{D}) = -\sum_{x \in \mathbf{D}} \left(\frac{n}{|\mathbf{G}|}\right) \cdot log_2 \left(\frac{n}{|\mathbf{G}|}\right), \tag{2.20}$$

where n is the number of string values in $\mathbf{G}$ that match with a string value x in $\mathbf{D}$, and $|\mathbf{G}|$ is the total number of string values in $\mathbf{G}$. Given the encoded version of $\mathbf{D}$ is $\mathbf{E}$, the conditional entropy of a database $\mathbf{D}$ given $\mathbf{E}$ is defined as [182]:

$$H(\mathbf{D}|\mathbf{E}) = -\sum_{e \in \mathbf{D}} \left(\frac{n}{|\mathbf{G}|}\right) \cdot H(\mathbf{D}|\mathbf{E} = e), \tag{2.21}$$

where n is the number of encoded string values in $\mathbf{G}$ that match with an encoded string value e in $\mathbf{E}$, and $|\mathbf{G}|$ is the total number of encoded string values in $\mathbf{G}$.

(b) **Information gain**
Information gain (IG) [38, 93] measures the difficulty of inferring the original plaintext values in a database $\mathbf{D}$, where a lower IG value means it is more difficult to re-identify the plaintext values [38, 184]. Given the encoded version of $\mathbf{D}$ is $\mathbf{E}$, the IG between $\mathbf{D}$ and $\mathbf{E}$ is defined as:

$$IG(\mathbf{D}|\mathbf{E}) = H(\mathbf{D}) - H(\mathbf{D}|\mathbf{E}), \tag{2.22}$$

where the $H(\mathbf{D}|\mathbf{E})$ is the conditional entropy between $\mathbf{D}$ and $\mathbf{E}$ [93, 182] (Equation 2.21). The value of IG can be normalised into the range 0 to 1, known as relative information gain (RIG) [93] which is defined as:

$$RIG(\mathbf{D}|\mathbf{E}) = \frac{IG(\mathbf{D}|\mathbf{E})}{H(\mathbf{D})}. \tag{2.23}$$

2. **Disclosure risk**
Disclosure risk (DR) [182] refers to the likelihood of correct re-identification of the original plaintext values. The result of DR is normalised into the range 0 to 1, where 0 means absolute privacy (no disclosure risk) and 1 means provable exposed risk (an adversary can correctly re-identify plaintext values). DR is measured by calculating the probability of suspicion, P_s, of an encoded value in the encoded database $\mathbf{E}$, $e \in \mathbf{E}$, that matches with n global string values in a global database $\mathbf{G}$. The P_s of e is defined as:

$$P_s(e) = \frac{\frac{1}{n} - \frac{1}{|\mathbf{G}|}}{1 - \frac{1}{|\mathbf{G}|}}, \tag{2.24}$$

where $|\mathbf{G}|$ is the total number of string values in $\mathbf{G}$. Different DR measures of encoded value e can be defined based on P_s, where these measures are:

(a) **Maximum risk**
Maximum risk (DR_{max}) [38, 182] measures the maximum risk of disclosure of $e \in \mathbf{E}$ which is calculated as:

$$DR_{max} = \max_{e \in \mathbf{E}}(P_s(e)). \tag{2.25}$$

(b) **Marketer risk**
Marketer risk (DR_{mark}) [38, 182] measures the number of encoded values $e \in \mathbf{E}$ that are correctly re-identified when their $P_s = 1$. The DR_{mark} is calculated as:

$$DR_{mark} = \frac{|\{e \in \mathbf{E} : P_s(e) = 1.0\}|}{|\mathbf{E}|}. \tag{2.26}$$

(c) **Mean risk**
Mean risk (DR_{mean}) [38, 182] measures the average of DR which is calculated as:

$$DR_{mean} = \frac{1}{|\mathbf{E}|} \sum_{e \in \mathbf{E}} P_s(e). \tag{2.27}$$

(d) **Median risk**
Median risk (DR_{med}) measures the median of distribution of P_s in $\mathbf{E}$. It is calculated as:

$$DR_{med} = \begin{cases} \frac{1}{2} \times [P_s((e)_{|\mathbf{E}|/2}) + P_s((e)_{|\mathbf{E}|/2+1})] & if\ |\mathbf{E}|\ is\ even, \\ P_s((e)_{(|\mathbf{E}|+1)/2}) & otherwise. \end{cases} \tag{2.28}$$

(e) **User acceptance mean risk**
A user accepts that $e \in \mathbf{E}$ is not at DR if it matches with more than a number of unique string values in a global database $\mathbf{G}$. The encoded values $e \in \mathbf{E}$ that are below the respective P_s can be removed.

2.6 Experimental Setup

We discuss the datasets and implementation environment that are used in the experiments of our proposed PPRL techniques as following:

- **Datasets**
  We conducted our experiments on both real-world and synthetic datasets, where we used different datasets for evaluating our PPRL techniques depending upon the aims of the research, as we discussed in Chapter 1. For real-world datasets, we used data from the North Carolina Voter Registration (NCVR) database[1]

[1] http://dl.ncsbe.gov/

from snapshots of 2011, 2016, and 2019, where we used 100,000 records from attributes first name, middle name, last name, street address, city, zip code, and telephone number. For synthetic datasets, we used the Python package *Faker*[2] to create 100,000 strings for credit card, barcode, and IBAN (International Bank Account Number) numbers datasets. We describe datasets that we used for each of our PPRL techniques in its corresponding chapters (Chapters 4 to 7).

- **Implementation environment**
  We implemented our proposed PPRL techniques by using Python, where for our PPRL hardening technique for BFs, PPRL for databases that contain missing values, and PPRL for accurate sub-string matching approaches (Chapters 4 to 6), we used Python 2.7 and for our PPRL for fast and high linkage quality of string matching (Chapter 7) we used Python 3.6. For all experiments, we ran our implemented techniques on a server with 2.4 GHz CPUs running Ubuntu 18.04.

2.7 Chapter Summary

In this chapter, we described record linkage and the steps of the linkage process which are data cleaning and standardisation, indexing, comparison, classification, and evaluation. We then described PPRL and its details including protocols, adversary models, and techniques. Next, we described Bloom filter (BF) based approaches in details including the hashing method, different BF approaches, cryptanalysis attack, and different hardening techniques that are used in BF encoding.

After that, we described Big data and its four characteristics (four Vs) which are volume, velocity, variety, and veracity as they are related to the challenges of PPRL. Then, we described evaluation techniques that are used in record linkage and PPRL contexts. Finally, we described datasets and the implementation environment that we used to evaluate our proposed PPRL techniques in this thesis. In the following chapter, we explain the existing research works that are related to this thesis.

[2] https://pypi.org/project/Faker/

Chapter 3

Related Work

In this chapter, we describe the current progress in record linkage and privacy-preserving record linkage (PPRL) related to this thesis. In Section 3.1, we describe existing approaches proposed for string matching, and in Section 3.2 we discuss approaches proposed for sub-string matching. We then discuss linkage approaches for improving the time complexity of string and sub-string matchings in Section 3.3. After that, we describe approaches proposed for numerical data matching in Section 3.4. In Section 3.5, we discuss approaches that use reference databases for improving the linkage quality of a PPRL protocol. We then discuss existing approaches proposed for linking databases that contain missing values in Section 3.6. Finally, in Section 3.7, we provide a summary of this chapter.

3.1 PPRL for String Matching

As we described in Chapter 2, Bloom filter (BF) encoding [155] is widely used for (approximate) string matching. However, there are other techniques that have been proposed for string matching [139, 145, 170]. Randall et al. [147] were the first who proposed to use Bloom filter (BF) encoding [155] to match large patient databases instead of using the traditional probabilistic linkage such as the approach proposed by Fellegi and Sunter [67]. Their evaluation results show that the BFs and probabilistic linkage are equally effective in terms of linkage quality, thus, the BFs can be used for linking large databases.

Later, Randall et al. [148] evaluated the linkage quality and privacy of BF encoding [140, 155] compared to the standard statistical linkage key (SLK)[101] and encoded SLK. The SLK technique uses plaintext record values to generate a key by concatenating the second and third letters of attribute first name, second, third, and fifth letters of attribute last name, the full value of attribute date of birth, and a value of either "1" (male) or "2" (female) for attribute gender. For example, a record with attribute values "*mary*" (first name), "*miller*" (last name), "03/10/1987" (date of birth), and "*female*" (gender) will generate the SLK as "*arile*031019872". The evaluation results show that the encoded SLK and BFs provide higher privacy than the standard SLK, while BF encoding provides higher linkage quality than both standard and encoded SLKs.

In 2019, Randall et al. [145] proposed a PPRL approach based on the concatenation of attributes, called match-keys. In this approach, they first define a threshold and a secret key. For each record in a database, they generate a list of match-keys, where each match-key value is a concatenation of different attribute values in a record. They then use the key-hash message authentication code (HMAC) [9] with the agreed secret key to encode each match-key value, resulting in a list of hash encodings. For example, a record contains attributes first name, last name, gender, and birth year with values "*mary*", "*miller*", "*f*", and "1987", respectively. They generate the list of match-key values as ["*marymillerf*1987", "*marymiillerf*", "*marymiller*"]. The match-key values are then encoded by using HMAC hash encoding and the agreed secret key, resulting in the list $[HMAC(\text{“}marymillerf1987\text{”}), HMAC(\text{“}marymillerf\text{”}), HMAC(\text{“}marymiller\text{”})]$.

To compare the lists of encodings of a pair of records, they use the expectation-maximisation (EM) algorithm [193] as proposed by Fellegi and Sunter [67] to calculate scores between match-keys at the same position of the two lists in a pair. They then conduct a summation of all calculated scores. If the total score is higher than the agreed threshold, they classify the pair of records as a match, otherwise, they classify the pair as a non-match. However, the frequency based privacy attack approach proposed by Vidanage et al. [187] has shown that their attack can successfully re-identify the original concatenated attribute values (match-keys) from the hash encodings in the list.

As we described in Section 2.2.3, tabulation hashing (TMH) [170] and two-step hashing [139] techniques use hash functions to encode sensitive values. Smith [170] proposed a tabulation hashing (TMH) technique for secure string matching. The approach applies min-hash locality sensitive hashing [19] and uses Jaccard similarity function for comparing bit arrays. In this approach, k sets of n tables, are first generated, where k is the number of hash functions, and both k and n are numbers that are agreed upon by the database owners (DOs) participating in a linkage protocol. For each table in a set, it contains either 2^n or l random bits, where l is also agreed by the DOs.

To encode strings in a database into bit arrays, a DO first converts each string into a q-gram set, and each q-gram (sub-string of length q characters) in this set is then encoded into k different hash values by using k hash functions. After that, the DO splits each hash value into n values, where each value is then used as a key to select a random bit in the corresponding table in the set of tables. Once the DO has selected random bits for all hash functions, it selects the least significant bit for each hash function. Finally, the DO concatenates all selected significant bits into a final bit array of length l. The TMH technique has high computational costs due to the generation of multiple hash tables used in the encoding process. However, it is considered as an alternative BF encoding [155] because it provides more accurate linkage results than BFs.

Ranbaduge et al. [139] proposed a two-step hashing approach for linking records between databases. The two steps are (1) hash encode q-grams into bit arrays and

(2) hash encode bit arrays into integer numbers. In this approach, the DOs that participate in the protocol first make an agreement on parameters to be used in the protocol which are the length of the bit array, l, k hash functions for bit array encoding, hash function h for integer encoding, and secret salt value, s. Each DO then converts its string into a q-gram set, $\mathbf{q}$. The DO encodes q-grams in $\mathbf{q}$ by using hash function h_i, where $0 \leq i < k$, resulting in k bit arrays, each of length l. These bit arrays can be conceptually considered as a $k \times l$ matrix, where each bit array is one row of the matrix. As a result, bits in the same position of k bit arrays are located in the same column of the matrix.

The bits in each column across all rows in the matrix then become a bit array. If this bit array contains at least 1-bit, the DO converts the bit array, column index (bit position), and the agreed secret salt value to strings. These strings are then concatenated into one string. The DO uses the agreed hash function h to encode this string into an integer number. Alternatively, the DO can use the generated string as a random seed and then uses this seed to generate an integer number. The integer numbers of the two DOs are used for calculating similarity to classify if the pair of strings is a match, where in this approach the authors used Jaccard similarity calculation. The evaluation of this approach on two real-world databases shows that it outperforms the BFs [155] and TMH [170] in terms of linkage quality while outperforming the TMH [170] in terms of scalability.

3.2 Record Linkage and PPRL for Sub-string Matching

Sub-string matching is a common problem for many application domains such as bioinformatics [167, 189] and cloud computing [14, 23] applications. These applications require positions of sub-strings that occur in strings to find the longest matching sub-string of a query string in large databases. However, the algorithms used in such applications often have high computational complexities due to complex functions used and a large number of sub-strings kept in the process.

In this section, we discuss existing linkage approaches for sub-string matching, which we group these approaches into two categories: (1) suffix tree and array and (2) homomorphic encryption based approaches.

3.2.1 Suffix Tree and Array based Approaches

Suffix trees [121] are often used for sub-string matching in bioinformatics applications to search sub-sequences in genome databases [191]. A suffix tree allows searching for a given sub-string with linear complexity in terms of the length of the query string that is being searched [121]. Ukkonen [174] showed how suffix trees can be used efficiently for string matching, however, the approach requires more storage space to hold a suffix tree than the original string collection.

Wang et al. [191] proposed a string matching protocol based on suffix tree and edit distance constraints. In their approach, they find all similar sub-strings for a

given query in a collection of strings, such that the edit distances of these sub-strings with the query are within a given threshold. They improve the efficiency of suffix tree generation by employing the Burrows-Wheeler Transformation [21] (BWT) to index the string collection. Query strings are first partitioned into segments where each segment is queried to find exact matching sub-strings to generate a group of candidate strings. Due to the partitioning of query strings, some segments can result in large edit distances which potentially can lead to missed matching strings.

Suffix trees and suffix arrays can also be used with the longest common prefix array to address sub-string matching [8, 104]. Babenko and Starikovskaya [8] use suffix arrays combined with the longest common prefix arrays to facilitate the longest common sub-string searching in suffix trees. In their approach, they first merge the two strings to be compared by using a special character such as "$". They then generate and sort suffix and longest common prefix arrays. These arrays are then compared using either a sliding window or tree based approach to achieve a linear time complexity in the lengths of the two strings that are being compared.

Similarly, Kimura et al. [104] proposed an approach based on suffix and longest common prefix arrays of sub-strings. In their approach, sub-strings in a database are extracted, where sub-strings with frequencies higher than a given threshold and of a minimum length are used as indexes for sub-strings matching. The processing time of this approach crucially depends upon the frequency and length threshold parameters used, where a longer minimum string length will reduce the success of sub-string matching.

The use of suffix trees in PPRL for sub-string matching has been investigated by Chase and Shen [23]. Their approach constructs a queryable encryption scheme for finding all occurrences of a query string by a client in the encrypted suffix tree stored on an untrusted server. However, this approach reveals information about client queries to the server which compromises the privacy of a client's data and can potentially lead to the identification of the encrypted string values.

Chen et al. [25] proposed a secure pattern matching approach based on suffix arrays and order hashing, where each hashed character in a string is concatenated with the hashed value of the next character in the string. A DO sends the encrypted data to an untrusted server and transmits the key to the clients. This key is then used by the clients for verifying if the encoded query sub-string results received from the server are correct. However, in this approach, the server can learn the actual string length from the encrypted suffix array received from the DO.

3.2.2 Homomorphic Encryption based Approaches

Homomorphic encryption based approaches have been proposed for PPRL sub-string matching [66, 167, 201], while some approach uses homomorphic encryption to encrypt suffix tree [24]. Chen et al. [24] proposed an application that homomorphically encrypts a suffix tree to be stored on a server and allows searching sub-strings in the suffix tree. They transform a suffix tree to a dictionary for a more effective

searching process. For querying a sub-string, clients first encrypt their query sub-strings and send them to the server. The server finds the indexes of the sub-strings that match with the query sub-strings, then returns these indexes to the clients. However, the limitation of this approach is that the server can learn query information from clients by knowing the lengths of query sub-strings.

Shimizu et al. [167] proposed an approach for searching similar string patterns in a genome database using a recursive oblivious transfer protocol based on additive homomorphic encryption [70]. They used the Burrows-Wheeler Transform (BWT) and Positional BWT (PBWT) data structures for querying sequences in a large genome database while ensuring no query leads to the identification of other similar strings in the query database. Clients first generate public and private keys and then send the public key to the server. Clients use the public key to encrypt their genome sequences which are then ranked and counted the occurrences of letters in sequences. These ranked and counted numbers are then sent to the server for generating lookup tables and random integers for finding the longest matching sequences. The server uses the rotated permutation function for encryption which rotates positions in the encrypted values. As a result, the server cannot identify other similar strings in the client's database. However, the approach does not scale to queries of long sequences because these incur high computational and communication costs due to the complex cryptographic functions used [167].

Essex [66] proposed a secure two-party protocol using the Damgård-Geisler-Krøigaard (DGK) homomorphic encryption [47] method and private set intersection cardinality. In this approach, each DO first generates a list of all possible q-grams (based on letters a to z), where a q-gram in this list is replaced by the encryption of 1 if such q-gram occurs in a string of a DO, otherwise, a q-gram is replaced by the encryption of 0. The first DO sends its list to the other DO to conduct the set intersection cardinality and calculate the Dice-coefficient [33] based on the lengths of the q-gram lists of the pair of strings, then returns the results to the first DO for decrypting, where the decryption of 1 means a pair of strings is classified as a match. However, due to the use of the lengths of the lists of q-grams to calculate the similarity values in this approach, it can lead to false matches because some q-grams in the two lists may not be common. Furthermore, this approach consumes a lot of memory as the list of all possible q-grams for each string needs to be kept in memory for the comparison process.

Recently, Zarazadeh et al. [201] proposed a protocol for secure pattern matching on a client and server architecture using the ElGamal homomorphic encryption [64] and bit arrays. This approach is able to match either exact or approximate patterns with or without wildcard characters, or patterns with random bit vectors added (for hiding the length of strings). The server cannot learn anything about the pattern matching results. However, the clients can learn the positions of a matched pattern in a string in the database stored on the server.

3.3 Record Linkage and PPRL for Fast String and Sub-string Matchings

As we described in Chapter 1, some organisations require record linkage and PPRL techniques that provide fast string or sub-string matching results [96]. Blocking (and clustering) techniques have been proposed to speed up the computation time of the comparison process for large scale databases by reducing comparison space [96, 97]. Suffix trees and homomorphic encryption based approaches, as we described in Section 3.2, are widely used for sub-string matching. However, most homomorphic encryption based techniques have high computation time and costs [125]. Therefore, not many research works have used these techniques for fast string matching.

An example of homomorphic encryption based approach for fast string matching was recently proposed by Nakagawa et al. [125] to improve the time complexity and communication costs of genome sequence matching. They use a recursive oblivious transfer technique and a compressed indexing data structure [69] to find the longest prefix and longest exact match of a query sequence in a genome database. In this approach, the time complexity depends upon the length of the genome sequence that is being queried rather than the size of the genome database. Therefore, it consumes less time to query a sequence from a large genome database compared to the approaches proposed by Shimizu et al. [167] (as we discussed earlier in Section 3.2.2) and Sudo et al. [171].

In this section, we describe blocking techniques for fast string matching. We then discuss suffix tree based approaches that have been proposed for improving the comparison time of string and sub-string matchings. In addition, we discuss the existing works for fast string and sub-string matchings based on q-grams (sub-string of length q characters).

3.3.1 Blocking (and Clustering) Techniques for Fast String Matching

Some techniques have been developed for large scale databases with the aim to reduce the time complexity in linkage protocols [120, 168, 178]. McCallum et al. [120] proposed a clustering technique for large databases. In their approach, they first generate sets of strings that have common sub-strings between sets based on two distance thresholds t_1 and t_2, where $t_1 > t_2$. These sets are called canopies.

To generate each canopy, they first select a string from a database and calculate distances between the selected string and other strings in the database. The strings that have distances within the threshold t_1 are inserted into a canopy, while the strings that have distances within the threshold t_2 are removed from being selected to generate another canopy. They repeat these steps until every string in the database has been processed. As a result, they have multiple canopies that have common sub-strings (distance within the threshold t_1) while strings in any canopies that are not possible to have common sub-strings with strings in other canopies are excluded.

They then calculate distances between strings in canopies and combine canopies that have the closest distances into a cluster. The idea of this approach is to exclude strings that have distances within the threshold t_2 from the cluster generation process because these strings are higher similar, thus, it can reduce the number of distance calculations and improve the time complexity of cluster generation.

Shin et al. [168] proposed an algorithm for grouping similar values (nodes) in Big data networks. They construct a network based on the similarity between values (nodes), v and w, where the similarity is calculated by using the common neighbouring values (nodes) that are linked in between, v and w. This similarity is calculated as:

$$sim(v, w) = \frac{|nei(v) \cap nei(w)|}{\sqrt{|nei(v)| \; |nei(w)|}}, \tag{3.1}$$

where $nei(v)$ is the set of neighbouring nodes of the node v and $nei(w)$ is the set of neighbouring nodes of the node w.

The network structure is reconstructed by pruning the edges where the similarity values between nodes are below a certain threshold. The nodes are labelled as core nodes when the similarities of their edges are higher than the threshold, otherwise, the nodes are labelled as non-core. The core nodes will be used to conduct the clustering process. The core nodes and their linked nodes (both core and non-core nodes) are assigned to the same cluster. Edges are removed if the non-core nodes are linked to other non-core nodes. The authors evaluated runtime compared with the Markov clustering algorithm (MCL) and the results show that their algorithm is faster than the MCL algorithm.

Beneventano et al. [12] proposed a blocking technique based on min-hash locality sensitive hashing (LSH) [111]. In this approach, they first insert record values into a table, where each row contains one record and columns contain different attributes. They then apply min-hash LSH [111] and calculate the similarities between record values in different rows. After that, they group records with similarities higher than the threshold, t_1, together, resulting in multiple groups of records. For each generated group, they iteratively apply min-hash LSH to each attribute (column) of records and calculate the similarities between attribute values of these records. The record pairs with similarities of all attributes are higher than the threshold, t_2, are then inserted into the same block. With calculating similarities for both rows and columns, records that have similar attribute values are inserted into the same block.

Karapiperis et al. [97] proposed a method for real-time (online) string matching that can be applied to any blocking techniques. In this approach, a set of blocks are first generated by using a blocking technique, where each block has its blocking key value, *bkv*. Each *bkv* is then inserted into a slot, called uniblock, along with a frequency of access. Every time the block is accessed, the frequency of access is counted and updated. When all blocks are accessed or new records are generated but they cannot be inserted into the existing blocks, new blocks are generated to accommodate the new records. The uniblock then releases less frequently accessed blocks to free up the memory space for allowing the new blocks to be inserted. The

idea of this approach is to keep frequently accessed blocks in the main memory. Therefore, the string matching protocol has fast computation because it minimises the access to the secondary storage which can slow down the linkage process.

Although blocking techniques can reduce comparison space, thus, reducing time complexity in linkage protocols, the frequency distribution of records in blocks can be vulnerable to frequency based attacks [40, 182]. Recently, Wu et al. [196] proposed two PPRL techniques based on differential privacy [60, 62] and Bloom filter (BF) encoding [155] to consider privacy, fairness, and cost in PPRL.

In both techniques, they first generate blocks and then generate BFs of records in each block. The first technique is fairness differential privacy. They create disjoint groups in a block. They then insert noises (dummy) records into the block by ensuring the groups of the block must provide equally probability of outcomes, for example, two groups of a block providing the same number of true matches. These noises are generated by flipping bits in the BFs and ensuring that the number of noises must guarantee privacy to prevent frequency reveal to an adversary. The second technique is cost constraint differential privacy. This technique inserts noises into a block in a way that the additional cost (which is caused by adding noises into a block) and fairness loss are minimised. Their experimental results show that their two proposed techniques outperform the standard differential privacy [110] in terms of privacy, fairness, and cost of the linkage protocol.

3.3.2 Tree based Approaches for Fast String and Sub-string Matchings

As we discussed in Section 3.2.1, various approaches for sub-string matching use the suffix tree based technique [23, 25]. Chan et al. [22] proposed pruning techniques to reduce the size of suffix trees generated from large string databases. Their approach aims to improve the querying of strings by pruning infrequent sub-string patterns and duplicate paths in a tree. However, pruning sub-strings with shorter lengths results in some string patterns not being matched. Similarly, Patil et al. [133] proposed a method that combines length and position filtering techniques for pruning suffix trees and inverted lists of sub-strings which results in a reduction of the query time in the matching process.

Wang and Li [190] used a suffix tree to find similar sub-strings between groups of sub-strings. In this approach, suffix trees are generated first, and overlapping paths (sub-strings) are then searched. Based on these overlapped paths, they calculate the Cosine similarity by using the weights that are calculated based on the term frequency/inverse document frequency (TF-IDF) [194] of non-overlapping paths. The authors then rank the similarities to find the highly similar groups of sub-strings.

In the online application domain, a suffix-tree based method was proposed by Pei et al. [134], where the aim of this approach is to find the shortest unique sub-string query. In this approach, suffix trees are employed because they can be used to find left-bound shortest unique sub-strings in constant time which helps to improve the efficiency of online query applications.

Recently, Yu et al. [200] proposed a top-k similarity search based on tree. They first generate an index tree for a database by grouping the strings that share a common sub-string. They then calculate similarity values between strings in a group and use a string with the highest similarity as a parent node of a tree while the other strings are used as a child node of this parent node. They iteratively generate nodes until every string record is added to the tree.

To search a query string, they generate a priority queue and define a threshold, t. They find the nodes (strings) of the tree in the queue that may match the query string by calculating the distance and counting common sub-strings between the query string and a node of the tree, where the counted number is used to calculate a bound between query string and node. If the distance is larger than $1 - t$, and the count number of common sub-strings is smaller than the calculated bound, they prune this node from the queue. Otherwise, they calculate the similarity between the query string and node string using Jaccard similarity calculation. If the similarity value between query and node is higher than the threshold t, they add the node to a set of matches.

3.3.3 Q-gram based Approaches for Fast String and Sub-string Matchings

Q-gram is a sub-string of length q that can be used for string matching. Gravano et al. [78] proposed an approximate string join technique based on q-grams of strings. They conduct three different filtering techniques which are count filtering, positional filtering, and length filtering techniques. Count filtering compares string pairs that the edit distance between strings is within a certain threshold. Positional filtering compares string pairs that the numbers of positions of common q-gram between strings do not differ more than a certain threshold. Length filtering compares string pairs that the lengths of strings in a pair do not differ more than a certain threshold.

Kim et al. [103] proposed an approximate string query based on positions of sub-sequences and q-grams of sub-sequences that occur in strings in a database. In this technique, they first generate sub-sequences of strings in a database. They then generate sub-sequence identifiers and insert sub-sequences and their identifiers into a table, called back-end table. For each sub-sequence in the back-end table, they insert a list of positions and string identifiers that the sub-sequences occur in the corresponding strings. They then generate q-grams from sub-sequences in the back-end table and insert them as a key into another table, called front-end table. For each key, they insert a list that contains pairs of sub-sequence identifiers and positions of q-grams that occur in sub-sequences as a value.

To query a string, they first generate q-grams from the query string. They then find the pairs of sub-sequence identifiers and positions of q-grams that corresponded to each q-gram of the query string from the front-end table. After that, they conduct an outer join merge between the sub-sequence identifiers they found and the sub-sequence identifiers in the back-end table, resulting in the common sub-sequence identifiers. Finally, they find the set of strings that the query string is matched with

by using the common sub-sequence identifiers that they found from the merged sub-sequence identifiers.

Bonomi et al. [18] proposed a PPRL approach to compare string values using bit arrays based on the embedding of frequent q-grams. The DOs that participate in the PPRL protocol individually apply differential privacy [60] to generate a table of frequent q-grams that occur in their databases. The DOs then send their tables to one of the DOs to find the common frequent q-grams (shared frequent q-grams) and send them to all DOs that participate in the protocol. Each DO then uses the shared frequent q-grams to embed their strings into bit arrays, and sends these bit arrays to a third party to compare pairs of bit arrays.

Hahn et al. [79] proposed a privacy-preserving secure sub-string (q-gram) querying approach where the frequency distributions of sensitive information are hidden by applying a frequency-hiding order preserving encryption [102]. This approach involves three parties for processing a secure q-gram query, which are (1) the DO who owns the sensitive information, encodes the q-grams, and generates the encoded data tables (indexes and q-grams), (2) the untrusted party who holds the index tables that are used for searching encoded q-grams, and (3) the clients who want to query their encoded q-grams. In this approach, the complexity of querying depends upon the q-gram length because it determines the number of iterations required for querying encoded q-grams from the untrusted party.

3.4 PPRL for Matching Numerical Values

Apart from PPRL approaches for string and sub-string matchings we described in previous sections, some approaches have been proposed for comparing numerical values [28, 114]. Vatsalan and Christen [181] proposed a technique for encoding numerical values based on Bloom filter (BF) encoding [155]. In this approach, a BF is generated based on a list that contains a number, x, to be encoded and the neighbouring values of x.

To find the neighbouring values of x, they first define the maximum difference between two numerical values, d_{max}, and the value n to be used for calculating the interval between numbers. They then set the range of numbers in the list as $[x - d_{max}/2, x + d_{max}/2]$, and calculate the interval between every two consecutive numbers in the list as:

$$d_{intv} = \frac{d_{max}}{2 \times n}. \tag{3.2}$$

For example, let us assume the number to be encoded is $x = 35$, $d_{max} = 4$, and $n = 2$. Therefore, the range of numbers in the list is $[35 - 4/2, 35 + 4/2] = [33, 37]$ and the interval between numbers is $d_{intv} = 4/(2 \times 2) = 1$. As a result, the list to be encoded of the number $x = 35$ is $[33, 34, 35, 36, 37]$ with the length $l = 2 \times n + 1$.

Once the list has been generated, they then use k hash functions to encode the list into a BF [155]. In this approach, they use the Dice-coefficient [33] to calculate the

Table 3.1: Example of table **T**, where the first DO (DO_A) encrypts e^1 = "1001".

	$j=1$	$j=2$	$j=3$	$j=4$
$i=0$	$E(r)$	$E(1)$	$E(1)$	$E(r)$
$i=1$	$E(1)$	$E(r)$	$E(r)$	$E(1)$

similarity between two BFs. This approach can apply to different types of numerical data, such as integer (as we described above), floating-point, and modulus data.

Karapiperis et al. [94] proposed an approach for encoding numerical values based on binary Hamming space. In this approach, they first define a distance threshold, t, and a list of real numbers, $\mathbf{n}$, from the range of n_{min} to n_{max}, where n_{min} is the minimum number and n_{max} is the maximum number to be encoded. They then use the defined t to define the interval, $intv$, for each $n_i \in \mathbf{n}$ as $intv_i = [n_i - t, n_i + t]$. Each interval establishes a hash function that can be used for setting bit at position i in a binary vector of length $|\mathbf{n}|$ to 1 or 0. The hashing, $h_i()$, of a numerical value, x, is conducted as:

$$h_i(x) = \begin{cases} 1, & if\ x \in [n_i - t, n_i + t], \\ 0, & otherwise. \end{cases} \tag{3.3}$$

An evaluation of this approach on real-world databases shows that this approach provides higher recall than the approach proposed by Vatsalan and Christen [181].

Lin and Tzeng [114] proposed an approach to compare encrypted numerical values based on the sets of their special 1- and 0-encodings, where each of the two DOs that participate in the protocol generates different sets of encodings and each encoding is a binary string. One DO (we assume the DO_A) generates the set of 1-encodings, $\mathbf{E}_x^1$, for its integer number x and the other DO (we assume the DO_B) generates the set of 0-encodings, $\mathbf{E}_y^0$, for its integer number y. The idea behind this approach is to use these two sets $\mathbf{E}_x^1$ of the DO_A and $\mathbf{E}_y^0$ of the DO_B to conduct a comparison. If there are common encodings between the sets $\mathbf{E}_x^1$ and $\mathbf{E}_y^0$, then $x > y$, while if no common encodings between these sets, then $x \leq y$.

To generate the set of encodings, each DO first converts its integer number into a binary string, b, of length l. The DO_A then generates 1-encodings, e^1, and the DO_B generates 0-encodings, e^0, for their binary strings. After that, the DO_A inserts e^1 into the set $\mathbf{E}_x^1$ and the DO_B inserts e^0 into the set $\mathbf{E}_y^0$.

To conduct a secure comparison between $\mathbf{E}_x^1$ and $\mathbf{E}_y^0$, for each $e^1 \in \mathbf{E}_x^1$, the DO_A generates a $2 \times |e^1|$ table, $\mathbf{T}[i,j]$, where $i \in \{0,1\}, 1 \leq j \leq |e^1|$. For each character $e_j^1 \in e^1$, the DO_A generates a ciphertext $E(1)$ and inserts it into $\mathbf{T}[e_j^1, j]$. For any $\mathbf{T}[i,j]$ in $\mathbf{T}$ that does not contain $E(1)$, DO_A generates encryption of a random value, r, resulting in a ciphertext $E(r)$, as we illustrate an example in Table 3.1. DO_A sends $\mathbf{T}$ to DO_B. If $|e^0|$ of DO_B is less than $|e^1|$ of DO_A, DO_B conducts a multiplication over its ciphertext encrypted of a character in e^0 that has the same position (i,j) of the ciphertext encrypted of a character in e^1 in $\mathbf{T}$. For other positions (i,j) (not common

positions between e^1 and e^0), DO_B multiplies each ciphertext in these positions with a random ciphertext. DO_B permutes all ciphertexts and sends them to DO_A. DO_A decrypts the received ciphertexts received from DO_B, where if the decryption of a ciphertext is 1, DO_A concludes that its number, x, is larger than the number, y, of DO_B, $x > y$, and otherwise, $x \leq y$. However, using the table **T** for comparison between ciphertexts in this approach leads to high memory consumption.

Cheon et al. [26] proposed a homomorphic encryption scheme, called Cheon-Kim-Kim-Song (CKKS), that supports approximate arithmetic, such as additive and multiplicative operations, to be conducted over the ciphertexts to be compared. They also proposed a packing method that encrypts multiple numerical values into a single ciphertext, where they allow decimal numbers to be encrypted. In addition, they provide an open-source implementation of their approach, called Homomorphic Encryption for Arithmetic of Approximate Numbers (HEAAN) [26].

Later, Cheon et al. [28] proposed an approach that allows approximate comparison between ciphertexts, where the results of their approximate comparison contain $2^{-\alpha}$ of errors, where α is a precision of a ciphertext. To find a minimum between two numbers, x and y, that are encrypted into ciphertexts $E(x)$ and $E(y)$, respectively, they calculate a minimum $min(E(x), E(y))$ as:

$$min(E(x), E(y)) = \frac{E(x) + E(y)}{2} - \frac{|E(x) - E(y)|}{2} = \frac{E(x) + E(y)}{2} - \frac{\sqrt{(E(x) - E(y))^2}}{2}. \tag{3.4}$$

However, to calculate a square root value, they proposed to use a recursive method, resulting in increased time complexity. In 2020, Cheon et al. [27] proposed an approach to improve the time complexity of the comparison process proposed by Cheon et al. [28] by using polynomial composition functions, $v = f(v) \circ g(v) = f(g(v))$, where v is initialised as $v = E(x) - E(y)$. The result value of the composition function $f(g(v))$ can then be used for calculating the minimum value between ciphertexts $E(x)$ and $E(y)$ (as we describe in Chapter 7). The time complexity of the approaches proposed by Cheon et al. [26, 27, 28] depends upon the parameter settings that are used for encryption, decryption, and comparison processes in the protocol.

3.5 Reference Values used in PPRL Approaches

As we described in Chapter 1, the higher privacy of PPRL often results in lower linkage quality. Some approaches have been proposed to use a global (publicly available) database to improve the linkage quality of PPRL. Pang et al. [132] were the first who proposed to use record values from a global database as reference values to improve the linkage quality in a three-party PPRL protocol.

In their approach, the DOs individually compare their record values with values in a global database by using the edit distance [127]. The DOs then encode or encrypt their record values by using a hash encoding or encryption function and send the

encoded or encrypted values along with the calculated distance to a linkage unit (LU) to conduct a comparison. The LU compares record values between the two databases sent to it by the DOs, where a pair of records is being classified as a match if the distance between them is below a given threshold. However, the performance of this approach depends upon the number of record values in the reference database which needs to cover record values in the two databases that are being linked in the linkage process.

Later, Karakasidis et al. [91] used a reference database and the nearest neighbour clustering algorithm for generating blocks of record values to minimise the computational costs of a three-party PPRL protocol. In their approach, the DOs first agree on a reference database and a similarity threshold to be used in the protocol. Each DO then uses the Dice-coefficient similarity calculation [33] and the agreed similarity threshold to generate blocks of reference values, where similar reference values are inserted into the same block. The DO then generates block identifiers by ensuring they generate these identifiers in the same manner.

For each generated block, the DO calculates the similarity between reference values in the block and record values in its database using the Dice-coefficient [33]. The DO inserts its record values into the block if the calculated similarity is above the threshold. As a result, each block contains values that are indistinguishable whether they are reference values or DO's record values. Each DO generates an identifier for each value in a block, and sends all blocks to a LU. The LU finds common block identifiers and compares records that are in the blocks under the same identifiers. This results in minimising the cost of the comparison process. However, this approach consumes high computational costs in the blocks generation process, because each DO needs to calculate Dice-coefficient similarities [33] between values in the reference databases and also between its record values and reference values in all blocks.

Similarly, Vatsalan et al. [183] proposed an approach for generating blocks by using a reference database and the sorted neighbourhood clustering (SNC) [86, 179] technique. In their approach, the DOs make an agreement on a reference database, similarity threshold, and number of records per block. Each DO selects a set of reference values from the agreed reference database. The DO then inserts the selected set of reference values into a list and then sorts this list. The DO generates SNC blocks by inserting record values in its database based on sorting key values (SKVs) into the sorted list of reference values. If a number of records in a block is less than an agreed number of records per block, the DO merges this block with another block that contains similar reference values. The DOs then exchange their reference values that corresponded to their blocks. These reference values are then merged and sorted. After that, the DOs apply a sliding window to generate candidate clusters. Only records in the two databases that are in these candidate clusters will be compared.

Recently, Mullaymeri and Karakasidis [124] proposed a two-party PPRL protocol based on polynomial coefficients generated using a reference database. They are the first to apply the Fuzzy Vault scheme [88] in the PPRL context, where the Fuzzy

Vault scheme [88] is a cryptographic structure that allows decoding to be conducted if and only if the keys used for the decoding and encoding are very similar. The idea behind this approach is that if the set of keys (generated from reference strings in the reference database) of the two strings in a pair are similar, the polynomial coefficients generated from the keys of these two strings must be the same, and therefore the two strings in a pair should be classified as a match. The main drawback of this approach is that the reference strings that are used to generate keys must be very similar to the strings in a pair to be compared to ensure that the polynomial coefficients of the two strings are the same. Therefore, the number of reference strings must be large enough to allow the two parties to generate the same polynomial coefficients.

3.6 Record Linkage and PPRL for Linking Databases with Missing Values

As we discussed in Chapter 1, this thesis mainly focuses on the improvement of the linkage quality of PPRL. Although various PPRL approaches have been proposed to improve linkage quality as well as privacy (as we described in previous sections), missing values have always been a challenge that can cause to lower linkage quality. However, it has seen limited attention in PPRL as well as in the record linkage contexts [6, 30]. In this section, we describe the existing approaches proposed for linking databases that contain missing values.

Ong et al. [130] proposed three methods which are weight distribution, linkage expansion, and distance imputation to improve the accuracy of linking records that contain missing values in the context of probabilistic record linkage [67]. Their proposed methods were developed based on the Fellegi and Sunter calculation [67] and a distance measure, such as edit distance [127], that are available as an open-source software package called fine-grained record linkage (FRIL) [89].

1. **Weight distribution**
   In the weight distribution method, the weights, w, of the attributes that contain missing values are distributed to attributes with no missing values based on the relative importance in the linkage process of attributes with no missing values. To avoid linking only low weight attributes of a pair of records, the following inequality is first checked,

$$\sum_{i=1,\mathbf{a}[i]\neq\mathcal{M}}^{|\mathbf{a}|} w_i - \sum_{i=1,\mathbf{a}[i]=\mathcal{M}}^{|\mathbf{a}|} w_i > 0, \tag{3.5}$$

   where $\mathbf{a}$ is a set of attributes to be linked, $|\mathbf{a}|$ denotes the length of set $\mathbf{a}$, and $\mathcal{M}$ means missing value. Any pairs that do not meet the inequality are not being compared and these pairs are categorised as non-matches. For the pair

of records x and y that meet the inequality, the match score is calcualted as:

$$match(x, y) = \sum_{i=1, \mathbf{a}[i] \neq \mathcal{M}}^{|\mathbf{a}|} w_i \times d_i, \tag{3.6}$$

where d is the distance between the values of a certain attribute i of x and y.

2. **Linkage expansion**
   Linkage expansion generates two sets of attributes which are a primary set and a backup set. The primary set contains attributes to be linked between records in a pair and the backup set contains both attributes to be linked and not being linked that were selected by a domain expert due to the importance of such attributes in the linkage process. The weights of agreement and disagreement [67] are then calculated based on the expectation-maximisation (EM) [50] algorithm for both primary and backup sets. The backup set is then used to provide additional information when a missing value is found in an attribute in a primary set and the weight of an attribute with a missing value in a primary set is distributed to other attributes by using the weight distribution.

3. **Distance imputation**
   Distance imputation calculates a distance between attributes with no-missing values in a pair of records, then uses the calculated distance to impute distances for attributes with missing values to classify a pair of records as a match or non-match. Similar to the weight distribution, for each pair of records, the inequality (Equation 3.5) is first checked to avoid linking only low weight attributes. A set of imputation rules is then generated, where this set contains three parts: (1) the attributes with no-missing values, (2) the attributes with missing values, and (3) the imputed values of either 0 or 100, where 0 (or FRIL-0) is a non-match and 100 (or FRIL-100) is a match.

   To calculate an imputed value for each attribute with a missing value, a conditional probability is calculated based on the similarity between attributes with no missing value of the two records, x and y, in a pair. The conditional probability is calculated as:

$$P_c = P(x_{\mathcal{M}} \sim y_{\mathcal{M}} \mid \forall x_{i \neq \mathcal{M}} \sim y_{i \neq \mathcal{M}}), \tag{3.7}$$

   where $0 \leq i < n$, n is a number of attributes to be linked, and each $x_{i \neq \mathcal{M}} \sim y_{i \neq \mathcal{M}}$ is an approximate match if the probability value P_c is at least a certain threshold, t. An imputed value is 100 if $P_c \geq t$, and otherwise, the imputed value is 0.

Ong et al. [130] evaluated experimental results on simulated databases and concluded that the linkage expansion method performed best when compared with the other two methods (weight distribution and distance imputation), although it requires an experienced domain expert to select suitable backup attributes (if such

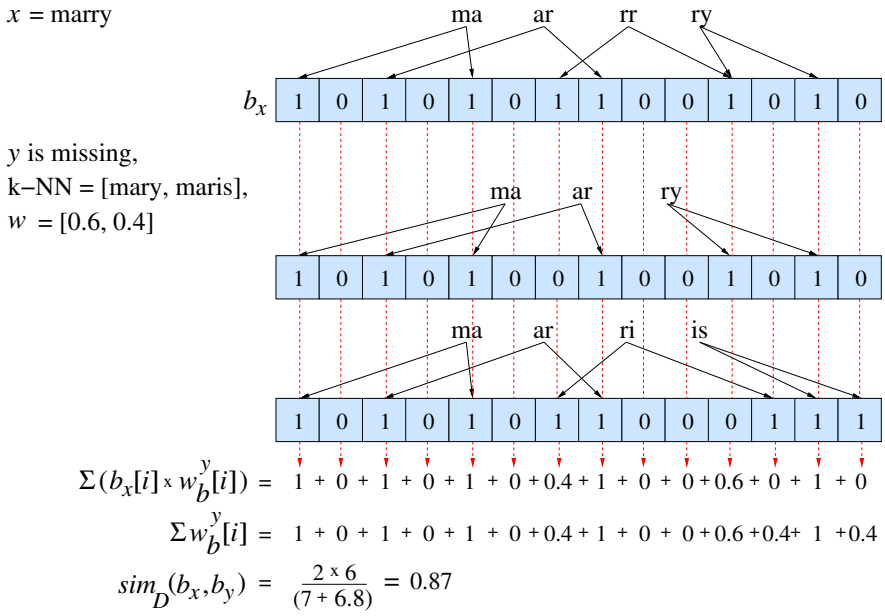

Figure 3.1: Example of k-NN based BFs comparison [30], where attribute value of the record y is missing and their k-NN attribute values are "*mary*" and "*maris*" with the weights 0.6 and 0.4, respectively. Red dashes show bits and weights that correspond to the same index position i in BFs, and the Dice-coefficient calculation (Equation 3.8) is used for calculating similarity between records x and y.

attributes are available in the databases to be linked). However, imputation has been generally proposed to use for linking databases that contain missing values.

In record linkage, Goldstein and Harron [76] used the multiple imputation technique to provide information for statistical analysis when the main database, called primary database, contains missing values. For each record with a missing value in a primary database, they impute values that refer to the same record from one or more databases, called secondary database, to the primary database. The imputation in this approach is based on the Markov Chain Monte Carlo (MCMC) [20] method and the conditional distribution that is derived from the joint distribution between primary and secondary databases.

Ferguson et al. [68] used the expectation-maximisation (EM) algorithm [50] to impute the value of non-missing attributes in a record in a pair to estimate the expected value which is used for classifying a pair of records as a match. They then used the EM algorithm again to predict a posterior probability, and this probability is then used with the estimated expected value to calculate a probability that a record pair should be classified as a match.

In the PPRL context, an imputation technique was proposed by Chi et al. [30]. In their approach, for each attribute of a record in a database, they generate an attribute-level Bloom filter (ABF) of length l [155]. If an attribute of a record contains a missing value, they use the Fellegi and Sunter [67] method to calculate a weight

score, w, between the record with a missing value and other records with no missing values in the same database. They then select the k-nearest neighbour (k-NN) records that have weights at least a threshold, t, $w \geq t$. The weights of these k records are used as the weight of the attribute of the record with a missing value.

To compare BFs between the two records in a pair, they apply the Dice-coefficient calculation [33] to calculate a similarity as:

$$sim_D(b_x, b_y) = \frac{2 \times \sum(b_x[i] \times w_b^y[i])}{\sum b_x[i] + \sum w_b^y[i]}, \tag{3.8}$$

where b_x is the BF of a record with non-missing value, b_y is the BF of a record with missing value, w is the total weight of the k-NN records corresponding to the record with a missing value (in this case is b_y such that the weight is w_b^y), and i is the index of bit or weight in the BFs, where $0 \leq i < l$ and l is the length of the BF. We illustrate the example of a similarity calculation of this approach in Figure 3.1.

However, this approach is based on attribute-level BFs (ABFs) [38] which is vulnerable to cryptanalysis attacks [39, 109, 129]. Furthermore, imputing similarity measures from the k-NN records requires groups of records with high similarities, such as families and households in census records, to improve the linkage quality.

Apart from the problem of lowering linkage quality, missing values also affect the block generation process when a missing value occurs in an attribute that is used as a blocking key [6]. Aninya et al. [6] analysed the impact of missing values in six blocking techniques which are standard blocking, sorted-neighbourhood, q-gram indexing, canopy clustering, string-map, and suffix array indexing [32, 33]. Their results show that the blocking techniques that insert each record into multiple blocks, such as canopy clustering and suffix array indexing [32, 33], perform best when the attributes used for blocking contain missing values, while suffix array based indexing is overall the best performing technique.

3.7 Chapter Summary

In this chapter, we described existing research works that relate to our research in this thesis. We first discussed approaches proposed for string matching. We then described record linkage and PPRL approaches for sub-string matching. Next, we discussed existing approaches proposed for fast string and sub-string matchings. After that, we explained approaches for numerical data matching. Then, we described approaches that use a reference database to improve the linkage quality of a PPRL protocol. Finally, we discussed existing works that proposed for linking databases that contain missing values. In the following chapter, we present our hardening technique proposed for improving the privacy and linkage quality of BF encoding for PPRL.

Chapter 4

Reference Values based Hardening for Bloom Filter based Privacy-Preserving Record Linkage

In this chapter, we discuss our proposed hardening technique to improve the degrees of privacy of Bloom filter (BF) encoding [155] and improve the linkage quality of the BLoom and flIP (BLIP) hardening technique which we described in Chapters 1 and 2. In Section 4.1, we provide an introduction to our proposed hardening technique. We then provide a brief description of the BLIP hardening technique in Section 4.2. In Section 4.3, we describe the protocol of our proposed approach, and in Section 4.4 we analyse our approach in terms of linkage quality, scalability, and privacy. We then discuss our experimental evaluation in Section 4.5. Finally, in Section 4.6, we summarise our proposed approach.

4.1 Introduction

As discussed in Chapters 1 and 2, Bloom filter (BF) encoding is a widely used perturbation technique in the privacy-preserving record linkage (PPRL) context [16, 155]. However, several researchers have shown that BFs can be attacked with the aim of re-identifying the sensitive plaintext values encoded in them [40, 42, 108, 109]. Most of these attack techniques exploit frequent BFs or bit patterns that correspond to frequent q-grams (sub-strings of length q) in the sensitive values that were encoded into BFs. To counteract such attack techniques, hardening techniques have been developed to improve the privacy protection level of BF encoding based PPRL techniques [157, 159], as we discussed in Chapter 2. Those hardening techniques modify BFs to reduce or even eliminate any frequency information that could be exploited by attack techniques. One drawback of existing hardening techniques is that they have a trade-off between privacy and linkage quality because modifications of BF bit patterns will likely lead to an increase in falsely matched and missed true matching record pairs. Certain hardening techniques have also been shown to be vulnerable to a frequency based cryptanalysis attack [40].

Table 4.1: Notation and terminology used in this chapter.

G	Global database
g	Global value in **G**
R	Set of reference values
r	Reference value in **R**
D	Database to be encoded
v	Plaintext value in **D**
E	Set of RBBF encoded values from **D**
H	Set of hash functions
A	Set of attributes
s_t	Similarity threshold for generating reference values
t	Similarity threshold for classifying a record pair as a match
$sim()$	A similarity function
sim^q_D, sim^b_D	Dice-coefficient between q-grams and Bloom filters, respectively
s_{max}	Maximum similarity value
q	Length of each q-gram
l	Length of a Bloom filter
k	Number of reference values per Bloom filter
m	BLIP hardening approach selection
f	BLIP flip probability
p	Bit position in a Bloom filter or hardened Bloom filter
b, b_q	Bloom filter
b', b_r, b_v	Hardened Bloom filter
$\|...\|$	Size or number of values in a database or set
BLIP	BLoom and flIP hardening technique
BLIP-A	BLIP technique proposed by Alaggan et al. [5]
BLIP-S	BLIP technique proposed by Schnell and Borgs [157]
RBBF	Reference based BLIP BF hardening technique
RBBF-A	RBBF based on approach proposed by Alaggan et al. [5]
RBBF-S	RBBF based on approach proposed by Schnell and Borgs [157]

One of the hardening techniques is BLoom and flIP (BLIP) [5, 157] which flips bit values at certain positions in a BF according to a differential privacy mechanism [60], as we described in Chapter 2 and briefly describe in the next section. However, such random bit flipping can decrease linkage quality, as we discuss in Section 4.5.

To overcome this weakness of the BLIP hardening technique, we use reference values from a publicly available global database to determine the bit positions to be flipped. The use of reference values ensures that similar BFs are modified in the same way while different BFs are modified differently. In other words, the idea of our approach is that the same plaintext values will likely have the same sets of reference values. When using these reference values to determine the bit positions to be flipped, the same BLIP based randomisation will be applied to the two BFs that encoded of the same plaintext value. As a result, the two (hardened) BFs are

classified as a match. We name our approach as Reference based BLIP BF hardening (RBBF). Table 4.1 provides the notation and terminology we use in this chapter.

4.2 BLoom and FlIP (BLIP)

We now provide a brief description of the BLoom and FlIP (BLIP) hardening technique. As described in Section 2.3, the BLIP hardening technique was originally proposed by Alaggan et al. [5] as a non-interactive differentially private [60] approach to randomising BFs in the context of privacy-preserving comparisons of user profiles in social networks. BLIP randomly flips a bit at a certain position, p, in a BF, b, based on a user-defined flip probability, f, as defined in Equation 4.1, resulting in a new bit at position p in the hardened BF.

$$b'[p] = \begin{cases} 1 & if\ b[p] = 0\ with\ probability\ f, \\ 0 & if\ b[p] = 1\ with\ probability\ f, \\ b[p] & with\ probability\ 1 - f. \end{cases} \tag{4.1}$$

Schnell and Borgs [157] were the first to explore BLIP in the context of PPRL. Their approach is based on the idea proposed by Erlingsson et al. [65], as part of the technique called RAPPOR, which allows anonymous collection of user statistics from software products such as web browsers. The RAPPOR technique randomly selects a bit to be set to 0 or 1 with equal probability as:

$$b'[p] = \begin{cases} 1 & with\ probability\ \frac{1}{2}f, \\ 0 & with\ probability\ \frac{1}{2}f, \\ b[p] & with\ probability\ 1 - f. \end{cases} \tag{4.2}$$

For example, the flip probability is set to $f = 0.05$ for a BF of length $l = 1,000$ bits. When using the approach proposed by Alaggan et al. [5] (Equation 4.1), 50 randomly selected bits will be flipped, while 950 bits are unchanged. With the approach proposed by Schnell and Borgs [157] (Equation 4.2), however, bits will not be flipped according to their original state, but rather 50 randomly selected bits will be set to 0 or 1 with equal probability. As a result, depending upon which BLIP approach is used, the numbers of 1-bits in the hardened BFs will likely differ.

If a BF has less than 50% 1-bits, then applying Equation 4.1 means it will have more 1-bits after hardening than when applying Equation 4.2. This can potentially lead to lower linkage quality because more 1-bits can increase the similarities between the hardened BFs, and thus leads to more false positive matches.

4.3 Reference based BLIP BF Hardening (RBBF) Protocol

In this section, we describe our proposed Reference based BLIP BF Hardening (RBBF) approach. As we illustrate in Figure 4.1, our approach includes two phases. In

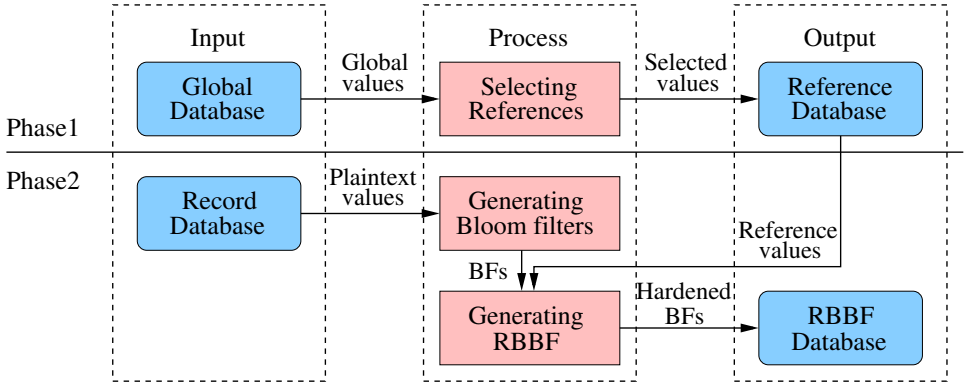

Figure 4.1: Overview of our Reference value based BLIP Bloom Filter Hardening.

the first phase, we select a set of suitable reference values from a publicly available database (global database). In the second phase, we generate an RBBF database by first generating a Bloom filter (BF) [155] for each record in a database to be encoded and select reference values for the generated BF. We then use either the BLIP hardening technique proposed by Alaggan et el. [5] or Schnell and Borgs [157] and the selected reference values for BFs to generate an RBBF database.

4.3.1 Selecting Reference Values

The database owners (DOs) [185] that participate in the PPRL protocol aim to encode their sensitive information. They first agree on the similarity threshold, s_t (we recommend $s_t \geq 0.8$), similarity function, $sim()$ (such as the Dice-coefficient), and they need to have access to the same publicly available global database **G**. The DOs then use s_t and $sim()$ to extract a set of reference values **R** from **G**. As a result, all DOs use the same set of reference values, **R**, for hardening their encoded sensitive information. The set **R** can be either generated in the same way by all DOs, or alternatively generated by one DO and distributed to all other DOs that participate in the PPRL protocol. This reference values selection process aims to find a set of string values that are all different to each other according to the agreed similarity threshold, s_t. In other words, no pair of selected reference values has an approximate string similarity above the similarity threshold, s_t, according to the used string similarity function $sim()$.

As outlined in Algorithm 4.1, in line 1, each DO that participates in a linkage protocol first initialises a set of reference values, **R**. The DO then loops over the global database, **G**, in line 2. For each global value g in **G**, the DO checks if the set **R** contains any reference values in line 3. If the set **R** is empty, the DO adds g into **R** in line 4. If the set **R** already contains reference values, the DO initialises a maximum similarity value, s_{max}, in lines 5 and 6.

Algorithm 4.1: ***Selecting Reference Values (Phase 1)***

Input:
- **G**: Publicly available global dataset
- s_t: Similarity threshold
- $sim()$: Approximate string similarity function

Output:
- **R**: Reference value set

```
1:  R ← {}                                          // Initialise the set of reference values
2:  for g ∈ G do:                                   // Loop over all values in the global database
3:    if R = {} do:                                 // Check if the reference set is empty
4:      R.add(g)                                    // Add the selected first global value to the reference set
5:    else:                                         // Check if the reference set R contains reference values
6:      s_max ← 0                                   // Initialise the maximum similarity value
7:      for r ∈ R do:                               // Loop over all reference values in R
8:        s ← sim(g, r)                             // Calculate similarity between value g and reference value r
9:        s_max ← max(s_max, s)                     // Get the maximum similarity
10:     if s_max ≤ s_t do:                          // Check maximum similarity is less than the threshold
11:       R.add(g)                                  // Add global value, g, to reference value set, R
12: return R
```

The DO then loops over each reference value r in **R** and uses the agreed similarity function $sim()$ to calculate the similarity value between r and g in **G** in lines 7 and 8. The DO keeps track of the maximum similarity, s_{max}, between g and any r in **R** in line 9. The DO repeats the steps in lines 7 to 9 until every r in **R** has been processed. If the resulting similarity s_{max} is lower than the agreed similarity threshold, s_t, the DO inserts g into **R** in lines 10 and 11, because in this case, g is different enough from all so far selected reference values. The DO repeats steps in lines 2 to 11 until every global value g in **G** has been processed.

4.3.2 Reference Values based BLIP BF Hardening

In the second phase of our approach, the DOs agree on the set of the hash functions (such as the hash function $SHA256$ [154] where the number of hash functions is set to optimal [122]), **H**, set of attributes (the attributes that the DOs aim to link between their databases), **A**, number of reference values per record, k (which can define the linkage quality as we describe in Section 4.4, using values such as $k = 1, 3$), length of q-gram, q ($q = 2, 3$ are commonly used [38]), length of BF, l ($l = 1,000$ is commonly used [38]), BLIP flip probability, f (we recommend $0.01 \leq f \leq 0.1$), and BLIP hardening approach, m, as either proposed by Alaggan et al. [5] or Schnell and Borgs [157]. According to our experimental result described in Section 4.5.2, we recommend using the BLIP approach proposed by Schnell and Borgs [157].

Based on the set of attributes **A**, each DO encodes attribute values of records in its database into BFs (attribute-level BFs as we described in Chapter 2), where each BF is with the length l. After that, the DO conducts the hardening of each BF by using the selected BLIP approach and the selected k reference values from Algorithm 4.1. The

Algorithm 4.2: ***Reference Value based BLIP Bloom Filter (RBBF) Hardening (Phase 2)***

Input:
- **D**: Database
- **R**: Reference value set
- **H**: Hash function set
- **A**: Attribute value set
- k: Number of reference values per BF
- q: Q-gram length
- l: BF Length
- f: BLIP flip probability
- m: BLIP approach (either *ala* [5] or *rap* [157])
- $sim()$: Approximate string similarity function

Output:
- **E**: Inverted index of RBBF encoded values of **D**

1: $\mathbf{E} \leftarrow \{\}$ // Initialise the inverted index of RBBF values
2: $\mathbf{p} \leftarrow genBitPosPermList(k \times l)$ // Generate a list of permuted bit positions
3: **for** $v \in \mathbf{D}$ **do:** // Loop over all plaintext values in a database
4: $\mathbf{q} \leftarrow genQgramSet(v, \mathbf{A}, q)$ // Generate q-gram set for v
5: $b_q \leftarrow genBF(\mathbf{q}, \mathbf{H}, l)$ // Generate BF for q-grams in the set $\mathbf{q}$
6: $\mathbf{r}_v \leftarrow getMostSimRefValSet(\mathbf{R}, \mathbf{A}, v, sim(), k)$ // Get the k most similar between $r \in \mathbf{R}$ and v
7: $b_v \leftarrow [\,]$ // Initialise a BF for v
8: **for** $r_v \in \mathbf{r}_v$ **do**: // Loop over reference values for v
9: $setRandomGeneratorSeed(r_v)$ // Initialise the PRNG
10: **if** $m = ala$ **do**: // Apply *ala* BLIP method, Equation 4.1
11: $b_r \leftarrow applyAlaBLIP(f, b_q)$
12: **else**: // Apply *rap* BLIP method, Equation 4.2
13: $b_r \leftarrow applyRapBLIP(f, b_q)$
14: $b_v \leftarrow b_v \| b_r$ // Concatenate b_r to final hardened BF for v
15: $b_v \leftarrow permuteBF(b_v, \mathbf{p})$ // Permute the final hardened BF for v
16: $\mathbf{E}[v] \leftarrow b_v$ // Add the final hardened BF to **E**
17: return **E**

DO then concatenates all hardened BFs into one new BF, resulting in a BF of length $k \times l$ for a given record. This BF is then permuted to ensure an adversary cannot identify the bit positions of an individual hardened BF generated using a certain reference value, where all hardened BFs are permuted in the same manner.

As outlined in Algorithm 4.2, in line 1, each DO initialises the inverted index for the BLIP hardened BFs, **E**. As all BFs in a database are being permuted in the same way, in line 2, the DO generates a list of permuted bit positions, **p**, by using the function $genBitPosPermList()$, where **p** contains all randomly permuted bit positions from 1 to $k \times l$ (the length of the final hardened BFs).

In line 3, for each plaintext value, v, of a record in a database, **D**, the DO uses the function $genQgramSet()$ to generate the set of q-grams, **q**, where **q** is generated based on the set of attributes, **A**, and the length of q-gram, q, in line 4. The DO then uses the function $genBF()$ to encode q-grams in **q** into a BF, b_q, of length l by using the set of hash functions, **H**, in line 5. The k most similar reference values to v based on the agreed set of attributes **A** and the function $sim()$ are identified from the set of reference values **R** (generated by Algorithm 4.1) by using the function $getMostSimRefValSet()$, resulting in the set $\mathbf{r}_v$ in line 6. The DO then initialises the BF, b_v, for v in line 7.

The DO loops over the selected reference value r_v in $\mathbf{r}_v$ in line 8 and uses it as the random seed for a pseudo-random number generator (PRNG) in the function

setRandomGeneratorSeed() in line 9. In lines 10 to 13, the DO generates the hardening, b_r, of the BF, b_q, by applying the selected BLIP approach, m, as either proposed by Alaggan et al. [5] (Equation 4.1) or Schnell and Borgs [157] (Equation 4.2), as we described in Section 4.2, where the flip bits are based on the flip probability, f.

The resulting BLIP hardened BF, b_r, is then concatenated at the end of the record BF, b_v, in line 14. The DO repeats the steps in lines 8 to 14 until every reference value r_v in $\mathbf{r}_v$ has been used as the random seed. In line 15, the DO permutes the concatenated BF, b_v, based on the generated list of permuted bit positions $\mathbf{p}$ by using the function *permuteBF*(). The permuted b_v is then inserted into the inverted index of RBBF encoded values, $\mathbf{E}$. The DO repeats the steps in lines 3 to 16 until the BLIP hardened BF of every record in a database $\mathbf{D}$ has been generated.

4.4 Analysis

We now analyse our proposed RBBF approach in terms of linkage quality, scalability, and privacy.

4.4.1 Linkage Quality Analysis

The two main parameters of our approach that will affect the final linkage quality (besides the quality of the input data and the general parameters used for BF encoding and BLIP hardening such as the flip probability, f) are the similarity threshold for selecting reference values, s_t, in Algorithm 4.1 and the number of reference values, k, in Algorithm 4.2.

If a lower s_t is used, then the set of reference values $\mathbf{R}$ will be smaller, and therefore it will be more likely that two dissimilar plaintext values will have the same reference value(s) in $\mathbf{R}$ which results in the same BLIP hardened BFs. This potentially increases false matches, resulting in lower linkage quality. A higher s_t leads $\mathbf{R}$ to contain more reference values, and therefore a higher likelihood that dissimilar plaintext values will have different reference values leading to different BLIP hardening of these dissimilar plaintext values. Therefore, a higher s_t results in higher linkage quality. The probability that records in a database have the same reference values can be calculated by $|\mathbf{D}|/|\mathbf{R}|$, where $|\mathbf{D}|$ is the size of the database and $|\mathbf{R}|$ is the size of the set of references.

When using more reference values, k, per plaintext value of a record, the linkage quality will likely increase because there is a higher chance that similar plaintext values share the same reference value(s), leading to similar BLIP hardening. On the other hand, when fewer reference values are used, it is more likely that dissimilar plaintext values share the same reference value(s) which can lower linkage quality.

4.4.2 Scalability Analysis

The computational complexity of Algorithm 4.1 depends upon the size of a global database, $\mathbf{G}$, as well as the similarity threshold, s_t. If s_t is set to a high value, then

more values in **G** are inserted into a set of reference values, **R**. In the worst-case scenario, if $s_t = 1.0$ (assuming the similarity function, $sim()$, returns a normalised value in between 0 and 1), then all values in **G** will be added into **R** leading to a complexity of Algorithm 4.1 of $O(|\mathbf{G}|^2)$.

In Algorithm 4.2, the DO iterates over all plaintext values of records in a database, **D**, leading to a complexity of $O(|\mathbf{D}|)$. Generating the q-gram set **q** and the BF b_q in lines 4 and 5 are of complexity $O(Q \times |\mathbf{H}|)$, where we assume Q is the average number of q-grams in a plaintext value $v \in \mathbf{D}$, and $|\mathbf{H}|$ is the number of hash functions used to encode q-grams into BFs. Finding the k most similar reference values to a given $v \in \mathbf{D}$ from **R** requires $|\mathbf{R}|$ similarity calculations, where $|\mathbf{R}|$ is the size of the set of reference values **R**. The BLIP generation in lines 8 to 14 has a complexity of $O(k \times l)$, where k is the number of selected reference values and l is the length of the BF b_q. Finally, the permutation of the concatenated BF b_v has a complexity of $O(k \times l)$ as a loop over all bit positions is required. Overall, the complexity of Algorithm 4.2 is $O(|\mathbf{D}| \times ((Q \times |\mathbf{H}|) + |\mathbf{R}| + (2 \times k \times l)))$.

4.4.3 Privacy Analysis

In the reference values selection process (Algorithm 4.1), the DOs that participate in the protocol select values from a global database, **G**, to generate a set of reference values, and therefore all the DOs know the set of reference values **R** that is being used in the protocol. However, in the BLIP hardening generation process (Algorithm 4.2), the DOs individually generate their BLIP hardening BFs without knowing the values of any other DOs, and thus no DO can learn any sensitive information from the other databases.

As our RBBF approach uses k reference values as a random seed for flipping bits in the BLIP hardening process, similar plaintext values will result in similar BLIP hardening. However, no sensitive information is being revealed when these BLIP hardened BFs are compared. This is because each reference value that is used for BLIP hardening can be used by multiple records (plaintext values) in a database **D**. Hence, there is no one-to-one relation between reference and plaintext values unless a reference value is used by only one record. Therefore, it is more difficult for an adversary to re-identify plaintext values of records in a database **D**. We illustrate the re-identification results using the frequency based cryptanalysis attack [40] in Section 4.5.4.

4.5 Experimental Evaluation

We discuss the datasets and experimental setup in Section 4.5.1. We then discuss the results of the experimental evaluation of our approach compared to BFs without hardening applied, and BLIP approaches proposed by Alaggan et al. [5] and Schnell and Borgs [157] in terms of linkage quality, scalability, and privacy in Sections 4.5.2 to 4.5.4, respectively.

4.5.1 Datasets and Experimental Setup

We evaluated our RBBF approach compared to the baselines by using North Carolina Voter Registration[1] (NCVR) real-world database, as we described in Section 2.6. We first extracted 1 million records from 7 million records of the snapshot from 2019 of the NCVR database and used them as the global database, **G**, to generate four sets of reference values for the attributes first name, last name, street address, and city. Using stratified sampling, we identified 100,000 record pairs from 7 million records of NCVR from a snapshot of 2019 for the same set of attributes, where we had 10,000 pairs in each of the ten similarity intervals $[0.0, 0.1)$, $[0.1, 0.2)$, ..., $[0.9, 1.0)$.

We set the similarity thresholds for selecting the reference values in Algorithm 4.1 to $s_t = [0.4, 0.8]$, and the numbers of reference values in Algorithm 4.2 to $k = [1, 3]$ to evaluate how dissimilar ($s_t = 0.4$) and similar ($s_t = 0.8$) of reference values in **R** and the number of reference values ($k = 1$ and $k = 3$) affect to the linkage quality of our approach. We converted attribute values (plaintext values) into q-grams using the length of q-gram $q = 2$ and encoded them into BFs of length $l = 1,000$ bits by using random hashing [129, 157] (as we described in Section 2.3.1) and optimal numbers of hash functions [122] for each attribute (dataset), which are 139 for first name, 116 for last name, 43 for street address, and 99 for city. We set the BLIP flip probabilities $f = [0.01, 0.1]$ to evaluate how the flip probability f influences the linkage quality.

We compared our approach (RBBF) with BFs without any hardening (No-BLIP) and the two BLIP approaches by Alaggan et al. [5] (named BLIP-A), and Schnell and Borgs [157] (named BLIP-S). We named our RBBF approach based on Equation 4.1 (proposed by Alaggan et al. [5]) and Equation 4.2 (proposed by Schnell and Borgs [157]) as RBBF-A and RBBF-S, respectively.

4.5.2 Linkage Quality Results

For the linkage quality evaluation, we used precision (Equation 2.13) and recall (Equation 2.14) to evaluate our approach and baselines. We first compared the Dice-coefficient calculated between q-gram sets [33], sim_D^q, (Equation 2.1) with the Dice-coefficient calculated between BFs [155], sim_D^b, (Equation 2.6). As we illustrate in Figure 4.2, for a pair of records, we assumed the q-gram Dice-coefficient sim_D^q to be the true similarity. For a given threshold, t, we classified the corresponding BF pair with its Dice-coefficient sim_D^b as:

- A BF pair is classified as true positive (TP) if $sim_D^q \geq t$ and $sim_D^b \geq t$.
- A BF pair is classified as false positive (FP) if $sim_D^q < t$ and $sim_D^b \geq t$.
- A BF pair is classified as true negative (TN) if $sim_D^q < t$ and $sim_D^b < t$.
- A BF pair is classified as false negative (FN) if $sim_D^q \geq t$ and $sim_D^b < t$.

[1] http://dl.ncsbe.gov/

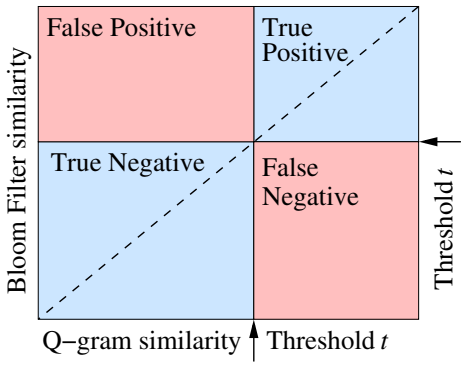

Figure 4.2: Classification outcomes of a Bloom filter pair based on q-gram and Bloom filter Dice-coefficient [175].

Table 4.2: Precision and recall for attribute first name with 139 hash functions used for Bloom filter encoding. The best results for each f and t setting are shown in bold.

BLIP method	Flip probability (f)	Number of references	References similarity	$t = 0.8$		$t = 0.9$	
				prec	*rec*	*prec*	*rec*
No-BLIP	-	-	-	0.62	1.0	0.39	1.0
BLIP-A	0.01	-	-	**0.67**	**0.99**	**0.51**	**1.0**
BLIP-S	0.01	-	-	0.64	1.0	0.43	1.0
RBBF-A	0.01	1	0.4	0.66	0.99	0.45	1.0
			0.8	0.67	0.99	0.44	1.0
		3	0.4	0.66	1.0	0.44	1.0
			0.8	0.66	1.0	0.47	1.0
RBBF-S	0.01	1	0.4	0.63	1.0	0.43	1.0
			0.8	0.64	1.0	0.41	1.0
		3	0.4	0.63	1.0	0.42	1.0
			0.8	0.64	1.0	0.43	1.0
BLIP-A	0.1	-	-	1.0	0.15	0.0	0.0
BLIP-S	0.1	-	-	0.87	0.58	1.0	0.02
RBBF-A	0.1	1	0.4	0.72	0.55	0.36	0.47
			0.8	0.84	0.43	0.28	0.36
		3	0.4	0.88	0.60	0.52	0.31
			0.8	0.97	0.43	0.52	0.29
RBBF-S	0.1	1	0.4	0.73	0.84	0.26	0.48
			0.8	0.81	0.77	0.24	0.37
		3	0.4	**0.81**	**0.93**	0.54	0.52
			0.8	0.84	0.84	**0.63**	**0.52**

Table 4.3: Precision and recall for attribute last name with 116 hash functions used for Bloom filter encoding. The best results for each f and t setting are shown in bold.

BLIP method	Flip probability (f)	Number of references	References similarity	$t = 0.8$		$t = 0.9$	
				prec	*rec*	*prec*	*rec*
No-BLIP	-	-	-	0.73	1.0	0.27	1.0
BLIP-A	0.01	-	-	**0.81**	**0.99**	**0.62**	**1.0**
BLIP-S	0.01	-	-	0.76	1.0	0.36	1.0
RBBF-A	0.01	1	0.4	0.78	1.0	0.35	1.0
			0.8	0.79	0.98	0.47	1.0
		3	0.4	0.79	1.0	0.41	1.0
			0.8	0.79	0.97	0.41	1.0
RBBF-S	0.01	1	0.4	0.75	1.0	0.28	1.0
			0.8	0.75	1.0	0.30	1.0
		3	0.4	0.75	1.0	0.31	1.0
			0.8	0.75	1.0	0.30	1.0
BLIP-A	0.1	-	-	0.99	0.03	0.0	0.0
BLIP-S	0.1	-	-	0.98	0.59	1.0	0.01
RBBF-A	0.1	1	0.4	0.82	0.47	0.23	0.47
			0.8	0.80	0.29	0.38	0.53
		3	0.4	0.97	0.54	0.53	0.33
			0.8	0.96	0.35	0.59	0.25
RBBF-S	0.1	1	0.4	0.83	0.77	0.20	0.47
			0.8	0.84	0.70	0.34	0.54
		3	0.4	**0.87**	**0.85**	0.54	0.56
			0.8	0.92	0.82	**0.65**	**0.53**

We then used these classified values to calculate precision by using Equation 2.13 and recall by using Equation 2.14. We illustrate the precision and recall results (calculated as described above) of our RBBF approach (RBBF-A and RBBF-S) compared to No-BLIP, BLIP-A, and BLIP-S on first name, last name, street address, and city attributes in Tables 4.2 to 4.5, respectively.

As shown in Tables 4.2 to 4.5 and as expected, in our approach, when the reference similarity threshold s_t increased, the linkage quality is slightly increased. This is because higher s_t means more reference values are inserted into the set of reference values **R** leading to BFs corresponding to similar records being hardened by using the same reference values while BFs corresponding to dissimilar records being hardened differently, resulting in higher precision values.

The higher number of reference values, k, per record results in higher precision and recall values. This is because it is a higher chance that similar plaintext values

Table 4.4: Precision and recall for attribute street address with 43 hash functions used for Bloom filter encoding. The best results for each f and t setting are shown in bold.

BLIP method	Flip probability (f)	Number of references	References similarity	$t = 0.8$		$t = 0.9$	
				prec	*rec*	*prec*	*rec*
No-BLIP	-	-	-	0.55	1.0	0.22	1.0
BLIP-A	0.01	-	-	**0.60**	**1.0**	**0.28**	**0.99**
BLIP-S	0.01	-	-	0.58	1.0	0.25	1.0
RBBF-A	0.01	1	0.4	0.59	1.0	0.25	1.0
			0.8	0.60	1.0	0.24	1.0
		3	0.4	0.59	1.0	0.25	1.0
			0.8	0.60	1.0	0.24	1.0
RBBF-S	0.01	1	0.4	0.56	1.0	0.23	1.0
			0.8	0.58	1.0	0.23	1.0
		3	0.4	0.57	1.0	0.23	1.0
			0.8	0.58	1.0	0.23	1.0
BLIP-A	0.1	-	-	0.99	0.35	0.0	0.0
BLIP-S	0.1	-	-	0.90	0.92	0.89	0.10
RBBF-A	0.1	1	0.4	0.69	0.72	0.29	0.80
			0.8	0.81	0.74	0.27	0.82
		3	0.4	0.82	0.80	0.42	0.73
			0.8	0.89	0.82	0.34	0.76
RBBF-S	0.1	1	0.4	0.67	0.95	0.27	0.84
			0.8	0.79	0.96	0.24	0.83
		3	0.4	0.70	0.98	**0.33**	**0.85**
			0.8	**0.79**	**0.98**	0.28	0.90

(similar records) use the same reference values for generating BLIP hardened BFs. As also shown in these tables, the recall values of 1.0 for BFs without hardening (No-BLIP) implies that the Dice-coefficient similarities of BFs are very close to the q-gram Dice-coefficient similarities. When the similarity threshold for classifying a match, t, is higher, all BLIP hardening techniques provide lower precision and recall values.

With higher flip probability ($f = 0.1$), BLIP-A leads to low precision and recall values and BLIP-S leads to low recall values, while our RBBF approach achieves results of higher precision and recall values. With lower flip probability ($f = 0.01$), BLIP-A seems to provide the highest precision and recall values. This is because it is possible that there are more 0-bits in the original BFs, resulting in more 1-bits in the BLIP-A hardened BFs, as we described in Section 4.2.

Overall, the results indicate that our approach based on the BLIP hardening approach proposed by Schnell and Borgs [157] from Equation 4.2 (RBBF-S) seems to

Table 4.5: Precision and recall for attribute city with 99 hash functions used for Bloom filter encoding. The best results for each f and t setting are shown in bold.

BLIP method	Flip probability (f)	Number of references	References similarity	$t = 0.8$		$t = 0.9$	
				prec	*rec*	*prec*	*rec*
No-BLIP	-	-	-	0.47	1.0	0.66	1.0
BLIP-A	0.01	-	-	0.48	1.0	0.66	1.0
BLIP-S	0.01	-	-	0.48	1.0	0.66	1.0
RBBF-A	0.01	1	0.4	**0.49**	**1.0**	**0.67**	**1.0**
			0.8	0.48	1.0	**0.67**	**1.0**
		3	0.4	0.48	1.0	0.66	1.0
			0.8	0.48	1.0	0.66	1.0
RBBF-S	0.01	1	0.4	0.47	1.0	0.66	1.0
			0.8	0.47	1.0	0.66	1.0
		3	0.4	0.47	1.0	0.66	1.0
			0.8	0.47	1.0	0.66	1.0
BLIP-A	0.1	-	-	1.0	0.31	0.0	0.0
BLIP-S	0.1	-	-	**0.76**	**0.98**	1.0	0.03
RBBF-A	0.1	1	0.4	0.76	0.95	0.72	1.0
			0.8	0.64	0.96	**0.95**	**1.0**
		3	0.4	0.77	0.95	0.72	1.0
			0.8	0.76	0.97	**0.95**	**1.0**
RBBF-S	0.1	1	0.4	0.63	1.0	0.70	1.0
			0.8	0.51	0.99	0.90	1.0
		3	0.4	0.57	1.0	0.66	1.0
			0.8	0.55	1.0	0.89	1.0

outperform our approach based on the BLIP hardening proposed by Alaggan et al. [5] from Equation 4.1 (RBBF-A). Our proposed RBBF approach outperforms both BLIP-A and BLIP-S approaches with regard to linkage quality for higher flip probability.

4.5.3 Scalability Results

In Table 4.6, we show the numbers of reference values that were generated by Algorithm 4.1 for different attributes (datasets) by using different reference similarity threshold values, s_t. As can be seen, a higher value of s_t results in more reference values are inserted into the set of reference values **R**. Furthermore, attributes with more unique values end up with more reference values leading to better linkage quality of our RBBF approach, as we discussed in Sections 4.4.1 and 4.5.2.

Table 4.6: Numbers of reference values generated for different attributes (datasets) using different values for the reference similarity threshold, s_t.

Attribute	Unique $g \in \mathbf{G}$	$s_t = 0.4$	$s_t = 0.8$
First name	50,321	1,888	29,514
Last name	91,205	3,813	63,845
Street address	479,379	3,839	90,540
City	223	123	221

Table 4.7: Average runtimes (in seconds) for BLIP (BLIP-A and BLIP-S) and RBBF (RBBF-A and RBBF-S, both with $s_t = 0.8$).

Attributes	$f = 0.01$			$f = 0.1$		
	BLIP	$k = 1$	$k = 3$	BLIP	$k = 1$	$k = 3$
First name	0.38	3.47	10.64	0.27	2.38	7.31
Last name	0.43	3.94	12.04	0.44	3.90	11.99
Street address	1.41	12.74	38.93	1.49	13.55	41.56
City	0.03	0.24	0.74	0.03	0.25	0.77

Table 4.7 shows the average runtimes for original BLIP (averaged between BLIP-A and BLIP-S) and our RBBF (averaged between RBBF-A and RBBF-S). As can be seen and as expected, the computational requirements of our RBBF approach increase when more reference values, k, are used, as also discussed in Section 4.4.2. As a result, our RBBF approach has longer runtimes than the original BLIP for all experiments. However, overall the flip probability, f, does not seem to affect runtimes.

4.5.4 Privacy Results

We show the re-identification accuracy of the cryptanalysis attack [40] on our approach, No-BLIP, BLIP-A, and BLIP-S in Figures 4.3 to 4.6. We assessed the re-identification accuracy of this attack by calculating the percentages of (1) correctly re-identified plaintext values encoded into BFs as one-to-one matches (one BF is correctly assigned to the plaintext value that was encoded into it); (2) correct guesses with one-to-many matches (a BF is assigned to several plaintext values and one of these values was the correct one encoded in the BF); (3) wrong guesses (a BF did not encode any of the plaintext values assigned to it); and (4) no guesses (the attack was not able to assign any plaintext values to a BF). We considered the accuracy of the attack on the 10, 20, 50, and 100 most frequent plaintext values from each dataset, respectively.

In Figures 4.3 to 4.6, the first column shows the results of flip probability $f = 0.01$ and the second column shows the results of $f = 0.1$. Figures 4.3 and 4.5 show the results of the number of reference values $k = 1$, while Figures 4.4 and 4.6 show the

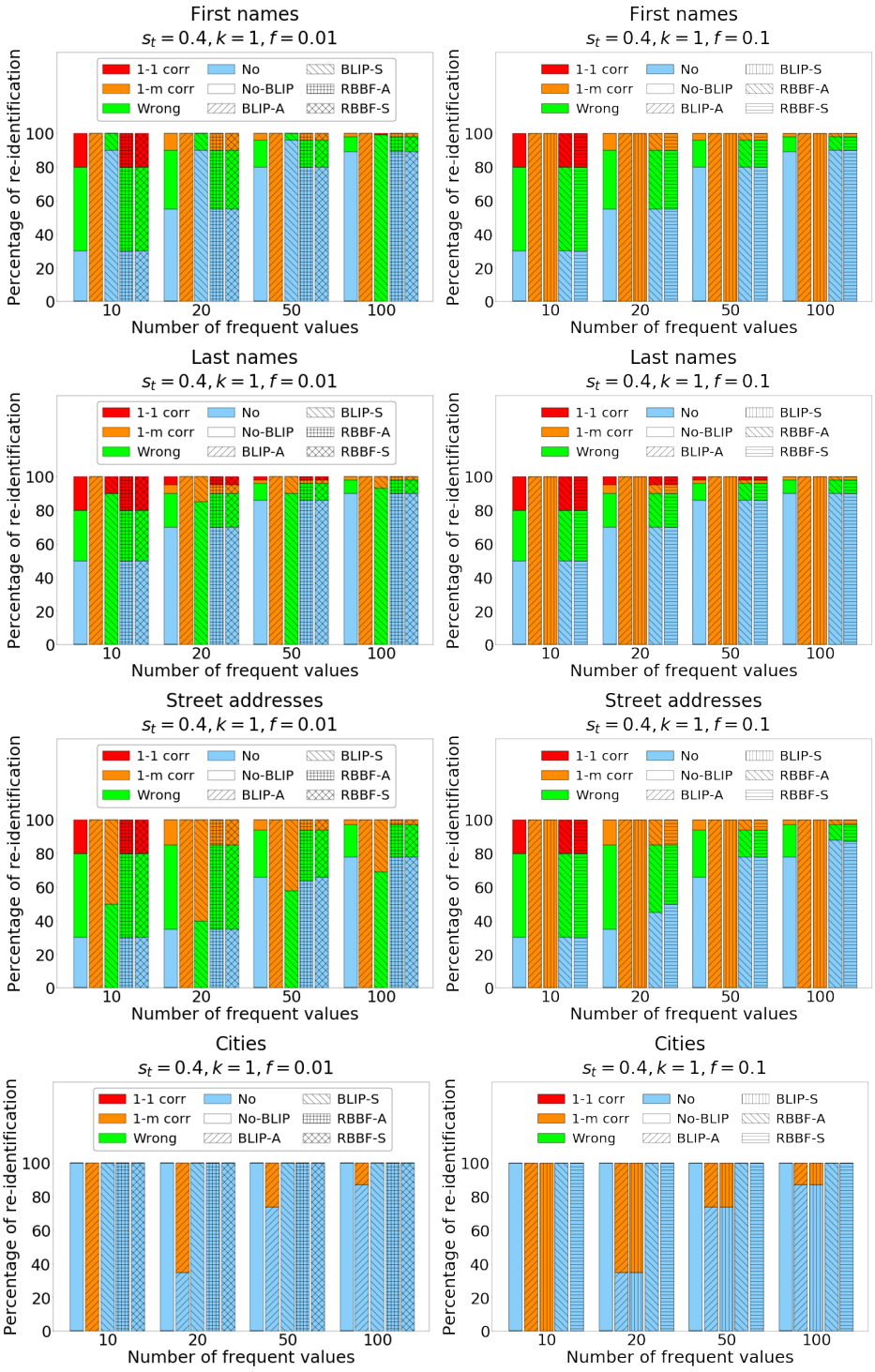

Figure 4.3: Re-identification results for our RBBF approach with reference similarity threshold $s_t = 0.4$ and number of reference values $k = 1$ compared to No-BLIP, BLIP-A, and BLIP-S approaches.

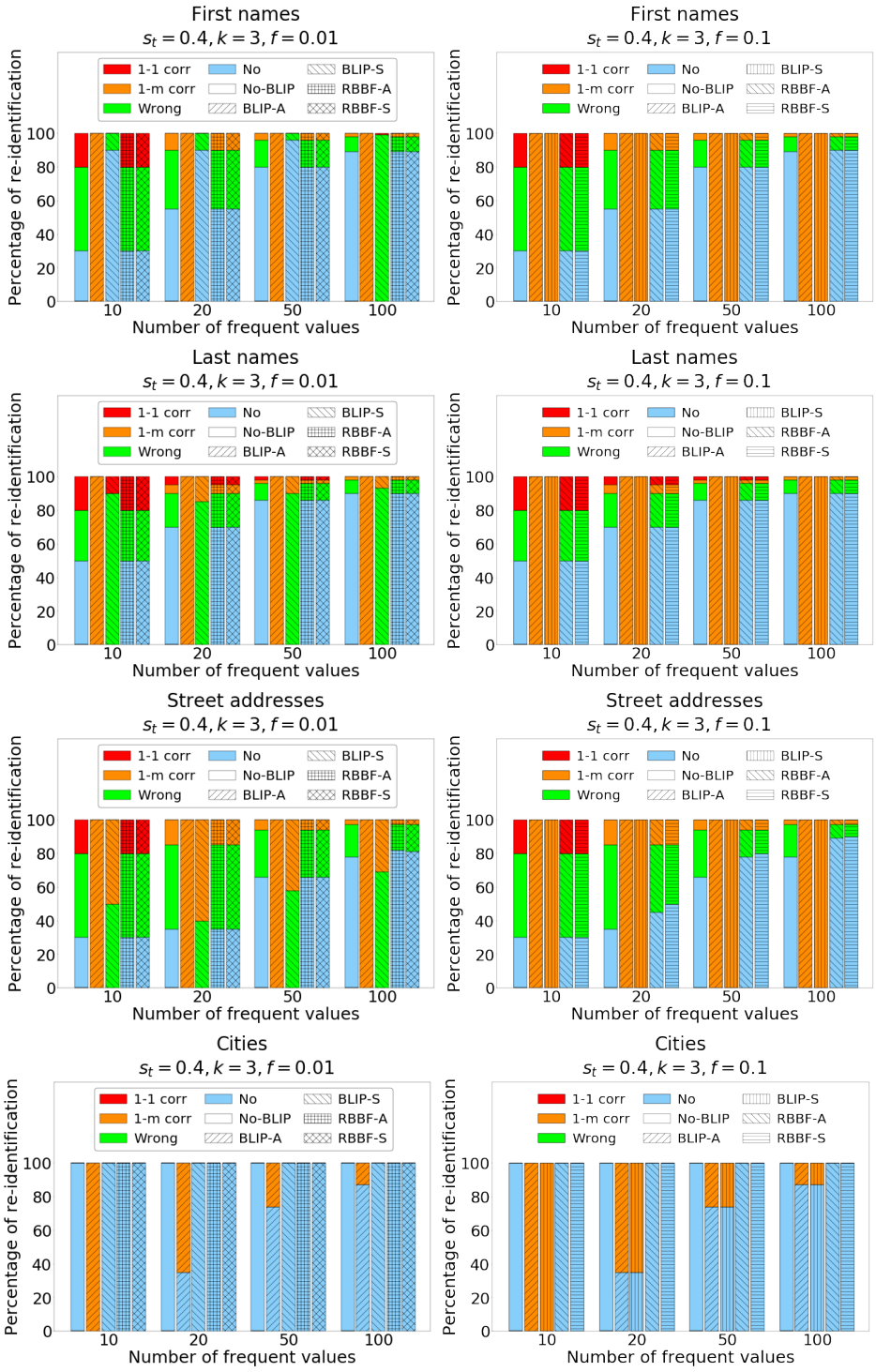

Figure 4.4: Re-identification results for our RBBF approach with reference similarity threshold $s_t = 0.4$ and number of reference values $k = 3$ compared to No-BLIP, BLIP-A, and BLIP-S approaches.

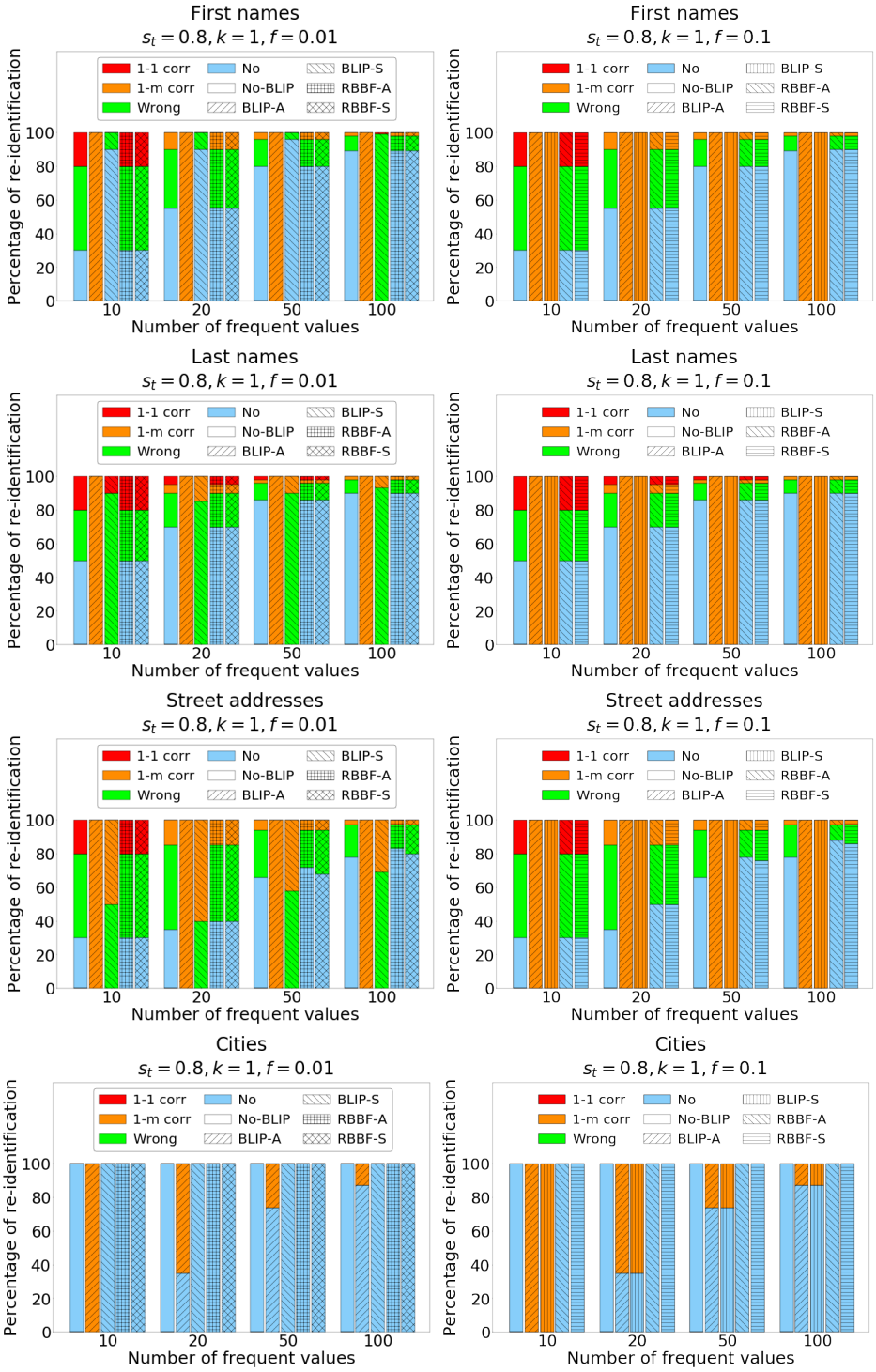

Figure 4.5: Re-identification results for our RBBF approach with reference similarity threshold $s_t = 0.8$ and number of reference values $k = 1$ compared to No-BLIP, BLIP-A, and BLIP-S approaches.

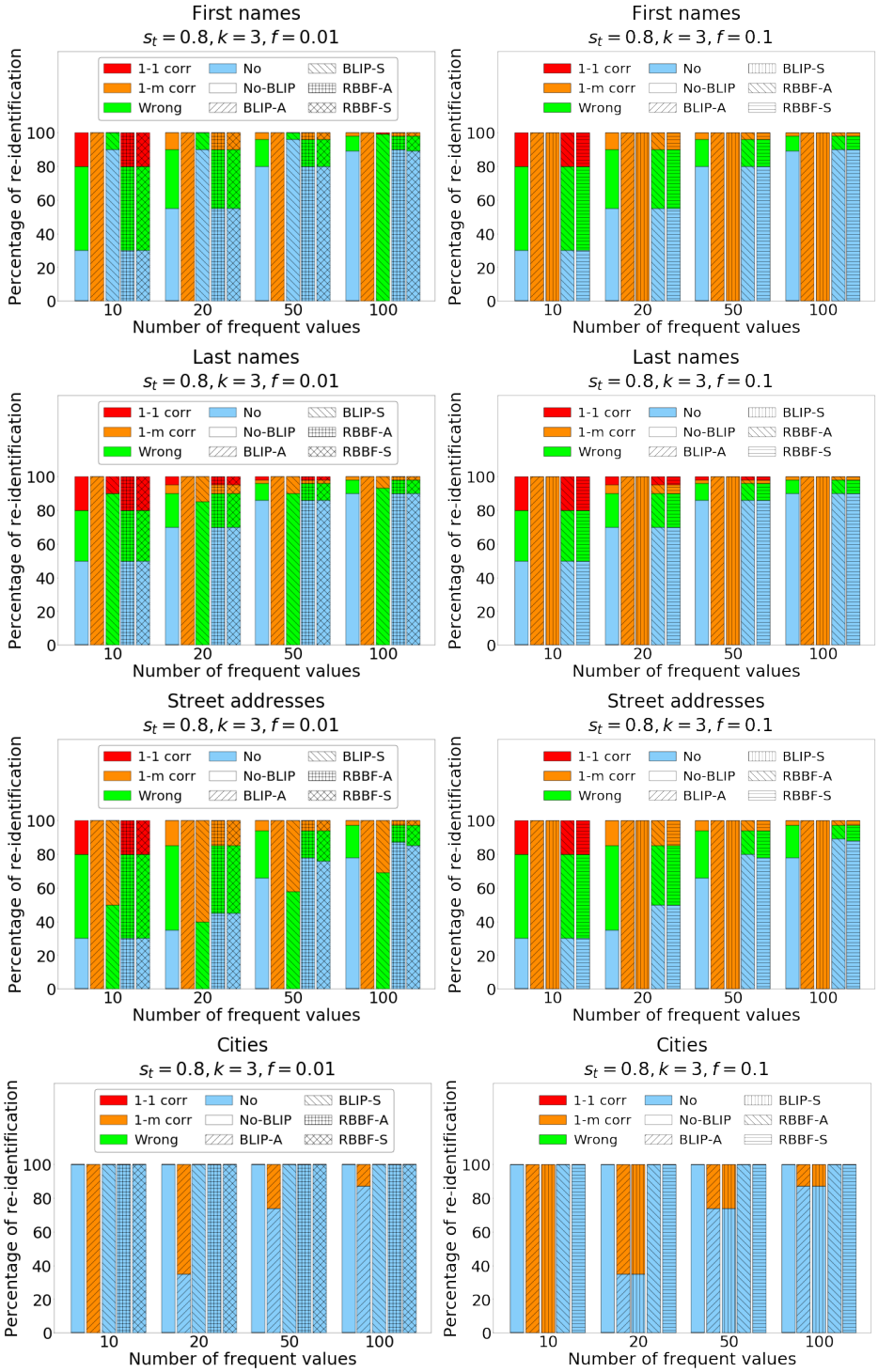

Figure 4.6: Re-identification results for our RBBF approach with reference similarity threshold $s_t = 0.8$ and number of reference values $k = 3$ compared to No-BLIP, BLIP-A, and BLIP-S approaches.

results of $k = 3$. The results of reference similarity threshold $s_t = 0.4$ are shown in Figures 4.3 and 4.4, and results of $s_t = 0.8$ are shown in Figures 4.5 and 4.6. The bars show the percentages of BFs and hardened BFs for which plaintext values were re-identified either correctly with one-to-one (1-1 corr), one-to-many (1-m corr), wrong, or no attribute values were re-identified.

As can be seen from Figures 4.3 to 4.6, our RBBF approach slightly improves the privacy compared to no hardened BF encoding (No-BLIP), especially for the street address attribute (dataset). This is because the street address is a long string, resulting in more q-grams to be encoded for each plaintext value which leads to more 1-bits in a BF. As a result, there is a higher possibility that more 1-bits are being flipped.

Our approach substantially improves privacy compared to BLIP-A and BLIP-S because the attacks on our RBBF approach mostly result in no and wrong guesses. It is interesting that the attack is still able to correctly re-identify of most datasets in a one-to-many manner in the original BLIPs, especially BLIP-A. This indicates that the original BLIP might not be as secure as hoped, and further research is required to investigate these results.

4.6 Chapter Summary

In this chapter, we have presented our Reference Values based BLIP BF Hardening (RBBF) approach to improve the BLIP hardening techniques [5, 157] for BF encoding for PPRL, as discussed in Chapter 2. Our approach selects reference values from a large publicly available database and uses these values to modify the original BLIP approaches (BLIP-A [5] and BLIP-S [157] described in Sections 2.3 and 4.2), such that similar plaintext values are randomised in the same way, as described in Section 4.3. We analysed the linkage quality, scalability, and privacy of our RBBF approach in Section 4.4. Our results on the NCVR real-world database show that our approach outperforms the original BLIP approaches [5, 157] while ensuring the hardened BFs are secure with regard to the frequency based cryptanalysis attack [40], as described in Section 4.5. However, our approach requires higher computational complexity than the original BLIPs, as we discussed in Section 4.5.3. In the following chapter, we present our PPRL technique for linking databases that contain missing values.

Chapter 5

Accurate Privacy-Preserving Record Linkage for Databases with Missing Values

In the previous chapter, we proposed a hardening technique for Bloom filter encoding [155] to increase the degrees of privacy and improve the linkage quality of the BLoom and flIP (BLIP) hardening technique [5, 157]. However, we did not consider missing values that might occur in the databases to be linked. In this chapter, we discuss our novel proposed privacy-preserving record linkage (PPRL) technique for linking databases that contain missing values based on a lattice structure.

In Section 5.1, we provide an introduction to our proposed technique. We then describe different types of missing data in Section 5.2. In Section 5.3, we explain a lattice structure that we use to represent missingness patterns of records in a database, and in Section 5.4 we give an overview of our proposed PPRL protocol. In Section 5.5, we describe the encoding and generating missing pattern processes, and in Section 5.6 we then explain two linkage methods (an iterative and a batch methods) and describe the record pairs comparison process. In Section 5.7, we analyse our approach in terms of linkage quality, scalability, and privacy, and in Section 5.8 we discuss the experimental evaluation. Finally, in Section 5.9, we summarise our proposed approach.

5.1 Introduction

As we discussed in Chapters 1 and 3, one major data quality aspect that so far has only seen limited attention in record linkage and privacy-preserving record linkage (PPRL) is missing values [6, 30, 68, 76, 130]. Missing values can occur for a variety of reasons and they can have different characteristics, as we discuss in Section 5.2. If records have missing values in the attributes used for linkage, then the similarities calculated between records will likely be lower, potentially affecting the final linkage quality. If alternatively those records or attributes that have missing values are not used for a linkage, then linkage quality will again likely suffer [76].

Table 5.1: Notation and terminology used in this chapter.

$\mathbf{D}, \mathbf{D}_A, \mathbf{D}_B$	Database, database A, and database B to be encoded, respectively
r	Record in a database $\mathbf{D}$
$\mathbf{L}_L, \mathbf{L}_A, \mathbf{L}_B$	Local lattice of a database $\mathbf{D}$, $\mathbf{D}_A$, and $\mathbf{D}_B$, respectively
$\mathbf{L}_C$	Lattice of common patterns between $\mathbf{D}_A$ and $\mathbf{D}_B$
m_C	A missingness pattern that occurs in $\mathbf{L}_C$
m_N	Neighbouring missingness patterns of m_C
$\mathbf{a}_L, \mathbf{a}_R$	List of linkage attributes and list of required attributes, respectively
n	Maximum number of missing attributes in $\mathbf{a}_L$
$\mathbf{w}$	List of attribute weights
$\mathbf{H}$	List of hash functions
g	Grouping method
d_t	Threshold of difference between Hamming weights
s_t	Similarity threshold
w_t	Weight threshold
q	Length of q-gram
l	Length of Bloom filter (BF)
$\mathbf{A}$	Inverted index of ABFs
$\mathbf{M}$	Missing pattern table
$\mathbf{P}$	Inverted index of partitions (grouped patterns)
$\mathbf{R}$	Inverted index of matched record pairs
$\|...\|$	Size or number of values in a database or list
MCAR	Data with missing completely at random
MAR	Data with missing at random
MNAR	Data with missing not at random
SSI	Secure set intersection protocol

As we discussed in Section 3.6, Chi et al. [30] investigated how to overcome the challenge of missing values in the PPRL process. The basic idea of their approach is to find the k most similar records to a record that has a missing value, and to use the similarities calculated between the available attribute values to estimate the similarity that a missing value would have had with the value in another record in a record pair. However, this approach uses an attribute-level Bloom filter (ABF) [155] which makes it vulnerable to cryptanalysis attacks [39, 109, 129], as we described in Section 2.3.3.

In this chapter, we present our proposed PPRL technique for linking databases that contain missing values in the attributes being linked. We first generate partitions of records based on their missingness patterns. We then use record-level Bloom filters (RBFs) [59] (as we described in Section 2.3.2) to encode values and improve the degrees of privacy of our approach. However, we employ different attribute weightings from the original RBF approach [59] to groups of records that are in different partitions, where we distribute the weights assigned to the attributes with missing values to non-missing attributes [130], and then select certain numbers of

bits based on the updated weights. We finally apply different permutations to BFs with different missingness patterns to prevent frequency based attacks [39, 108, 109].

To link partitions in a privacy-preserving way, we propose two methods, which are (1) an iterative method that requires multiple communication steps but needs less record pairs to be compared, and (2) a batch method that only requires one communication step but leads to a larger number of record pair comparisons. We provide the notation and terminology that we use in this chapter in Table 5.1.

5.2 Missing Data

Missing values are a common issue in many real-world databases. There are various reasons why missing values can occur, ranging from equipment malfunction or data items not considered to be important, to the deletion of values due to inconsistencies between attribute values of different records, or even the refusal of individuals to provide information [38]. Missing data can be categorised into different types based on their characteristics [117].

- **Missing completely at random**
  Data missing completely at random (MCAR) are missing values that occur without any patterns or correlations at all with any other values in the same record or database, and therefore they cannot be predicted by using any other data in the same database.

- **Missing at random**
  Data missing at random (MAR) are missing values that can be predicted by other data in the same record and/or database. As an example, for a record with a missing last name in a household database, the last name could be predicted by imputing from the most common last name of people who live in the same household.

- **Missing not at random**
  Data missing not at random (MNAR) are missing values that occur for some specific reasons, for example, a student lost a book that she rented from a library, therefore, a return date of this book in a library's database will contain a missing value.

- **Structurally missing**
  Structurally missing data are values that are missing because they should not exist and missing is their correct value. For example, a student who withdrew from a course should not have a final score for this course.

Any of these types of missing data can occur in the attributes used to link databases. While data imputation can be applied with the aim to fill such missing values before the databases are linked [87, 117] (as we discussed in Section 3.6), in

(a)

Record identifiers	First name	Last name	City	Zip code
r_1	marry		wilson	27896
r_2	mary	miller		27896
r_3	alan	hazel		27253
r_4	william	robert	lumberton	
r_5	julia	johnson	newland	28657
r_6	albert	bruce		
r_7	newton	kevin	henderson	27536
r_8	michael		jones	28752
r_9	maria		clayton	27520

(b)

First name	Last name	City	Zip code
1	0	1	1
1	1	0	1
1	1	0	1
1	1	1	0
1	1	1	1
1	1	0	0
1	1	1	1
1	0	1	1
1	0	1	1

(c)

$\mathbf{M}[1111] = \{r_5, r_7\}$
$\mathbf{M}[1110] = \{r_4\}$
$\mathbf{M}[1101] = \{r_2, r_3\}$
$\mathbf{M}[1011] = \{r_1, r_8, r_9\}$
$\mathbf{M}[1100] = \{r_6\}$

(d)

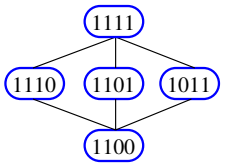

Figure 5.1: An example database where some records have missing values are shown in (a) and their corresponding missingness patterns are shown in (b). The missing pattern table, **M**, is shown in (c), and the resulting lattice structure is shown in (d).

our work, we assume that not all missing values can be imputed. Specifically, the imputation is not possible for missing values of types MCAR and structurally missing. For example, if a first name is missing for an individual, then neither middle name, last name, nor address can help predict the missing first name value (assuming no external database with a complete record of that person is available).

5.3 Lattice Structure to Represent Missingness Patterns

A lattice structure is an abstract structure used in various application areas, including database management, data mining, data warehousing, and information retrieval, to represent multidimensional data [82]. A lattice structure consists of elements of a set where there exists a partial order such that two elements of the set have a unique supremum and a unique infimum, as we illustrate in Figure 5.1 (d).

In our approach, we build a lattice structure from the missingness patterns of the attributes in a database that are used in a linkage process. Each such pattern is represented as a bit vector that becomes a node in the lattice structure, where a 0 represents a missing value while a 1 represents a non-missing value in a certain attribute. For example, the bit vector 1111 in Figure 5.1 (b) represents all records in a database that have no missing values in four attributes. These missingness bit patterns are arranged into a lattice structure where all bit patterns with a certain Hamming weight (number of 1-bits) form one layer of the lattice, and an edge connects two nodes if their Hamming weights differ up to a threshold $d_t \geq 1$ (they differ

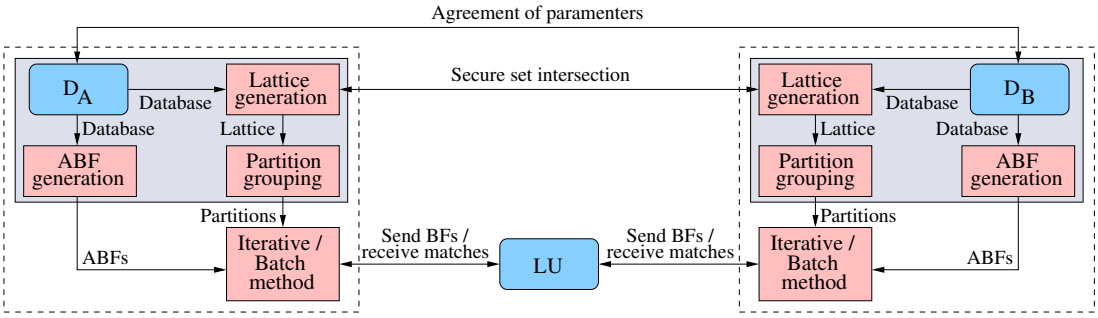

Figure 5.2: Overview of our proposed PPRL protocol for linking databases with missing values. The two databases, $\mathbf{D}_A$ and $\mathbf{D}_B$, and the linkage unit (LU) are shown in blue, while red boxes show the main steps of our approach. The shaded areas are the common steps to generate partitions and encode values.

by up to d_t 1-bits). The lattice structure allows us to identify common missingness patterns that occur in the databases to be linked, where these patterns are the basis of how we generate partitions of records.

5.4 Protocol Overview

Our proposed PPRL approach for missing data involves three parties, the two database owners (DOs) and a linkage unit (LU), where we assume our approach follows the honest-but-curious (HBC) adversary model [74, 81], as we described in Section 2.2.2. Each of the two DOs has a database $\mathbf{D}_A$ and $\mathbf{D}_B$, respectively, which they aim to link without having to share the sensitive attribute values in their databases with any other party. The LU is used to conduct the linkage where it only identifies the matching record pairs between the two databases. As illustrated in Figure 5.2, the DOs first agree on the parameters to be used in the protocol, which are:

- The list of linkage attributes, $\mathbf{a}_L$, list of hash functions (such as the hash function $SHA256$ [154] where the number of hash functions is set to optimal [122]), $\mathbf{H}$, length of q-gram, q (commonly used are $q = 2, 3$ [38]), and length of Bloom filter (BF), l (commonly used is $l = 1,000$ [38]), to be used for generating BF, as we describe in Section 5.5.1.
- The n number (maximum number of missing attributes in $\mathbf{a}_L$, where $n \leq \mathbf{a}_L - 1$) and the list of required attributes, $\mathbf{a}_R$, to be used for controlling the linkage quality, as we describe in Section 5.5.2. These parameters require knowledge of the DOs to decide how many and which attributes can be missing.
- The list of attribute weights, $\mathbf{w}$ (the higher importance attribute must be set to higher weight, for example, the most important attribute in the list is given the weight 0.9), the weight threshold, w_t (such as $w_t = 0.7$), the threshold of difference between Hamming weights, d_t (such as $d_t = 2$), and the grouping

method, g, to be used for grouping partitions, as we describe in Section 5.5.4. The parameters w_t, d_t, and g can be decided based on the missing values in the databases to be linked, where the higher w_t and d_t increase the chances of matched record pairs as we describe in Section 5.7.

- The order of partitions being sent to the LU from the DOs, as we describe in Section 5.6.1, and the similarity threshold, s_t, to be used for classifying a pair of records as a match or non-match where the higher s_t (such as $s_t = 0.8$) means records in a pair are highly similar, as we describe in Section 5.6.3.

The DOs individually generate a set of attribute-level BFs (ABFs) [155] (as we described in Sections 2.3.2 and 5.5.1) for each record in their database, where one ABF is generated for each non-missing attribute in a record that is used for the linkage.

Next, the DOs individually generate local lattice structures, $\mathbf{L}_A$ and $\mathbf{L}_B$, where each represents the unique missingness patterns that occur in the attributes to be linked in their databases $\mathbf{D}_A$ and $\mathbf{D}_B$. Each such pattern will represent records in a database depending upon how missing values occur in records, as we describe in detail in Section 5.5.2. The DOs then participate in a secure set intersection (SSI) protocol [54] (as we described in Section 2.2.3) to identify the common missingness patterns between their databases, resulting in the lattice structure of common patterns, $\mathbf{L}_C$, as we describe in Section 5.5.3. The SSI will hide information about the not common patterns of a DO from the other DO, and thereby prevent the DOs from being able to potentially re-identify each other's sensitive attribute values, as we discuss in Section 5.7.3.

Each DO now generates partitions that consist of records that have one or more missingness patterns, as we describe in Section 5.5.4. For each common missingness pattern $m_C \in \mathbf{L}_C$, neighbouring missingness patterns, m_N, are grouped into one or more partitions based on the local and common lattice structures with grouping either upwards, downwards, or in both directions. Based on these groupings, each record will be added to one or multiple partitions. For each of these partitions, based on the missingness pattern of that partition, different bits will be selected from the ABFs of the records in that partition and concatenated into one BF per record. Different permutations of bit positions are then applied on these record BFs to improve privacy, as we describe in Section 5.6.1. Figure 5.3 illustrates the BFs of two records with different missing patterns that are generated by using our approach.

To compare partitions of BFs between DOs, we propose two linkage methods, iterative and batch methods, to identify matching record pairs as we describe in detail in Section 5.6. In the iterative linkage method, each DO sends a partition consisting of a set of BFs to the LU, which can then calculate the Dice-coefficient [33] similarity (Equation 2.6) between pairs of BFs. The LU returns the classified matched record pairs to the DOs, which are then removed from the next partition to prevent redundant comparisons. The DOs send the next partition to the LU, and the process is repeated until all partitions have been compared.

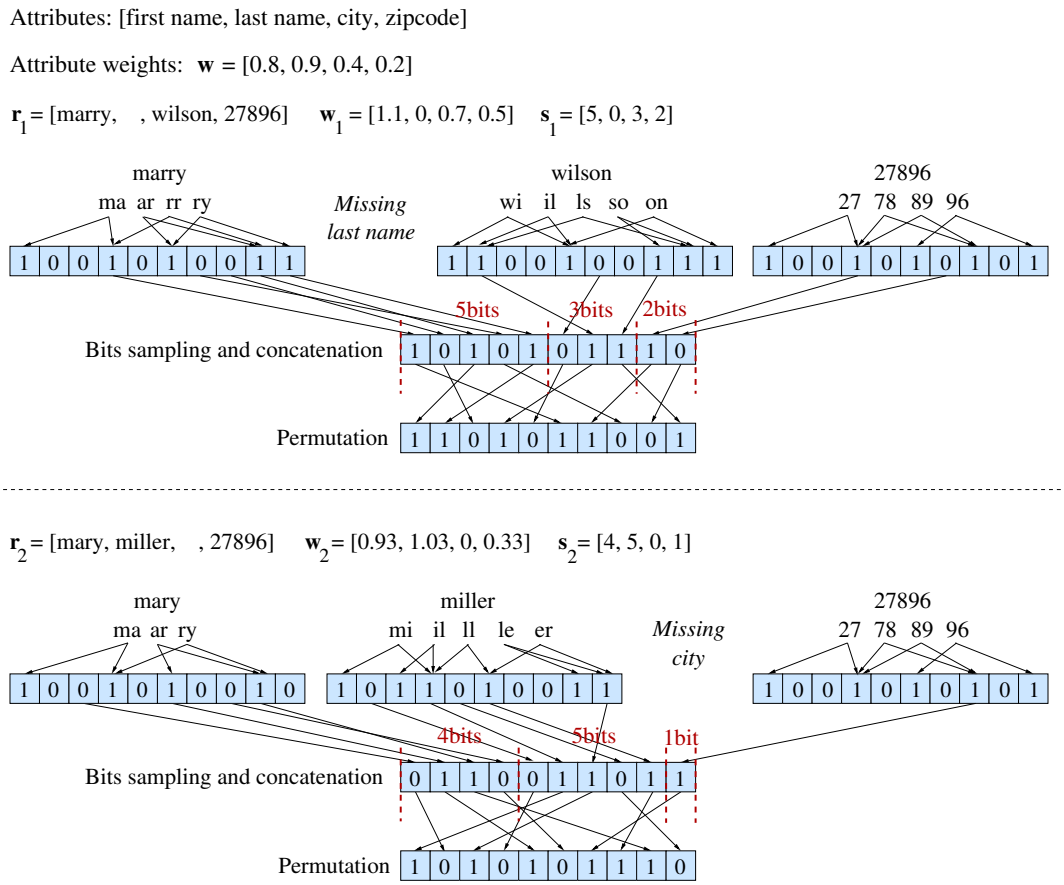

Figure 5.3: Record BF generation of our approach for records r_1 and r_2 from Figure 5.1 that have different missingness patterns (r_1 has a missing last name while r_2 has city missing). We describe this process in detail in Section 5.5.

One drawback of the iterative linkage method is that it requires multiple rounds of communication between the DOs and the LU. This might not be possible in certain application scenarios (especially when multiple databases need to be linked [180]). We describe the worst time complexity of the iterative linkage in Section 5.7.2. In the batch linkage method, all partitions are combined into one encoded database by each DO, and then sent to the LU to conduct a comparison process, as we describe in detail in Section 5.6.2.

5.5 Encoding and Generating Missing Pattern

Based on the parameter settings that have been agreed upon by the DOs, as we described in Section 5.4, to encode records and process missingness patterns of records, each DO first initialises an inverted index of ABFs, **A**, a missing pattern table, **M**, an inverted index of partitions, **P**, and a local lattice structure, $\mathbf{L}_L$, in lines 1 and 2 of Algorithm 5.1.

Algorithm 5.1: ***Common Steps Performed by a Database Owner***

Input:
- **D** : Database to be encoded
- $\mathbf{a}_L$: List of linkage attributes
- $\mathbf{a}_R$: List of required attributes
- **w** : List of attribute weights
- **H** : List of hash functions
- g : Grouping method
- n : Maximum number of missing attributes in $\mathbf{a}_L$
- w_t : Weight threshold
- l : ABF length
- d_t : Threshold of Hamming weights different
- q : Q-gram length

Output:
- **A** : Inverted index of ABFs
- **M** : Missing pattern table
- **P** : Inverted index of partitions (grouped patterns)

1: $\mathbf{A} \leftarrow \{\}, \mathbf{P} \leftarrow \{\}, \mathbf{M} \leftarrow \{\}$ // Initialise data structures
2: $\mathbf{L}_L \leftarrow \{\}$ // Initialise local lattice structure
3: **for** $r \in \mathbf{D}$ **do**: // Loop over each record in the database
4: $\mathbf{b}_r \leftarrow [\,]$ // Initialise list for ABFs of the current record r
5: **for** $a \in \mathbf{a}_L$ **do**: // Loop over linkage attribute list
6: $b_a \leftarrow genABF(r.a, q, \mathbf{H}, l)$ // Generate the ABF for an attribute a
7: $\mathbf{b}_r.append(b_a)$ // Add ABF to list
8: $\mathbf{A}[r.id] \leftarrow \mathbf{b}_r$ // Add ABF list of record r to inverted index
9: $m_r \leftarrow genMissPattern(r, \mathbf{a}_L, \mathbf{a}_R)$ // Generate missing value pattern for r
10: **if** $(|\mathbf{a}_L| - HW(m_r)) \leq n$ **do**: // Check number of non-missing values
11: $\mathbf{M}[m_r].add(r.id)$ // Add identifier of r to pattern m_r
12: $\mathbf{L}_L \leftarrow genLattice(\mathbf{M})$ // Generate the local lattice structure for the database
13: $\mathbf{L}_C \leftarrow secSetIntersect(\mathbf{L}_L)$ // Find common patterns across databases
14: $\mathbf{P} \leftarrow groupPattern(\mathbf{L}_L, \mathbf{L}_C, \mathbf{w}, g, d_t, w_t)$ // Group patterns into partitions
15: return **A**, **M**, **P**

5.5.1 Attribute-level BF Generation

For each record r in a database **D**, a DO first initialises a list of attribute-level BFs (ABFs), $\mathbf{b}_r$, in lines 3 and 4 of Algorithm 5.1. The DO then loops over the list of linkage attributes, $\mathbf{a}_L$, to generate an ABF of each attribute value $r.a$ to be used in the linkage process by using the function $genABF()$ in lines 5 and 6. An ABF of length l bits is generated by converting $r.a$ into a set of q-grams, where each q-gram is with the length q. The set of q-grams is then encoded into an ABF, b_a, by using the list of hash functions, **H**, that have been agreed upon by the DOs. If the attribute value $r.a$ is missing, a BF with only 0-bits is returned by $genABF()$. In line 7, each ABF is inserted into the list of ABFs for record r, $\mathbf{b}_r$. This list is then inserted into the inverted index, **A**, with the record's identifier, $r.id$, in line 8. The inverted index **A** will be used in later steps of the protocol when partitions are generated.

5.5.2 Missing Pattern Table Generation

For each record $r \in \mathbf{D}$, in line 9 of Algorithm 5.1, a DO generates its missingness pattern, m_r, based on the list of attributes that are used for the linkage, $\mathbf{a}_L$. The DO defines a subset of required attributes $\mathbf{a}_R \subset \mathbf{a}_L$, where the attributes in $\mathbf{a}_R$ are those that cannot be missing because they contain crucial information that is needed to link

records, such as the first and last names of individuals. If values in these attributes would be missing then it will not be possible to accurately link records.

For each attribute in $\mathbf{a}_L$, the function $genMissPattern()$ generates a 0-bit if an attribute value is missing in record r, or a 1-bit if a value exists in r. The function returns a missingness pattern which is a bit vector m_r of length $|\mathbf{a}_L|$. If any of the required attributes in $\mathbf{a}_R$ is missing, then the function $genMissPattern()$ will return a missingness pattern m_r consisting of only 0-bits, which means a record will not be considered further in the linkage process. If more than n attribute values are missing (the number of 1-bits, calculated using the function $HW()$ which returns the Hamming weight of a bit vector (missingness pattern m_r), is too low), a record will not be considered either. The parameter n and the list $\mathbf{a}_R$ are used to control the linkage quality for those records that contain many missing values.

In line 11, the DO inserts the missingness pattern m_r into the missing pattern table $\mathbf{M}$ (implemented as an inverted index) as a key and inserts a set of all record identifiers, $r.id$, of the records with the pattern m_r as a value under the key m_r. We illustrate an example of $\mathbf{M}$ in Figure 5.1 (c).

5.5.3 Lattice Structure and Common Pattern Generations

Each DO now uses its missing pattern table, $\mathbf{M}$, to create a local lattice structure, $\mathbf{L}_L$, that represents the patterns of missing values in its database in line 12 of Algorithm 5.1. Patterns are sorted in descending order based on their Hamming weights of missingness patterns.

While the length of the missingness patterns is determined by the linkage attributes $\mathbf{a}_L$, the shape of the lattice structure being generated is determined by the list $\mathbf{a}_R$, and the maximum number n of missing attributes in $\mathbf{a}_L$. Assuming $|\mathbf{a}_L| - n > 1$ (more than one attribute needs to contain a value), then the lower part of a lattice structure will be incomplete, as can be seen in Figure 5.4. There can however be situations where a single attribute value can be useful to meaningfully link records of the same entity. An example can be a mobile phone number attribute, where (even if all other attributes are missing), two records with the same mobile phone number likely refer to the same person.

In line 13 of Algorithm 5.1, the DOs participate in a secure set intersection (SSI) protocol [54] (the function $secSetIntersect()$), where the elements of the sets to be intersected are the missingness patterns in the local lattice structure, $\mathbf{L}_L$, of each DO. The output of the SSI protocol is the set of missingness patterns that occur in the local lattice structures of both DOs, which form a new lattice structure of all common bit patterns, $\mathbf{L}_C$.

The generated common lattice structure, $\mathbf{L}_C$, can be categorised into one of four types, which are (1) all missingness patterns are common (which means $\mathbf{L}_C \equiv \mathbf{L}_L$ for the local lattices of all DOs), (2) some missingness patterns are common across all local lattice structures, where there are common patterns in each layer of the generated common lattice structure (at least one pattern with a certain number of 1-bits occurs

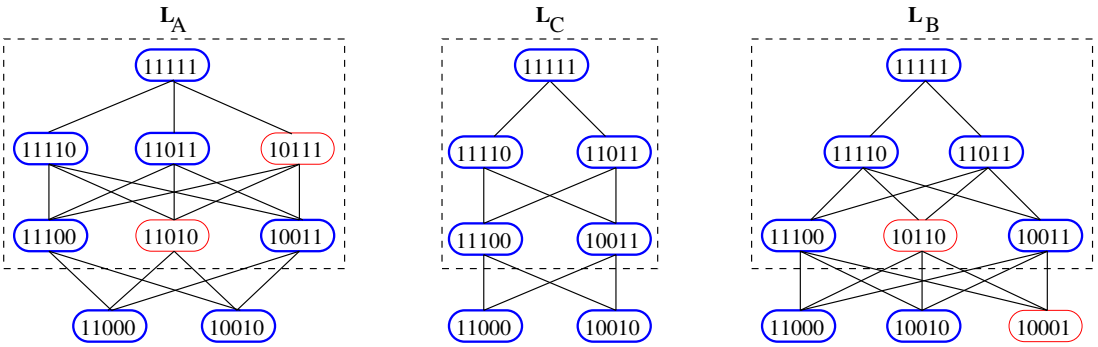

Figure 5.4: Local, $\mathbf{L}_A$ and $\mathbf{L}_B$, and the common $\mathbf{L}_C$ (middle), lattice structures where at least one common missingness pattern occurs in each layer of the $\mathbf{L}_C$. We set the maximum number of attributes that can be missing to $n = 3$ [177].

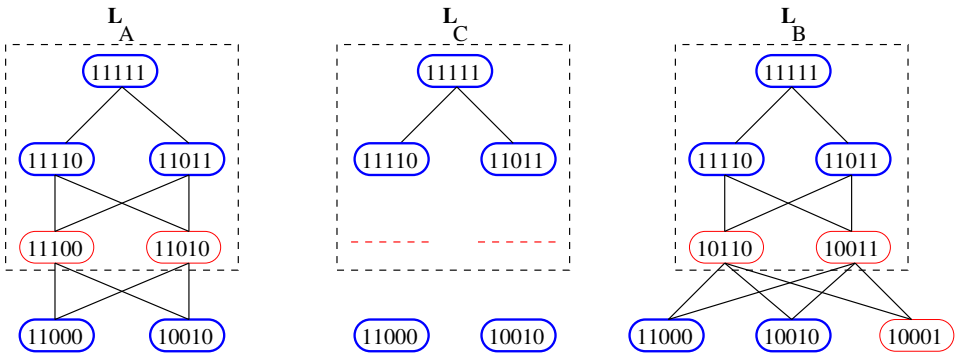

Figure 5.5: Local, $\mathbf{L}_A$ and $\mathbf{L}_B$, and the common $\mathbf{L}_C$ lattice structures, where $\mathbf{L}_C$ has a missing layer [177].

in common), (3) some patterns are common across all local lattice structures, however, there are some layers in the generated common lattice structure, $\mathbf{L}_C$, that have no common patterns, and (4) none of the patterns from the local lattice structures occur in common.

Our approach works with types (1) to (3), while for type (4) (no common patterns), it would be difficult to identify a way how records with different missingness patterns can be compared to obtain meaningful similarities between encoded records. Figures 5.4 and 5.5 show examples of types (2) and (3), respectively, where blue nodes show common missingness patterns between the two databases, red nodes show not common missingness patterns, and red dashed lines show a layer with no common missingness patterns. Solid links connect nodes to their upper and lower layers, and the dashed boxes show examples of the *upper* and *lower* grouping methods with Hamming weight differences by up to a threshold $d_t = 2$, as we describe next.

5.5.4 Grouping Missing Patterns into Partitions

A missingness pattern in the common lattice structure, $m_C \in \mathbf{L}_C$, is the basis of how partitions are formed, where records are inserted into one or more partitions based on their missingness patterns. We expand line 14 of Algorithm 5.1, where the function $groupPattern()$ is called, as outlined in Algorithm 5.2.

To compare records that have similar missingness patterns between DOs, for a common pattern m_C, its neighbouring patterns from the local lattice structure, $m_N \in \mathbf{L}_L$, are grouped into a set $\mathbf{m}_N$ based on the number of 1-bits that differ between their missingness patterns. The sets of ABFs that correspond to the records in those grouped patterns then form a partition, $\mathbf{P}$, as we formally describe in Definition 5.1.

Definition 5.1. ***Partition***
Given $m_C \in \mathbf{L}_C$, and one or more $m_N \in \mathbf{L}_L$, where m_C and each m_N represent a set of records, $\mathbf{r}_C \in \mathbf{D}$ and $\mathbf{r}_N \in \mathbf{D}$, respectively, in the database $\mathbf{D}$, and where $\mathbf{r}_C \cap \mathbf{r}_N = \emptyset$ for all $m_N \in \mathbf{L}_L$. We define a partition as $\mathbf{P} = \mathbf{r}_C \cup \mathbf{r}_N$ that represents the patterns m_C and one or more patterns m_N.

Generally, multiple neighbouring missingness patterns, m_N, are grouped with a common pattern to form a partition. Note that the set of neighbouring patterns can include both common and not common patterns. The records represented by a missingness pattern are likely grouped into several partitions based on the grouping method, g, used in Algorithm 5.2. The grouping of patterns can either be performed by grouping a given neighbouring pattern m_N to one or more common patterns m_C in the same and upper lattice layers (g = *upper*), to common patterns in the same and lower lattice layers (g = *lower*), or to common patterns in the same, upper, and lower lattice layers (g = *both*), as we discuss in more detail below.

For example in Figure 5.4, in $\mathbf{L}_A$, for the common pattern 11111 and with the threshold $d_t = 2$ (Hamming weights differ up to $d_t = 2$), the following other patterns will be its neighbouring patterns with grouping $upper$: 11110, 11011, 10111, 11100, 11010, and 10011. On the other hand, for the common pattern 10011 and grouping *lower*, the following other patterns will be its neighbouring patterns: 11111, 11110, 11011, 10111, 11100, and 11010.

We assume that the patterns in a local lattice structure, $\mathbf{L}_L$, are ordered based on their numbers of 1-bits (or their Hamming weight, calculated using the function $HW()$) with decreasing numbers of 1-bits. Based on the selected grouping method, g, and a maximum difference between Hamming weights (calculated using the function $HD()$ which returns the number of 1-bits that differs between two patterns), for each common pattern, we can define its neighbourhood of other local missingness patterns as in Definition 5.2.

Definition 5.2. ***Pattern neighbourhood***
For a common missingness pattern m_C and threshold of difference between Hamming weights $d_t \geq 1$, the set $\mathbf{m}_N$ contains the local missingness patterns $m_N \in \mathbf{L}_L$, with $m_N \neq m_C$, where the number of 1-bits that differs between m_C and a $m_N \in \mathbf{m}_N$ is not larger than d_t. Depending upon the grouping method, g, used, $\mathbf{m}_N$ will be limited to:

- $g = upper$: $\mathbf{m}_N = \{m_N \in \mathbf{L}_L : HD(m_N, m_C) \leq d_t \wedge HW(m_N) \leq HW(m_C)\}$
- $g = lower$: $\mathbf{m}_N = \{m_N \in \mathbf{L}_L : HD(m_N, m_C) \leq d_t \wedge HW(m_N) \geq HW(m_C)\}$
- $g = both$: $\mathbf{m}_N = \{m_N \in \mathbf{L}_L : HD(m_N, m_C) \leq d_t\}$

With g = *upper*, a neighbouring pattern m_N is grouped with a common pattern m_C in the same and upper lattice layers, where a m_C has the same or less missing attribute values. The larger number of missing attribute values in records with pattern m_N means their corresponding BF will contain more 0-bits compared to records with pattern m_C. With g = *lower*, a neighbouring pattern m_N is grouped with a common pattern m_C in the same and lower lattice layers, where a m_C has the same or more missing attribute values. Only the attribute values that are not missing in the pattern m_C are included in the BFs generated for this partition, while non-missing attributes in the records with pattern m_N are not included in these BFs. Less detailed information from a smaller number of attributes is encoded into BFs, and this can potentially lead to several records having the same or similar values in these attributes.

Finally, g = *both* generates partitions that include neighbouring patterns that are in the same, above, and below layers of a common pattern in the lattice structure. The generated partitions will be larger than for the two other grouping methods, and each record is likely inserted into more partitions. Only those attributes with no missing value in the common pattern, m_C, are encoded into BFs. Therefore, the neighbouring patterns in lower lattice layers will have more 0-bits in their corresponding BFs, while for neighbouring patterns in upper lattice layers, not all of their non-missing attributes will be encoded into BFs.

For every grouping method, we need to group patterns in the same layer of a lattice structure to ensure the records that possibly refer to the same entity will be compared. For example, assuming the DOs aim to link records of five attributes which are first name, last name, street address, city, and zip code. The first DO has the record r_1 with the pattern $m_N = 11101$ (attribute city has missing value) and the second DO has the record r_2 with the pattern $m_N = 11110$ (attribute zip code has missing value). There is the common pattern $m_C = 11011$ (attribute street address has missing value). Grouping the patterns in the same layer will allow r_1 with $m_N = 11101$ and r_2 with $m_N = 11110$ to be grouped with the common pattern $m_C = 11011$, and therefore the records r_1 and r_2 that might have the same first and last names will be compared.

As outlined in Algorithm 5.2, the inverted index of partitions, $\mathbf{P}$, is first initialised in line 1. The DO then loops over the missingness patterns in the common lattice structure $m_C \in \mathbf{L}_C$ and inserts the current common pattern m_C into $\mathbf{P}$ in lines 2 and 3. The set of neighbouring missingness patterns, $\mathbf{m}_N$, is then generated in line 4.

Algorithm 5.2 : ***Grouping Patterns into Partitions***

Input:
- $\mathbf{L}_L$: Local lattice structure of a database
- $\mathbf{L}_C$: Common lattice structure
- $\mathbf{w}$: List of attribute weights
- g : Grouping method
- w_t : Weight threshold
- d_t : Threshold of Hamming weights different

Output:
- $\mathbf{P}$: Inverted index of partitions

```
1:  P ← {}                                          // Initialise inverted index of partitions
2:  for m_C ∈ L_C do:                               // Loop over common missingness patterns
3:    P[m_C] ← m_C                                  // Start the partition with the common pattern itself
4:    m_N ← getNeighbours(m_C, L_L, g, d_t)         // Get neighbouring patterns of m_C
5:    for m_N ∈ m_N do:                             // Loop over neighbours of the common pattern
6:      w_N ← sumDiffWeight(w, m_C, m_N)            // Sum weights of difference bits between m_C and m_N
7:      if w_N ≤ w_t do:                            // Check if weight is at most the weight threshold
8:        P[m_C].add(m_N)                           // Add neighbouring pattern to the current partition
9:  return P
```

In lines 5 to 8, each neighbouring pattern $m_N \in \mathbf{m}_N$ is further assessed based on the weights assigned to attributes, $\mathbf{w}$, which is defined by the DOs to identify the importance of each attribute in $\mathbf{a}_L$, where a higher weight means the attribute is more relevant for the linkage process. For example, the DOs might assign a higher weight to an attribute first name compared to an attribute zip code. This is because the attribute first name is generally more important to identify records that refer to the same individual than zip code. The weights in $\mathbf{w}$ can either be set based on a domain expert or using the match weight calculations employed in probabilistic record linkage [67, 87].

The weight threshold, w_t, ensures that only those records that are of close importance (based on their weights) are grouped into the same partition. Therefore, two patterns, m_C and m_N, that are highly dissimilar, for example a record with a first name value and a record with a missing first name value, will not be grouped into a partition. In line 6, the calculation of the weight w_N in the function $sumDiffWeight()$ first identifies the attributes with missing values that differ between m_C and m_N by applying the bit-wise XOR operation ($\oplus$) on these bit patterns: $m_d = m_C \oplus m_N$, where any 1-bit in the bit vector m_d that corresponds to an attribute in $\mathbf{a}_L$ can be either missing in m_C or m_N, but not both. The weight w_N is then calculated by summing all attribute weights in $\mathbf{w}$ where the corresponding bit in m_d is set to 1:

$$w_N = \sum_{i=1}^{|\mathbf{a}_L|} \begin{cases} \mathbf{w}[i] & if\ m_d[i] = 1, \\ 0 & otherwise. \end{cases} \tag{5.1}$$

If the summed weight w_N is below or equals the weight threshold w_t for a neighbouring pattern m_N, then m_N will be added to the partition in lines 7 and 8. The weight threshold w_t therefore determines how different the records in the missingness patterns grouped into the same partition can be with regard to the attributes that have missing values. A smaller weight threshold w_t means that the patterns m_N

and m_C need to be very similar in order to be grouped into the same partition. For example, let us assume $w_t = 0.7$ and the five attributes first name, last name, street address, city, and zip code being linked are assigned the weights 0.8, 0.9, 0.6, 0.4, and 0.2, respectively. As shown in the lattice of database $\mathbf{D}_A$, $\mathbf{L}_A$, in Figure 5.4, assuming we use a grouping method g = *upper* and a threshold $d_t = 1$, then the neighbouring pattern 11010 can possibly be grouped with the common pattern 11110, 11011, 11100, and/or 10011. We can calculate the weights of the corresponding different bits as:

- 11110 $\oplus$ 11010 = 00100 $\rightarrow w_N = 0.8 \times 0 + 0.9 \times 0 + 0.6 \times 1 + 0.4 \times 0 + 0.2 \times 0 = 0.6$
- 11011 $\oplus$ 11010 = 00001 $\rightarrow w_N = 0.8 \times 0 + 0.9 \times 0 + 0.6 \times 0 + 0.4 \times 0 + 0.2 \times 1 = 0.2$
- 11100 $\oplus$ 11010 = 00110 $\rightarrow w_N = 0.8 \times 0 + 0.9 \times 0 + 0.6 \times 1 + 0.4 \times 1 + 0.2 \times 0 = 1.0$
- 10011 $\oplus$ 11010 = 01001 $\rightarrow w_N = 0.8 \times 0 + 0.9 \times 1 + 0.6 \times 0 + 0.4 \times 1 + 0.2 \times 1 = 1.1$

As a result, the weight $w_N = 0.6$ for $m_N = 11010$ and $m_C = 11110$ (with missing zip code), and $w_N = 0.2$ for $m_N = 11010$ and $m_C = 11011$ (with missing street address) are only two below the threshold $w_t = 0.7$, and therefore the $m_N = 11010$ (with missing street address and zip code) is added to the partitions formed by the common missingness patterns 11110 and 11011. The output of Algorithm 5.2 is an inverted index of partitions, $\mathbf{P}$, where each record in a database occurs in one or more partitions (unless the record contains more than n missing values or an attribute in $\mathbf{a}_R$ is missing).

5.6 Linkage Process

We now describe two different methods of how the DOs communicate with the LU, and how the LU compares the record pairs in the partitions it receives from the DOs.

5.6.1 Iterative Linkage

The iterative linkage method aims to ensure that each record pair is compared only once even if a certain record occurs in several partitions. As outlined in Algorithm 5.3, the DO first initialises the inverted index of matched pairs, $\mathbf{R}$, and the set of matched record identifiers, $\mathbf{c}$, in lines 1 and 2. The DO then sorts partitions in $\mathbf{P}$, that were generated in Algorithm 5.2, with an increasing number of missing values in their common missingness pattern, m_C, in line 3. This is because partitions with less missing values allow for a more meaningful comparison of records given they contain more non-missing attribute values.

In line 4, the DO loops over these sorted partitions, where the DO retrieves the common missing pattern m_C and the list of all patterns that were grouped into this partition, $\mathbf{m}_p$. The DO then initialises a list $\mathbf{B}_p$ which will hold all record BFs that will be generated for this partition in line 5, and loops over all missingness patterns, m, in the partition in line 6. For each such pattern, in line 7, the DO retrieves the identifiers of all records with that pattern as the list $\mathbf{r}_m$ from the missing pattern table $\mathbf{M}$. In line 8, the DO removes any record identifier from $\mathbf{r}_m$ that is in the set of matched records,

Algorithm 5.3: ***Iterative Linkage Method***

Input:
- **A** : Inverted index of ABFs
- **M** : Missing pattern table
- **P** : Inverted index of partitions
- **w** : List of attribute weights
- s_t : Similarity threshold

Output:
- **R** : Inverted index of matched record pairs

```
1:  R ← {}                                          // Initialise inverted index of matched pairs
2:  c ← {}                                          // Initialise set of matched record IDs
3:  P ← sortPartByNumMiss(P)                        // Sort partitions
4:  for (m_C, m_p) ∈ P do:                          // Loop over partitions
5:    B_p ← {}                                      // Initialise the inverted index of BFs for partition
6:    for m ∈ m_p do:                               // Loop over missing patterns in partition
7:      r_m ← M[m]                                  // Get record identifiers with pattern m
8:      r'_m ← remCompRec(r_m, c)                   // Remove records already matched
9:      for r.id ∈ r'_m do:                         // Loop over all records in r'_m
10:       b_r ← A[r.id]                             // List of ABFs for record r.id
11:       b_r ← genBF(b_r, m_C, w)                  // Generate the BF for record r.id
12:       b'_r ← permBF(b_r, m_C)                   // Partition specific BF permutation
13:       r.eid ← encrRecID(r.id, m_C)              // Encrypt record ID
14:       B_p[r.eid] ← b'_r                         // Add b'_r under key r.eid to this partition
15:   sendToLU(B_p, s_t)                            // Send the BFs to the LU for matching
16:   R_p ← receiveFromLU()                         // Receive matched record pairs
17:   for (r_1.eid, r_2.eid) ∈ R_p do:              // Process matched record pairs
18:     s ← R_p[(r_1.eid, r_2.eid)]                 // Get record pair similarity
19:     r_1.id, r_2.id ← decrRecIDs(r_1.eid, r_2.eid, m_C)   // Get original IDs
20:     R[(r_1.id, r_2.id)] ← s                     // Add to all matches
21:     c.add(r_1.id)                               // Add to set of matched records
22: return R
```

c, to prevent a previously matched record to be compared again. The DO then loops over the set $\mathbf{r}'_m$ of not yet matched records with a given missingness pattern m, retrieves the corresponding list of ABFs, $\mathbf{b}_r$, and generates one BF b_r per record using the function $genBF()$, which as input also takes the missingness pattern m_C of this partition, and the list of attribute weights, **w**, in lines 9 to 11.

To ensure the BFs in all partitions are generated with the same length, we distribute the weights of those attributes that have missing values to other attributes that are non-missing, based on the missingness pattern, m_C [130]. This weight distribution is calculated as:

$$\mathbf{w}'[a] = \mathbf{w}[a] + \frac{\sum \mathbf{w}[a_m]}{HW(m)}, \tag{5.2}$$

where $\mathbf{w}[a]$ is an attribute weight, $\mathbf{w}[a_m]$ is the weight of a missing value attribute a_m, $HW()$ calculates the Hamming weight, and m is a missingness pattern.

For example, for a given record, let us assume the attributes first name, last name, and street address are non-missing, while attributes city and zip code are missing. The attributes were originally assigned the weights 0.8 (first name), 0.9 (last name), 0.6 (street address), 0.4 (city), and 0.2 (zip code), respectively. The weights 0.4 and 0.2 of city and zip code are distributed to the other three attributes as:

$$\begin{aligned}
\mathbf{w}'[\text{first name}] &= 0.8 + (0.4 + 0.2)/3 = 1.0, \\
\mathbf{w}'[\text{last name}] &= 0.9 + (0.4 + 0.2)/3 = 1.1, \\
\mathbf{w}'[\text{street address}] &= 0.6 + (0.4 + 0.2)/3 = 0.8.
\end{aligned}$$

After distributing the weights of missing value attributes, we sample bits from each corresponding ABF to generate the final record BF [59]. For each non-missing attribute a we calculate its number of bits, $\mathbf{s}[a]$, to be sampled (rounded to the nearest integer) as:

$$\mathbf{s}[a] = \left\lfloor \frac{\mathbf{w}'[a]}{\sum_a \mathbf{w}'[a]} \times l \right\rceil, \tag{5.3}$$

where $\mathbf{w}'[a]$ is the weight for attribute a adjusted using Equation 5.2, and l is the length of an ABF. Continuing the above example and assuming an ABF length of $l =$ 1,000 bits, the number of sampled bits will be 345 for first name, 379 for last name, 276 for street address, 0 for city and zip code (because they are missing), resulting in the final record BF again of length l = 1,000 bits.

The generated BFs for all records in a partition are then permuted in line 12 of Algorithm 5.3, where the permutation is conducted in a way agreed by the DOs but kept secret from the LU. The partition's common missingness pattern, m_C, is used as the seed (for example, a pseudo-random number generator (PRNG) [38]) upon which the permutation is based. As a result, it will be difficult for the LU to successfully apply a frequency based cryptanalysis attack [39, 108, 109] across the BFs from different partitions. Furthermore, to prevent the LU can identify the BFs that correspond to the same record across partitions (using the record identifiers, *r.id*), in line 13, the DO generates a partition specific encrypted record identifier, *r.eid*. Only these encrypted record identifiers are sent to the LU together with their partition specific BFs and the similarity threshold s_t used to decide if compared BFs are matching or not (in lines 14 and 15).

Once the LU has compared and linked the BFs in a partition, as we describe in Section 5.6.3, it returns a set of matched record pairs (with their similarities) from that partition, $\mathbf{R}_p$, in line 16. In lines 17 to 19, the DO then loops over these matches, retrieves the similarity s of a matched record pair, and decrypts the encrypted record identifiers back to their original values. The DO then adds the pair to the set **R** of all matches in line 20, and the identifier of the local record of the pair (assumed to be r_1) to the set **c** of matched record identifiers in line 21. This means a matched record will not be included in any following partitions. The DO repeats steps in lines 4 to 21 until all partitions in **P** have been processed.

5.6.2 Batch Linkage

The aim of the batch linkage method is to limit the communication between the DOs and the LU to one single exchange of messages. As outlined in Algorithm 5.4, the DO first initialises the inverted index of BFs, **B**, and inverted index of record identifiers mapping, **E**, in lines 1 and 2. In line 3, the DO loops over the partitions, **P**, generated

Algorithm 5.4: ***Batch Linkage Method***

Input:
- **A**: Inverted index of ABFs
- **M** : Missing pattern table
- **P**: Inverted index of partitions
- **w**: List of attribute weights
- s_t: Similarity threshold

Output:
- **R**: Inverted index of matched record pairs

```
1:  B ← {}                                          // Initialise the inverted index of BFs
2:  E ← {}                                          // Initialise the inverted index of record ID mappings
3:  for (m_C, m_p) ∈ P do:                          // Loop over partitions
4:    for m ∈ m_p do:                               // Loop over missing patterns in partition
5:      r_m ← M[m]                                  // Get record identifiers with pattern m
6:      for r.id ∈ r_m do:                          // Loop over all records with this pattern m
7:        b_r ← A[r.id]                             // List of ABFs for record r.id
8:        b_r ← genBF(b_r, m_C, w)                  // Generate the BF for record r.id
9:        b'_r ← permBF(b_r, m_C)                   // Partition specific BF permutation
10:       r.eid ← encrRecID(r.id, m_C)              // Encrypt record ID
11:       E[r.eid] ← (r.id, m_C)                    // Keep record ID mapping
12:       B[r.eid] ← b'_r                           // Add b'_r to inverted index of BFs
13: sendToLU(B, s_t)                                // Send the BFs to the LU for matching
14: R' ← receiveFromLU()                            // Receive matched record pairs
15: R ← getBestMatches(R', E)                       // Select the best matching record pairs
16: return R
```

in Algorithm 5.2 and retrieves the common missingness pattern, m_C, and the list of all patterns that were grouped into this partition, $\mathbf{m}_p$. The DO then loops over all patterns in the partition in line 4. For each such pattern m, the DO retrieves the identifiers of all records with that pattern as the list $\mathbf{r}_m$ from the missing pattern table **M** in line 5.

To generate record BFs for all records with a given missing pattern m, the DO loops over the list $\mathbf{r}_m$ in line 6, and for each record, the DO retrieves the list of ABFs, $\mathbf{b}_r$, to generate one BF b_r per record using the function $genBF()$ according to the missing pattern m_C and the list of attribute weights **w** in lines 7 and 8. The function $genBF()$ applies the weight distribution (as discussed in Section 5.6.1). The generated BF is then permuted in line 9, using the pattern m_C as the seed of the permutation function, and a partition specific encrypted record identifier, $r.eid$, is generated in line 10. The mapping of the original to encrypted record identifier is stored in the mapping table **E** in line 11. The permuted BF with its corresponding encrypted identifier is then added to the inverted index of all BFs, **B**, in line 12.

Once all record BFs are generated and added to **B**, the DO sends **B** with the similarity threshold, s_t, to the LU to conduct matching in line 13. The LU compares and matches the BFs received from the DOs as we describe in Section 5.6.3, and returns all matched record pairs, $\mathbf{R}'$, together with their similarities in line 14.

From the matched record pairs, $\mathbf{R}'$, the DO received from the LU, in line 15, the DO uses the function $getBestMatches()$ to identify the best matching record pairs in $\mathbf{R}'$. This is achieved in a similar way as is shown in lines 17 onward in Algorithm 5.3

Algorithm 5.5: ***Blocking and Linking RBFs***

Input:
- $\mathbf{B}_1$: Inverted index of encrypted record identifiers and BFs from the first DO
- $\mathbf{B}_2$: Inverted index of encrypted record identifiers and BFs from the second DO
- s_t: Similarity threshold

Output:
- $\mathbf{R}$: Matched record pairs

```
1:  R ← {}                                   // Initialise inverted index of matched record pairs
2:  G_1 ← genBlocks(B_1)                     // Generate blocks for BFs from first DO
3:  G_2 ← genBlocks(B_2)                     // Generate blocks for BFs from second DO
4:  G_c ← G_1 ∩ G_2                          // Get the common blocks
5:  for g ∈ G_c do:                          // Loop over common blocks
6:    for (r_1.eid, b_1) ∈ B_1[g] do:        // Loop over BFs in block from first DO
7:      for (r_2.eid, b_2) ∈ B_2[g] do:      // Loop over BFs in block from second DO
8:        s ← sim_D(b_1, b_2)                // Calculate Dice-coefficient, Equation 2.6, between BFs
9:        if s ≥ s_t do:                     // Check if record pair is a match
10:         R[(r_1.eid, r_2.eid)] ← s        // Add record pair to matches
11: return R                                 // Send matched record pairs to DOs
```

for the iterative linkage method. This function uses the mapping table $\mathbf{E}$ to obtain the original record identifiers and partition keys (m_C) to group the matched record pairs into their partitions. Record pairs with the smallest number of missing values are then identified as the final best matches first, resulting in the set of best matching record pairs, $\mathbf{R}$, which is returned in line 16 of Algorithm 5.4.

5.6.3 Record Pair Comparison by the Linkage Unit

In Algorithm 5.5, we outline the steps that the LU performs for linking a pair of inverted indexes (partitions) of BFs. In line 1, the LU first initialises an inverted index of matched record pairs, $\mathbf{R}$. The LU then applies a blocking technique on the BFs received from both DOs in lines 2 and 3, where we assume this blocking technique is a black box that groups records into blocks [99, 184].

In line 4, the LU then identifies the common blocks between both databases, and in line 5, it loops over these blocks, $\mathbf{g}$. The nested loops in lines 6 and 7 retrieve all encrypted record identifiers, $r_1.eid$ and $r_2.eid$, and their corresponding BFs, b_1 and b_2, in a block $\mathbf{g}$ and iterate over all possible pairs of records in the block. In line 8, the LU compares the BFs for a record pair using the Dice-coefficient similarity function, $sim_D()$, given in Equation 2.6. If the resulting similarity s is at least the similarity threshold s_t, then a pair is classified as a match and inserted into the inverted index of matches $\mathbf{R}$ with its similarity s in lines 9 and 10. Finally, in line 11, the LU sends $\mathbf{R}$ back to the DOs.

5.7 Analysis

We now analyse our approach with regard to linkage quality, scalability, and privacy. We focus on the specific aspects of grouping records with different missingness

patterns into partitions, encoding these partitions using different weights assigned to non-missing attributes, and applying partition specific permutations of bit positions. For general discussions about the privacy of BF encoding for PPRL we refer the reader to Durham et al. [59] and Christen et al. [38, 39].

5.7.1 Linkage Quality Analysis

The patterns of how missing values occur in the attributes used for linkage will be the biggest factor influencing linkage quality [6, 30, 68, 76]. In our approach, based on the maximum number of missing values, n, and the sets $\mathbf{a}_L$ and $\mathbf{a}_R$, a user can customise a linkage based on their knowledge of both data quality and missingness patterns in the databases to be linked.

For PPRL based on BF encoding, the major parameters that determine linkage quality are the BF length, l, the q-gram length, q, and the number of hashes [155, 181] used. If RBF encoding [59] is employed, as we adopt in our approach, then bit sampling based on weights assigned to attributes, calculated for example using the probabilistic record linkage approach [67], will also influence linkage quality.

In our approach, the grouping method, g, and the thresholds, d_t and w_t, will influence how partitions of BFs are generated, which in turn determines how record pairs with certain missingness patterns will be compared. Larger d_t and w_t values mean more missingness patterns are grouped into a partition, and therefore records are inserted into more partitions. This leads to more record pairs being compared, thus, an increased chance for the true matching record pairs to be compared (higher recall) and more false matching pairs to be classified as matches (lower precision).

The grouping method influences linkage quality. This is because it determines how records are grouped into partitions and then compared by the LU. For g = *upper*, neighbouring pattern, m_N, is grouped with a common pattern, m_C, in an upper lattice layer leading to a decrease in the number of true matching record pairs (lower recall) but also to less false matching (higher precision). This is because the larger number of missing values in records with pattern m_N means their corresponding encoded BFs contain more 0-bits compared to records with pattern m_C.

For g = *lower*, neighbouring pattern m_N is grouped to a common pattern m_C in a lower lattice layer which can lead to an increase in the number of true matching record pairs (higher recall) but also more false matching pairs (lower precision). This is because only the attribute values that are not missing in the pattern m_C are included in the BFs generated for this partition, and there are non-missing attributes in the records with pattern m_N that are not included in these BFs. With less detailed information from a smaller number of attributes encoded into BFs, this potentially leads to several records having the same or similar values in these attributes.

Finally, g = *both* generates partitions that include neighbouring patterns m_N that are in the same, above, and below lattice layers of a pattern m_C. The generated partitions will be larger than with the other grouping methods, and each record will potentially be grouped into a larger number of partitions. As a result, the linked

databases might have an increased number of true matching record pairs (higher recall), but potentially also a larger number of false matching pairs (lower precision).

When the iterative linkage method is employed, the DOs can order the partitions being generated and sent to the LU for linkage, where record pairs with the lowest number of missing attribute values are compared and matched first. This ensures high linkage quality because once a compared record pair is classified as a match, each DO marks its corresponding record as matched (line 21 in Algorithm 5.3), and therefore the record will not be considered in later partitions (and compared with records that might have more missing values).

With the batch linkage method, the LU compares pairs of BFs based on their bit patterns because it does not know in which partition a BF occurs which can lead to wrongly matched record pairs returned from the LU to the DOs. However, because the DOs know the partitions of BFs (based on the mapping table **E** used in Algorithm 5.4), they can group record pairs back into partitions and link those pairs with the smallest number of missing values and highest similarities first.

5.7.2 Scalability Analysis

We analyse the scalability of the main steps of our approach, as shown in the five algorithms. The first step performed by the DOs is the generation of ABFs and the missing pattern table, **M**. The complexity of this step is $O(|\mathbf{D}| \times |\mathbf{a}_L|)$, where $|\mathbf{D}|$ is the number of records in the database being encoded and $|\mathbf{a}_L|$ is the number of linkage attributes. The DOs then build their local lattice structure based on the table **M** which results in the maximum number of missingness patterns (lattice nodes) is $p = \sum_{i=0}^{|\mathbf{a}_L|-n} \binom{|\mathbf{a}_L|}{i}$, where n is the maximum number of allowed missing attribute values.

Using a secure set intersection protocol [53], the local lattice structures are exchanged between the DOs. The sets to be intersected are of size $O(p)$, where efficient protocols that have a communication and computation complexity linear in the sizes of the input sets are available [53]. Once the common lattice structure is obtained, each DO generates the partitions, **P**, where the number of partitions is limited by $O(p)$ because each partition has a missingness pattern from the common lattice structure as its key.

In the iterative linkage method, one BF is generated per record from a maximum of $|\mathbf{a}_L|$ attributes used for linkage. Depending upon the grouping method used, each record can be inserted into several partitions. The number of neighbouring patterns to be considered for a partition depends upon the value of the threshold, d_t. For $d_t = 1$, the number of neighbouring patterns is $O(|\mathbf{a}_L|)$, for $d_t = 2$ it is $O(|\mathbf{a}_L|^2)$, and so on. The largest number of neighbouring patterns to be grouped into a partition will occur when grouping *both* is used. If we assume each record in a database **D** is inserted into $|\mathbf{a}_L|^{d_t}$ partitions, then the iterative linkage method will have a worst-case complexity of $O(|\mathbf{a}_L|^{d_t} \times |\mathbf{D}|)$ of the total number of BFs to be generated by each DO. The number of partitions to be sent from each DO to the LU is $O(p)$. In practice,

however, the complexity of the iterative linkage method will be much lower because once a record has been classified as a match it is not considered in later iterations. As a result, as more records are matched, partitions are getting smaller.

The batch linkage method has the same worst-case complexity as the iterative method. However, because there is only one communication step, the filtering of matched records used in the iterative method cannot be applied, thus, a single communication step of $O(|\mathbf{a}_L|^{d_t} \times |\mathbf{D}|)$ BFs is required from each DO to the LU.

Finally, the number of comparisons of pairs of BFs by the LU depends upon the blocking technique employed [38]. In a worst-case scenario, we can assume no blocking is conducted and every possible pair of BFs in a partition is compared. Assuming p partitions, the number of pairs in the iterative linkage method would be $O((|\mathbf{a}_L|^{d_t} \times |\mathbf{D}|)^2/p)$, while for the batch method it would be $O((|\mathbf{a}_L|^{d_t} \times |\mathbf{D}|)^2)$.

5.7.3 Privacy Analysis

We now analyse what the parties involved in our approach can learn from the data they receive from each other. We assume all parties follow the honest-but-curious (HBC) adversary model [116] (as described in Section 2.2.2), where however the DOs are not colluding with the LU [30].

The DOs first agree on the parameter settings to be used in the linkage process. While the list of linkage attributes, $\mathbf{a}_L$, and the list of attribute weights, $\mathbf{w}$, allow each DO to learn about the common attributes and their importance in the other databases, no sensitive information about individuals is being revealed in this step. None of the parameters needed to generate the BFs, nor the method of how the permutation of BFs are conducted, and neither the similarity threshold, s_t, used to classify compared RBFs as matches or non-matches, will reveal any sensitive information.

The DOs then generate their own missing pattern table and local lattice structure, and identify the common patterns by employing a secure set intersection protocol [53]. From this protocol, each DO learns missingness patterns that occur and do not occur in the other database (the missingness patterns in the local lattice structure of a DO that do not occur in the common lattice structure). However, a DO cannot learn the number of records in a partition, nor any sensitive information about individual records in the other database.

The DOs individually encode their sensitive attribute values, first into ABFs [155] and then into RBFs [59], where it has been shown that RBFs are more secure than ABFs with regard to several cryptanalysis attacks [39, 109, 129, 188]. In our approach, we employ an adapted RBF method where, like the original RBF approach [59], we sample bits and permute the final BFs. However, while the full databases are encoded in the same way (same bit positions sampled and bits permutation applied to all BFs) in the original RBF method, we apply different sampling (based on weight distribution) and different bit position permutations to each partition.

Given most known cryptanalysis attacks on BF encoding for PPRL exploit frequent bit patterns in a set of BFs [39, 109, 129, 188], our approach will be more secure

Table 5.2: Example of records with patterns $m_C = 1111$ and $m_N = 1011$.

Record identifier	First name	Last name	City	Zip code
r_1	mary	cooper	wilson	27896
r_2	marry	miller	wilson	27895
r_3	mary		wilson	27896

compared to the original RBF approach. This is because bit patterns in BFs cannot be analysed across partitions. Furthermore, in the batch approach, all partitions are concatenated into one single set of BFs before being sent to the LU, thus, the LU cannot identify which subset of BFs corresponds to a partition.

During the iterative linkage method, and as the final step of the batch linkage method, the LU sends the encrypted identifiers of those record pairs classified as matches (with a similarity of at least s_t, as outlined in Algorithm 5.5) back to the DOs. Because the DOs know how the BFs were constructed, how record identifiers were encrypted, and how missingness patterns were grouped into a partition, they can analyse these matched record pairs and their similarities. As with any other PPRL protocol, for any record pair with a similarity of $s = 1$ (an exact match), both DOs learn that they have a record in common where all compared attributes are the same (depending upon the missingness pattern of records in a partition). However, such information leakage does not happen if several patterns are grouped into a partition.

For each record pair that has a similarity $s < 1$ (an approximate match), there are several possible cases of how this similarity was obtained. For example, assume the common missingness pattern $m_C = 1111$ and the neighbouring pattern $m_N = 1011$ have been grouped into a partition (using g = *upper*), and the three records as we illustrate in Table 5.2 are one (r_1) from the database $\mathbf{D}_A$, and two (r_2 and r_3) from the database $\mathbf{D}_B$. The similarities for both record pairs (r_1, r_2) and (r_1, r_3) are below 1. For both pairs, the DO of $\mathbf{D}_A$ cannot learn if the records in $\mathbf{D}_B$ contain missing values or not, because it does not know the missingness pattern of records from the other database. Therefore, the grouping methods make it more difficult for a curious DO to identify the record values of the other DO in any matching record pair that has a similarity $s < 1$.

One exception to this improved privacy is with g = *lower*. Because with this method likely a smaller number of attributes is encoded into BFs, there is a higher chance that record pairs end up with a similarity of $s = 1$. As seen in the example in Table 5.2, if the last name is not compared, then the record pair (r_1, r_3) will end up with a similarity of $s = 1$. As a result, the DOs will learn which values in a subset of attributes are the same in record pairs that have a similarity $s = 1$. The LU will learn nothing besides that a record pair has a similarity of $s = 1$, but it is not able to learn this subset nor the actual compared values.

To summarise the privacy of our three proposed grouping methods, g = *both* will lead to the largest partitions, each including several missingness patterns, and therefore likely results in the highest uncertainty about which pattern a certain record has. The g = *upper* method also provides increased uncertainty because the missingness pattern of a record still cannot be determined with certainty if a partition does contain several patterns. Only if a partition consists of the common missingness pattern only can a DO determine that an exact match with similarity $s = 1$ means all not missing attribute values are the same in a pair of compared records. Finally, the g = *lower* method provides the least privacy protection because less attributes are being compared, and it will be more likely that exact matches will occur for record pairs.

In comparison processed by the LU, given the generally wide range in the lengths of attribute values (with the exception of an attribute that contains values with the same lengths, such as zip code), a curious LU will not be able to distinguish the BFs that correspond to records with a missing value in an attribute from those that do have shorter values across their attributes. Even if the LU could identify which records have a missing value, the use of RBFs and sampling of bit positions does mean no information about the not missing values in that record is being revealed [59].

5.8 Experimental Evaluation

We first describe the generation of datasets with missing values and the experimental setup in Sections 5.8.1 and 5.8.2, respectively. We then discuss the experimental results of our approach compared to the baselines, starting with the linkage quality results in Section 5.8.3. After that, in Section 5.8.4, we discuss the scalability results, and in Section 5.8.5 we then assess the privacy of our approach compared to the baselines using a cryptanalysis attack [39].

5.8.1 Generating Datasets with Missing Values

We based all our experiments on a large real-world database, the North Carolina Voter Registration[1] (NCVR), from a snapshot of 2019, as we described in Chapter 2. We used attributes first name, middle name, last name, street address, city, and zip code as the set of linkage attributes, $\mathbf{a}_L$. We modified the GeCo data corruptor [41] to generate pairs of datasets with different data quality characteristics, where each dataset in a pair contained 100,000 records. During the corruption process, we kept the identifiers of the selected and modified records, which allowed us to identify true matches and calculate linkage quality. We then generated a corrupted version of these datasets by applying corruption functions [41] on between 1 to 3 randomly selected attribute values on 20% of all records, resulting in 80% of true matching record pairs were exact duplicates and 20% of pairs were only approximate matching.

[1] http://dl.ncsbe.gov/

Table 5.3: Numbers of records with missing values in different attributes for the two datasets ($\mathbf{D}_A$ / $\mathbf{D}_B$) out of 100,000 records, where K = 1,000 records. Both datasets with 0% and 20% of corruption have the same numbers of records with missing values.

Dataset	First Name	Middle Name	Last Name	Street Address	City	Zip Code
No missing layer, 20% missing	0/0	11K/11K	12K/12K	6K/10K	10K/ 8K	0/0
No missing layer, 50% missing	0/0	27K/27K	30K/30K	15K/25K	25K/20K	0/0
With missing layer, 20% missing	0/0	13K/ 7K	13K/14K	4K/13K	11K/ 7K	0/0
With missing layer, 50% missing	0/0	34K/19K	34K/34K	11K/34K	27K/19K	0/0

We then introduced missing values into 20% and 50% of records, respectively, for the attributes middle name, last name, street address, and city. We did not introduce any missing values into the attributes first name and zip code in order to be able to use these for blocking (to ensure blocking is not affected by missing values). Note that true matching record pairs can have the same or different missingness patterns in their attributes.

We introduced different missingness patterns into the generated datasets to allow us to evaluate our proposed approach under two scenarios: (1) missing completely at random (MCAR), and (2) missing at random (MAR), as we discussed in Section 5.2. For MCAR, we randomly sampled a subset of records from a dataset and introduced missing values into the attributes based on the generated missingness patterns. This ensures missing values are introduced into records without any correlations with any other attribute values in the same record. For MAR, we introduced missingness patterns according to the zip code attribute by first randomly sampling records based on their zip code values. Next, we selected a missingness pattern randomly for such records and then introduced missing values into the corresponding attributes according to that pattern. As a result, the missingness patterns are correlated with the values in the zip code attribute.

We also generated pairs of datasets that resulted in a common lattice of types (2) and (3), as we illustrated in Figures 5.4 and 5.5. In total, we have generated eight pairs of MCAR and eight pairs of MAR datasets, where four pairs had missingness patterns with a missing layer (type (3)) and four pairs had missingness patterns in all lattice layers (type (2)). Each of these four pairs consists of two with 0% and 20% corruptions, one each with 20% and 50% missing values, respectively. Table 5.3 shows the number of records containing missing values in certain attributes in the datasets of 100,000 records, and Table 5.4 shows the numbers of records with a specific missingness pattern in these datasets. The numbers of records with a certain missingness pattern are the same for MCAR and MAR datasets.

5.8.2 Experimental Setup

We compared our approach with two baseline approaches. The first baseline is the k-nearest neighbour (named k-NN) based PPRL approach for missing values proposed

Table 5.4: Missingness patterns and their numbers of records in the dataset pairs ($\mathbf{D}_A/\mathbf{D}_B$) with 100,000 records, where K = 1,000 records.

Pattern	Without missing layer		With missing layer	
	20% miss	50% miss	20% miss	50% miss
111111	80K / 80K	50K / 50K	80K / 80K	50K / 50K
101111	2K / 2K	5K / 5K	0 / 0	0 / 0
110111	2K / 2K	5K / 5K	3K / 4K	8K / 8K
111011	2K / 0	5K / 0	0 / 0	0 / 0
111101	2K / 2K	5K / 5K	4K / 3K	8K / 8K
100111	2K / 2K	5K / 5K	3K / 0	8K / 0
101101	2K / 0	5K / 0	3K / 0	8K / 0
101011	0 / 2K	0 / 5K	0 / 3K	0 / 8K
110101	2K / 2K	5K / 5K	0 / 0	0 / 0
110011	0 / 2K	0 / 5K	0 / 3K	0 / 8K
100011	2K / 2K	5K / 5K	3K / 3K	7K / 7K
100101	2K / 0	5K / 0	3K / 0	7K / 0
101001	0 / 2K	0 / 5K	0 / 0	0 / 0
110001	1K / 1K	3K / 3K	0 / 3K	0 / 7K
100001	1K / 1K	2K / 2K	1K / 1K	4K / 4K

by Chi et al. [30], as discussed in Chapter 3 and Section 5.1. We used $k = [3, 5, 10]$ as the number of nearest neighbour records that are similar to a record with a missing value in a certain attribute. We calculated the similarities between ABFs with missing and not missing values by using the Dice-coefficient similarity (following Equation 3.8 as proposed by Chi et al. [30]). We obtained very similar results for the three values of k, and therefore we only report the results when using $k = 10$.

The second baseline is the record-level Bloom filter (RBF) approach proposed by Durham et al. [59], which we described in Chapter 2. This approach does not consider missing values but has been shown to obtain high linkage quality for datasets without missing values. We first generated ABFs, and then sampled bits in these ABFs based on either setting all attribute weights equal to 1 (named EW) or by using the Fellegi and Sunter [67] weight calculation (named FS). The sets of sampled bits are then concatenated and permuted to generate one RBF per record.

For both the baselines and our proposed approach, we generated ABFs of length l = 1,000 bits for each attribute using the length of q-gram $q = 2$ and random hashing [129, 157] (as we described in Section 2.3.1), and set the number of hash functions to optimal, such that around half of all bits in the ABFs are set to 1 [181]. As with the RBF baseline, we set the attribute weights, $\mathbf{w}$, of our approach, to be either all equal (EW) or to weights calculated by using the approach by Fellegi and Sunter (FS) [67]. We compared the baseline approaches with both of our iterative and batch methods, where we applied all three grouping methods (*upper*, *lower*, and *both*). We also ran experiments without any grouping (named *No group*) to evaluate the impact of grouping on the linkage quality. For our grouping methods, we set the parameters $d_t = [1, 2, 3]$ and $w_t = [1, 2, 3]$ for the EW, and $w_t = [7, 14, 21]$ for the FS weighting approach based on a series of set-up experiments to find suitable weights.

5.8.3 Linkage Quality Results

In terms of linkage quality, we calculated precision (Equation 2.13) and recall (Equation 2.14) [83], as we described in Section 2.5.1. We show precision and recall at different similarity thresholds s_t ranging from 0.5 to 1.0 in 0.1 steps. Figure 5.6 shows the precision-recall plots for the iterative and batch linkage methods for different thresholds of difference between Hamming weights, d_t. As can be seen, our approaches generally provide higher recall when d_t is increased because more missingness patterns (and their corresponded BFs) are grouped into a partition.

With the g = *upper*, patterns with more missing attribute values are grouped into partition(s) with less missing attribute values. This results in more BF pairs classified as non-matches. This is because they have lower numbers of 1-bits, thus, increasing the number of missed matches. With g = *lower*, patterns with less missing values are grouped into partitions that contain more missing values in their records. The corresponding BFs are constructed using less attributes leading to an increase in the number of false matches when similarity thresholds are low, especially with datasets that contain a missing layer. This results in g = *lower* to provide more false matches (but also more true matches) compared to g = *upper*. As can be seen in the g = *lower* plots in Figure 5.6, recall values are generally higher than with g = *upper*. However, the best results are obtained by g = *both*, where each pattern is grouped with its neighbouring upper and lower patterns.

Comparing the iterative and batch linkage methods, g = *upper* provides similar results. However, iterative linkage slightly outperforms batch linkage when the two other grouping methods are used. This is because with the batch linkage method, matched pairs of BFs cannot be removed after each iteration. The best matches selection process employed by the DOs leads to a slightly lower recall because it generally identifies only the best one-to-one matching BF pairs. With the iterative linkage method, one-to-many matches can occur within the same partition because the LU classifies matches rather than the DOs.

As can also be seen in Figure 5.6, rather unexpectedly recall becomes lower for g = *lower* for lower similarity thresholds (while one would expect the recall to increase with lower thresholds). This is because true matching record pairs that contain missing values will occur in partitions that are handled in later iterations. However, because less attributes are compared in g = *lower*, more false matches are classified in the earlier iterations, and once such records are matched they are removed from later iterations (thus, not compared to their true matching record in the other database).

In Figure 5.7, we compare our approach to the RBF [59] and k-NN [30] baselines, as well as our approach without any grouping methods applied (where only common patterns are compared, each as its own partition). As can be seen from this figure, the influence of the weighting approach, of either equal (EW) or based on the Fellegi and Sunter [67] (FS) weight calculation, is small, FS outperforms EW in only some experiments (obtaining higher precision and slightly reduced recall). Our approach without grouping (*No group*) leads to low recall results, which shows

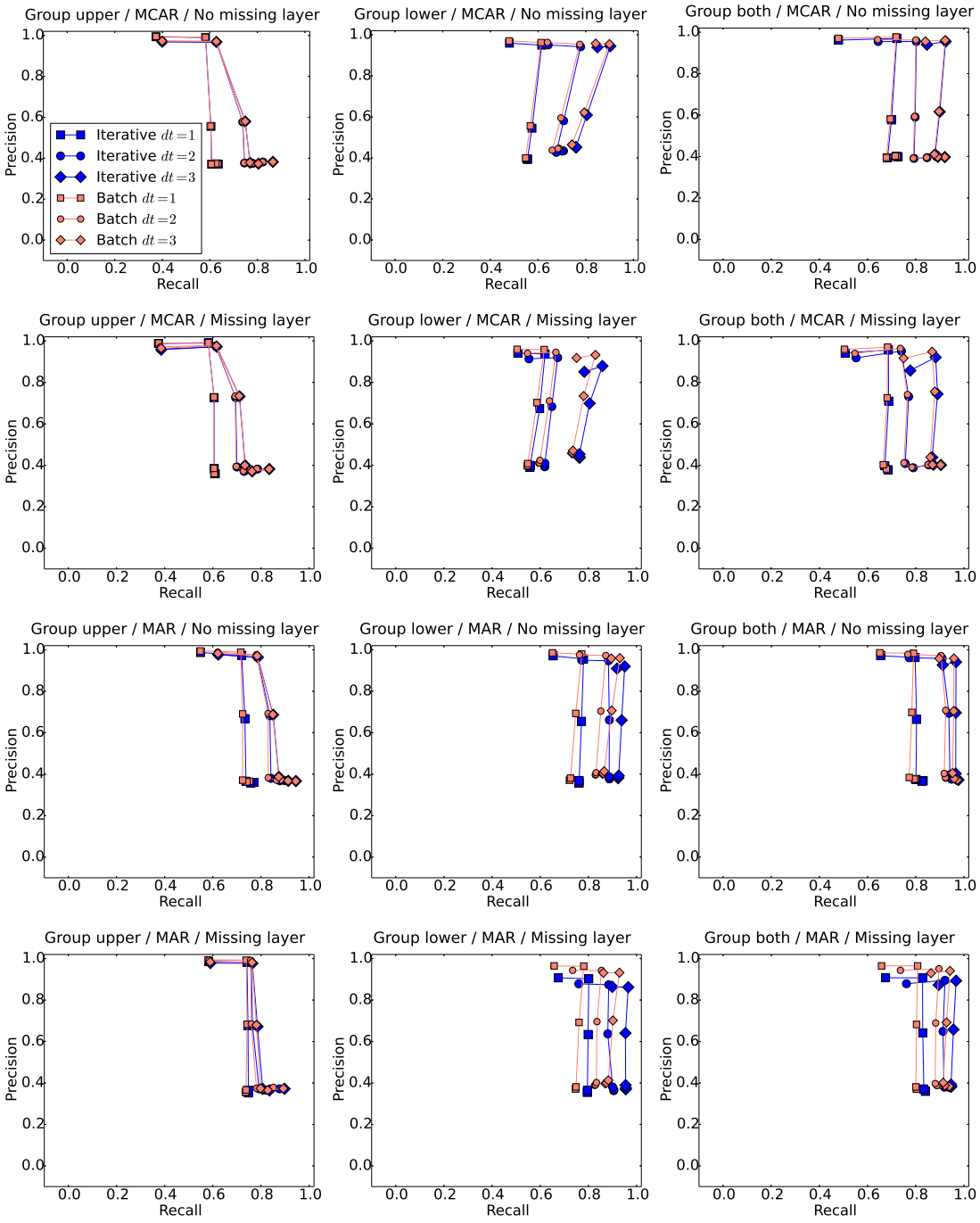

Figure 5.6: Precision-recall plots of the iterative and batch linkage methods for different thresholds d_t, and different grouping methods (column-wise). The plots of MCAR dataset are shown in the first and second rows, while the plots of MAR dataset are shown in the third and last rows, both with and without missing layers of patterns.

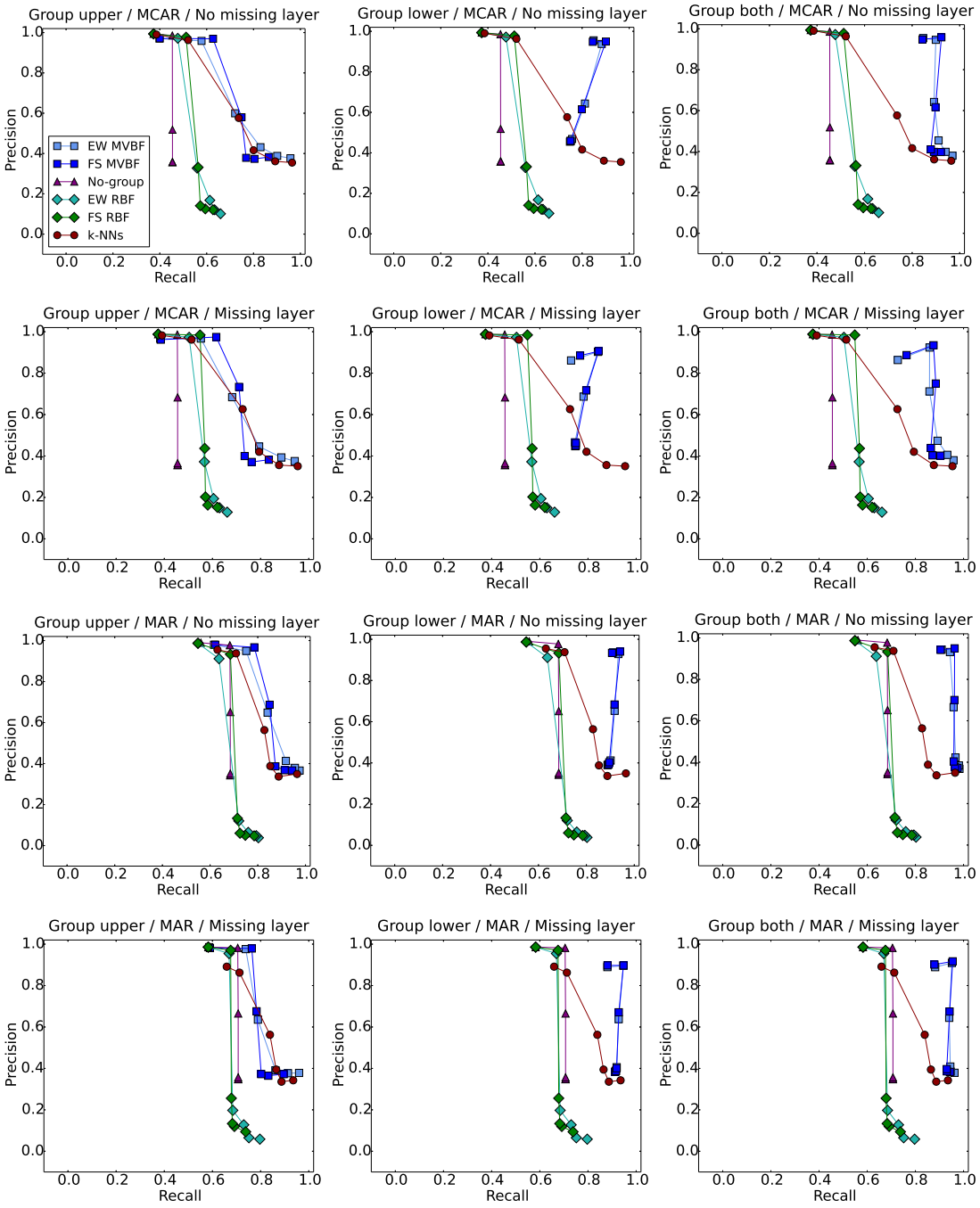

Figure 5.7: Precision-recall plots of our approach (named MVBF) as averaged over the iterative and batch linkage methods, compared to the RBF, k-NN, and our approach without grouping applied. Results for different grouping methods are shown in column-wise, and results for the MCAR dataset are shown in the first and second rows, while the results for the MAR dataset are shown in the third and last rows. EW refers to equal weights for all linked attributes and FS refers to Fellegi-Sunter [67] weight calculation.

the importance of grouping patterns. Without grouping patterns into partitions, true matching record pairs that have different missingness patterns in each of their two records will not be compared because they are inserted into different partitions.

The k-NN approach [30] shows both lower precision and recall results compared to our approach. It results in higher numbers of false matches (lower precision) because the imputation of missing attribute values based on the k-nearest neighbours for a record with missing values is not always correct. Thus, the k-NN approach is highly sensitive with regard to this imputation process. This is especially the case for lower similarity thresholds where it is more likely that less similar neighbouring records are being selected resulting in incorrectly matched record pairs. As can be also seen, k-NN performs better on the MAR compared to the MCAR datasets, which is because MAR results in groups of records that have more similar neighbouring records with the same missingness pattern. Therefore, it seems that the k-NN approach performs best on databases that contain similar groups of records, such as families or households in census data. For databases that do not contain such groups, the k-NN approach might not be applicable in realistic scenarios.

The RBF baseline [59] results in more missed matches for these datasets that contain missing values. This is because the generated RBFs contain 0-bits sampled from those ABFs that correspond to attributes with missing values. This results in reduced similarities between record pairs, which potentially increases the number of missed true matches and leads to a lower recall. On the other hand, with low similarity thresholds, more RBF pairs are classified as matches because the non-missing attributes are still being compared without an adjustment of the weights assigned to attributes. This results in higher similarities, thus, more false matches.

Furthermore, as can be seen in both Figures 5.6 and 5.7, the linkage results obtained with the MAR datasets are generally better than with the MCAR datasets. This is because records in the MAR datasets have been generated such that they have the same missingness patterns if they have the same zip code values. This indicates that our grouping based approach can make use of MAR patterns leading to improved linkage quality results.

5.8.4 Scalability Results

In Table 5.5, we show the average numbers of records (averaged of different missingness patterns and data corruptions) that each DO generates and sends to the LU for the MCAR datasets generated using the batch linkage method. The number of records increases as the threshold, d_t, is increased because this means a record with a certain missingness pattern will be inserted into a larger number of partitions.

We illustrate the average runtimes (averaged between all datasets) required by a DO and the LU for the different linkage approaches in Table 5.6. As expected, the iterative linkage method consumes less runtime compared to the batch linkage method because after each iteration record pairs classified as matches are removed, and therefore less records are included in the following partitions. On the other

Table 5.5: Numbers of records (averaged on different missingness patterns and data corruptions) of the MCAR datasets that generated by each DO ($\mathbf{D}_A / \mathbf{D}_B$) in the batch linkage method. With *No group*, the number is equivalent to the average number of records that corresponds to common patterns which is 45,568 records.

Grouping method	$d_t = 1$	$d_t = 2$	$d_t = 3$
$g = upper$	124,500 / 116,500	207,250 / 207,750	225,250 / 227,000
$g = lower$	272,000 / 280,000	417,000 / 411,750	531,750 / 524,750
$g = both$	309,750 / 309,750	500,250 / 499,000	633,000 / 631,250

Table 5.6: Average runtimes (in seconds) and standard deviation ($\pm$) used by a DO and the LU for linking datasets using different approaches.

Linkage approach		DO	LU
RBF		1101.38 $\pm$ 76.04	1.87 $\pm$ 0.64
k-NN		7233.35 $\pm$ 386.97	1655.67 $\pm$ 477.55
Iterative, $g = upper$	$d_t = 1$	616.37 $\pm$ 29.62	0.80 $\pm$ 0.10
	$d_t = 2$	977.95 $\pm$ 173.99	1.24 $\pm$ 0.21
	$d_t = 3$	1108.78 $\pm$ 158.71	1.31 $\pm$ 0.46
Iterative, $g = lower$	$d_t = 1$	808.90 $\pm$ 82.60	0.87 $\pm$ 0.13
	$d_t = 2$	1328.88 $\pm$ 247.15	1.23 $\pm$ 0.19
	$d_t = 3$	1607.77 $\pm$ 403.10	1.37 $\pm$ 0.34
Iterative, $g = both$	$d_t = 1$	857.66 $\pm$ 74.32	0.91 $\pm$ 0.08
	$d_t = 2$	1465.13 $\pm$ 249.83	1.34 $\pm$ 0.21
	$d_t = 3$	1793.72 $\pm$ 317.51	1.42 $\pm$ 0.21
Batch, $g = upper$	$d_t = 1$	4110.44 $\pm$ 495.07	0.77 $\pm$ 0.12
	$d_t = 2$	8225.33 $\pm$ 2073.46	1.35 $\pm$ 0.25
	$d_t = 3$	10029.39 $\pm$ 3013.89	1.62 $\pm$ 0.63
Batch, $g = lower$	$d_t = 1$	10374.71 $\pm$ 2153.90	4.08 $\pm$ 1.60
	$d_t = 2$	15028.00 $\pm$ 4870.79	6.07 $\pm$ 3.56
	$d_t = 3$	21502.24 $\pm$ 6250.22	9.45 $\pm$ 5.19
Batch, $g = both$	$d_t = 1$	11255.37 $\pm$ 1732.49	3.77 $\pm$ 1.37
	$d_t = 2$	18343.72 $\pm$ 6610.39	9.10 $\pm$ 6.29
	$d_t = 3$	24915.95 $\pm$ 8752.60	10.30 $\pm$ 5.31

hand, with the batch method, the whole database is sent to the LU once only, where each record likely occurs in multiple partitions.

As can be seen from Table 5.6, g = *lower* has longer runtimes than g = *upper* for the iterative linkage method. This is because the first partition of g = *upper* contains more BFs compared to g = *lower*, thus, more BFs will be matched in the first iteration. As a result, these BFs will not be included in later partitions. With g = *lower*, a smaller partition is processed in the first iteration and BFs with different missingness patterns will be included in later iterations, and therefore these BFs are only matched later. This results in overall longer runtimes for g = *lower*.

With g = *both*, the runtimes depend upon how many records in the first partition, which is generally the largest, are classified as matches and are therefore removed. However, given g = *both* has shown to achieve the best linkage results, as we discussed in Section 5.8.3, we recommend to use this grouping method over the other grouping methods (for both the iterative and batch linkage methods).

For the iterative linkage method, no clear increase in runtimes with larger d_t is observable. This is due to the iterative processing of partitions, where with larger values of d_t more BFs are part of the first partition. These BFs can be classified as matches and removed, resulting in smaller partitions from the second iteration onward. Besides the grouping method and threshold d_t, the actual similarities between BFs also affect runtimes. As can be seen from Table 5.6, the LU requires longer runtimes with g = *upper* than g = *lower* because it compares more BF pairs in the first partition, where the pairs that were not classified as matches are again included in the following partitions. Even for $d_t = 1$, the LU requires longer runtimes because smaller partitions are generated, thus, less BF pairs can be matched in the earlier iterations.

Comparing our approach with the baseline approaches, the RBF approach results in the shortest runtimes because each record is only encoded into one BF and then sent to the LU once for comparison. The k-NN approach results in substantially higher runtimes compared to the other approaches because of the similarity calculations it is conducting across corresponding (k = 10) nearest neighbouring records for each record that contains a missing value. The high computational efforts by the DOs are because they need to identify the k-nearest neighbours for each record with a missing attribute value, and then calculate the similarity estimate for that missing value. The LU requires more runtime because it needs to calculate the weight summation of the k neighbours for every missing attribute value (based on the 1-bit positions in the correspond ABFs in the other database), where these weights are then incorporated into the similarity calculations.

5.8.5 Privacy Results

To evaluate privacy, we used the cryptanalysis attack proposed by Christen et al. [39], as we described in Section 2.3.3. This attack aligns frequent BFs and plaintext values in a publicly available global database **G** with the aim to re-identify the most frequent values encoded in these BFs. We assume the LU acts as the adversary and tries to re-identify the attribute values encoded in the BFs sent to it by the DOs [39, 109, 129]. We conducted this attack assuming the worst-case scenario of the LU gaining access to a database **D** of one DO, where $\mathbf{D} \equiv \mathbf{G}$, and trying to re-identify the values in **D** by using the BFs of the other DO. However, such an attack is unlikely in practice since the DOs do not send their own plaintext databases to any other party.

In Figure 5.8, we show the re-identification results using the cryptanalysis attack [39] applied to all evaluated approaches with both the MCAR and MAR datasets. Similar to Chapter 4, we assessed the re-identification accuracy of the attack by calculating the percentages of (1) correctly re-identify attribute values with one-to-one

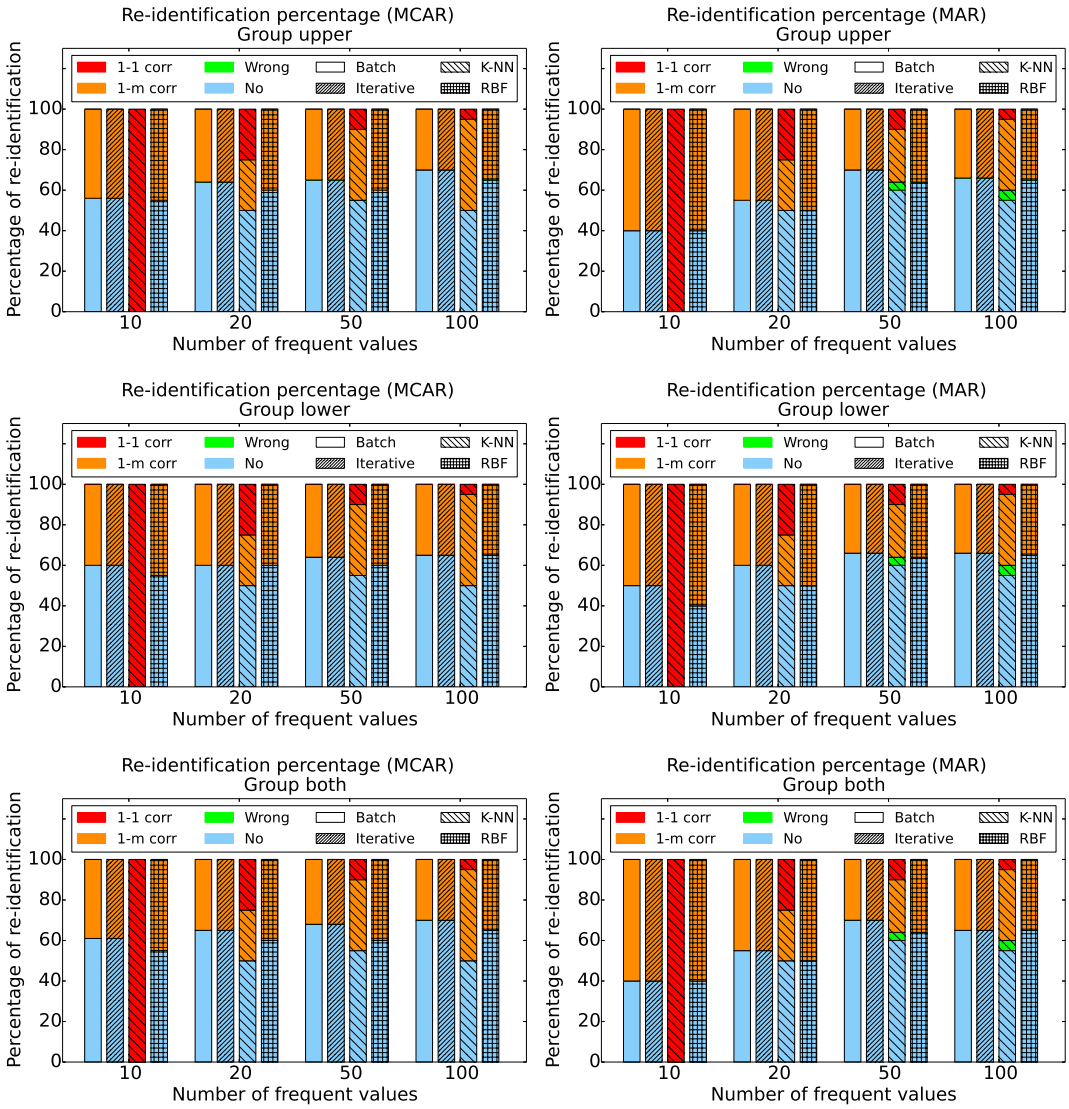

Figure 5.8: Re-identification results for the MCAR (left) and MAR (right) datasets, where g = *upper* (first row), g = *lower* (middle row), and g = *both* (last row) were applied. The bars show the percentages of BFs that were re-identified either correctly with only 1 (1-1 corr), several (1-m corr) values, wrong, or no re-identified.

matches; (2) correct guesses with one-to-many matches; (3) wrong guesses; and (4) no guesses, where we considered the accuracy of the attack on the 10, 20, 50, and 100 most frequent attribute values from the plaintext datasets, respectively [39].

As can be seen from Figure 5.8, due to the grouping of BFs with different missingness patterns into partitions, the attack cannot correctly identify any one-to-one matches between BFs and plaintext values in our approach. The attack is able to identify a larger number of one-to-many matches between BFs and plaintext records with the MAR datasets compared to MCAR datasets. This is because records with

similar missingness patterns can generate similar BFs which allows the attack to match them to plaintext values. However, since the number of records that share the same missingness pattern is large, the attack cannot identify any one-to-one matches between BFs and plaintext values.

As the results in Figure 5.8 show, the k-NN baseline provides the weakest privacy protection (the attack achieved the best re-identification results) for both MCAR and MAR datasets with one-to-one correct re-identifications for the 10 most frequent values. Because the k-NN approach is based on ABFs, the attack can successfully align frequent plaintext values to their corresponding ABFs which allows correct re-identifications. These results highlight the weaknesses of ABFs, and therefore the k-NN approach which relies on this encoding method.

In contrast to k-NN, RBF encoding (which we use in our approach) provides better privacy due to the weight distribution and random bit sampling process it uses. The re-identification results show that our approach provides stronger privacy compared to the baselines which makes it more applicable in realistic PPRL scenarios.

5.9 Chapter Summary

In this chapter, we have presented our approach to consider missing values of different missing data types (as we described in Section 5.2) in PPRL using record-level Bloom filter (RBF) encoding. Our approach is based on a lattice of missingness patterns (as we described in Section 5.3) from which partitions of BFs are generated. Each record can be inserted into multiple partitions using different grouping methods, which ensures records with different missingness patterns will be appropriately compared, as we discussed in Sections 5.5 and 5.6.

We conducted an extensive analysis in Section 5.7, and an experimental evaluation of our approach compared to baselines in Section 5.8 on a real-world database with different missingness patterns and numbers of missing values. Our results show that our approach can achieve high linkage quality while providing privacy against cryptanalysis attacks [39], where it substantially outperforms the RBF approach proposed by Durham et al. [59] and the *k*-nearest neighbour (k-NN) based PPRL approach for missing values proposed by Chi et al. [30]. In the following chapter, we present our accurate and efficient PPRL techniques for string matching.

Chapter 6

Accurate and Efficient Privacy-Preserving String Matching

In Chapters 4 and 5 we proposed a hardening technique for Bloom filter encoding [155] and a privacy-preserving record linkage (PPRL) technique for linking databases that contain missing values, respectively. While these approaches provide high linkage quality, none of them provides accurate string matching results (a pair of strings and their corresponding encoded values have the same similarity value). Therefore, in this chapter, we discuss our new proposed PPRL approaches for accurate string matching based on the longest common sub-string (LCS) between strings.

In Section 6.1, we provide an introduction to our proposed approaches. We then give an overview of our protocols in Section 6.2, where we propose two approaches, which are (1) a shifting hash encoded q-grams based approach, as we describe in Section 6.3, and (2) a bit array based approach, as we describe in Section 6.4. Based on the results of these two approaches, we then calculate the length of the LCS between compared strings, as we discuss in Section 6.5. In Section 6.6, we describe the scalability aspect of our two approaches, and in Section 6.7 we analyse our approaches in terms of linkage quality, scalability, and privacy. In Section 6.8, we discuss our experiment evaluation. Finally, in Section 6.9, we summarise this chapter.

6.1 Introduction

As we described in Chapters 1 and 3, some application domains, such as healthcare and financial service providers [38], require the matching sequences of characters between two strings (sub-string matching) to find the longest common sub-strings between records in two databases [79, 80]. To solve this problem, various techniques have been developed, as we illustrate in Table 6.1. As we described in Chapter 2, a widely used privacy-preserving record linkage (PPRL) technique that allows string comparison is based on converting strings into sets of q-grams (sub-strings of length q characters) and encoding these sets into Bloom filters (BFs) [155], where these BFs can be compared by using set-based similarity functions such as the Dice-coefficient [33]. It has been shown that BFs based PPRL is efficient and can achieve high linkage results compared to non PPRL approaches [146, 147].

Table 6.1: Existing privacy-preserving string matching techniques, where we show the complexity for encoding and matching one string. In this table, l is the length of a string, $|\Sigma|$ is the size of the alphabet Σ, n_h is the number of hash functions used, l_b is the length of a Bloom filter or bit array, t is the number of hash tables, q is the length of a sub-string (q-gram), and $|\mathbf{D}|$ is the size of a plaintext (string) database $\mathbf{D}$.

Method	Encoding complexity	Matching complexity	Application
Bloom filter [155]	$O(l \times n_h)$	$O(l_b)$	PPRL
Tabulation hashing [170]	$O(l \times t \times n_h)$	$O(l_b)$	PPRL
Damgård-Geisler-Krøigaard string matching [66]	$O(\Sigma^q)$	$O(\Sigma^q)$	PPRL
Burrows-Wheeler Transformation [167]	$O(l \times \sqrt{l \times \lvert\Sigma\rvert})$	$O(l^2 \times \lvert\Sigma\rvert)$	Genomics
Longest prefix match [125]	$O(l \times \lvert\mathbf{D}\rvert)$	$O(l)$	Genomics
Encrypted suffix tree [23]	$O(l \times l_b)$	$O(l \times l_b)$	Cloud
Bloom filter tree [14]	$O(l^2 \times n_h)$	$O(l \times \log\ l)$	Cloud
Secure pattern matching [25]	$O(l)$	$O(l)$	Cloud
Secure query sub-string [79]	$O(2l)$	$O(l)$	PPRL
Frequent q-grams matching [18]	$O(3l)$	$O(l_b)$	PPRL
Secure pattern matching [201]	$O(l \times \lvert\Sigma\rvert)$	$O(l^2)$	Cloud

Table 6.2: Example string pairs from a real US voter database [34] that have the same set of q-grams with $q = 2$ (bigrams), and therefore Jaccard or Dice-coefficient similarities of 1.0, while their correct edit distance similarities [33] are much lower [176].

Attribute	String 1	String 2	Bigram set	Edit distance similarity
Zip code	27828	28278	{27, 28, 78, 82}	0.600
First name	amira	ramir	{am, ir, mi, ra}	0.600
First name	geroge	roger	{er, ge, og, ro}	0.500
Last name	avera	raver	{av, er, ra, ve}	0.600
Last name	gering	ringer	{er, ge, in, ng, ri}	0.333

A related similar approach to BFs [155] based on tabulation hash (TMH) encoding was proposed by Smith [170], as we discussed in Section 3.1. The TMH approach applies min-hash locality sensitive hashing [19] and uses Jaccard similarity calculation for comparing bit arrays. However, set-based comparisons as used with the BFs and TMH techniques have drawbacks, including (1) the sequence of characters in a string is lost when the string value is converted into a q-gram set and (2) they only allow the calculation of overall similarity between two strings. Therefore, they are unable to identify the longest common sub-string (LCS) between strings in a pair.

In certain cases [46], as shown in Table 6.2, two different strings can result in the same q-gram set which would be encoded into the same bit pattern. This can lead to falsely matched record pairs because of too high similarities between different string values [33]. The number of two different strings with the same q-gram set increases if the size of the alphabet Σ (the set of unique characters in strings to be encoded) becomes smaller because less unique q-grams can be generated. Therefore,

Table 6.3: Notation and terminology used in this chapter.

$\mathbf{D}, \mathbf{D}_A, \mathbf{D}_B$	Database, database A, and database B to be encoded, respectively
q	Length of q-gram
$\mathbf{q}$	List of q-grams
α, β	Padding characters
Σ	Alphabet in $\mathbf{D}_A$ and $\mathbf{D}_B$ (our first approach) or alphabet in $\mathbf{D}_A$ and $\mathbf{D}_B$ that involves α and β (our second approach)
s	Secret salting value
$\mathcal{H}()$	One-way hash function
h	Hash encoded q-grams
r	Random number
lce, lcb, lcs	Length of the longest common elements, bits, and sub-string, respectively
$\|...\|$	Size or number of values in a database or list
$\|\|$	Strings or bit arrays concatenation
$+$	Lists concatenation
$\mathbf{h}$	List of hash encoded q-grams or shifted hash encoded q-grams
$\mathbf{h}[i:]$	Sub-list of the list $\mathbf{h}$, where $\mathbf{h}[i:] = [\mathbf{h}[i], \mathbf{h}[i+1], \ldots \mathbf{h}[\|\mathbf{h}\| - 1]]$ and $0 \leq i < \|\mathbf{h}\|$
$\mathbf{h}[:j]$	Sub-list of the list $\mathbf{h}$, where $\mathbf{h}[:j] = [\mathbf{h}[0], \mathbf{h}[1], \ldots, \mathbf{h}[j-1]]$ and $0 < j \leq \|\mathbf{h}\|$
$\mathbf{h}[i:j]$	Sub-list of the list $\mathbf{h}$, where $\mathbf{h}[i:j] = [\mathbf{h}[i], \mathbf{h}[i+1], \ldots, \mathbf{h}[j-1]]$, with $i < j$, where $0 \leq i < \|\mathbf{h}\|$ and $0 < j \leq \|\mathbf{h}\|$
$\mathbb{h}$	Rotated (shifted) list for comparison, where $\mathbb{h} = \mathbf{h}[i:] + \mathbf{h}[:i]$ and $0 \leq i < \|\mathbf{h}\|$, or concatenated list, where $\mathbb{h} = \mathbf{h} + \mathbf{h}$
$\mathbf{T}_\Sigma$	Table of bit array of all possible q-grams generated from Σ
$\mathbf{T}_R$	Table of random bit arrays
b_q, b'_q, b_f	Bit array of a q-gram, of all q-grams in $\mathbf{q}$, and of a string (final bit array), respectively
l_q, l_t	Length of a q-gram bit array and of a string bit array, respectively
l_s	Length of the longest q-gram list in $\mathbf{D}_A$ and $\mathbf{D}_B$
l_γ	Segment length for fast bit arrays comparison
l_{lcb}	Minimum required length of LCB
bkv	Blocking key value
t_q	Length of q-gram set permutations to be used to generate bkv
p_b	Bit percentage to be used for generating bkv
l_b	Length of bit array to be used as bkv
m	Minimum length of LCS
TMH	Tabulation hashing
DGK	Approximate secure string matching based on Damgård-Geisler-Krøigaard homomorphic encryption
LCS	Longest common sub-string

strings generated using only digits (alphabet of size $|\Sigma| = 10$), such as zip codes or telephone numbers, will more likely result in increased q-gram set similarities compared to strings that contain letters ($|\Sigma| = 26$), such as first and last names.

In this chapter, we present two PPRL approaches, where we encode each string based on its list of q-grams. In the first approach, we encode the q-grams in each list into hash values, while in the second approach, we encode the q-grams in each list into a bit array of fixed length to improve the privacy protection of q-grams. However, the second approach requires more runtime for the encoding and comparison tasks than the first approach, which results in a trade-off between the privacy and scalability of our two approaches.

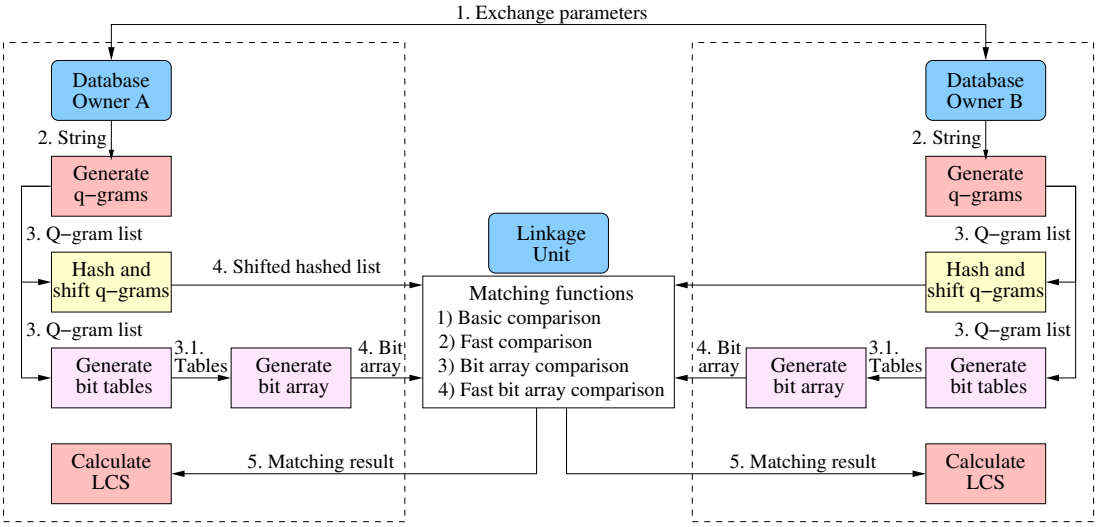

Figure 6.1: Overview of our proposed approaches. The blue boxes are the DOs and LU. The steps conducted by the DOs are shown in the dashed rectangles, where the red boxes are the steps common to both of our approaches. The yellow box shows the step of the shifted hash encoded q-gram based approach (described in Section 6.3), while the pink boxes show the steps of the bit array based approach (described in Section 6.4). The comparisons by the LU are shown in the white box.

In both approaches, we randomly shift the encoded q-grams in order to hide position information that could be exploited by an adversary. The encoded strings are then sent to a third party for identifying the length of the longest common encoded sub-string for each pair of encoded strings, which is then used to calculate the length of the LCS between the corresponding strings in a pair. We provide the notation and terminology that we use in this chapter in Table 6.3.

6.2 Protocol Overview

Our PPRL approaches involve three parties, the two database owners (DOs) and a linkage unit (LU), who we assume follow the honest-but-curious (HBC) adversary model [74, 81], as we described in Section 2.2.2. The DOs want to find the length of the longest common sub-string (LCS) between pairs of sensitive strings in their databases and they do not want to communicate with each other, except to agree on the parameters to be used. The LU is therefore used to compare the strings sent to it by the DOs. However, the DOs do not want to reveal the sensitive string values in their databases to any other party that participates in the protocol. Therefore, these strings need to be encoded before being sent to the LU, such that the LU cannot learn anything about them.

In some cases, the first characters in a string value can reveal some information. For example, the distribution of the first digits in numerical values can follow Benford's law [13], while the first letters in first and last names can follow Zipf's law [202]. This potentially allows an adversary to learn some of the encoded q-grams at the beginning of encodings by identifying the q-grams that occur frequently in a publicly available database [38]. Therefore, we propose two encoding approaches to prevent any q-grams from being re-identified.

As we illustrate in Figure 6.1, in our proposed approaches, the DOs first make an agreement on the parameter settings to be used in the protocols, which are:

- The padding characters, α and β (such as $\alpha =$ "\$" and $\beta =$ "#"), to be added to string values to avoid any incorrect length of LCS calculations where these characters must not be in the databases to be linked, as we describe in Sections 6.3.1 and 6.7.1.
- The length of q-gram, q, to be used for generating q-grams, as we describe in Section 6.3.1. The commonly used values for q are $q = 2, 3$ [38].
- The secret salting value, s, to be concatenated with the generated q-grams, as we describe in Sections 6.3.2 and 6.4.1. This value is used to avoid a dictionary attack by the LU [38].
- The alphabet Σ of all characters in the databases to be linked, as we describe in Sections 6.3.1 and 6.4.1.
- The one-way hash function [38], $\mathcal{H}()$ (such as SHA [154]), to be used for hashing q-grams in our shifted hash encoded q-gram based approach before sending the hashed q-grams from the DOs to the LU, as we describe in Section 6.3.2.
- The minimum length of the LCS, m, where $m \geq q$. This is used for selecting those string pairs that have a LCS of at least m, as we describe in Section 6.5. This parameter m can be set based on the length of the strings in the linked databases. For example, linking telephone numbers should require $m > 2$ because two telephone numbers should not be classified as matches when they have a common sub-string of only length 2.

The DOs generate q-grams from their unique sensitive string values. After that, the DOs individually encode all q-grams of each unique string in their databases and send these encoded q-grams to the LU. The LU compares the encoding of a pair of strings and returns the length of the longest common sequence of hash encoded q-grams (elements) in a pair, called the longest common elements (LCE), or the length of the longest common bit sequence between the bit arrays in a pair, called the longest common bits (LCB), back to the DOs. Finally, the DOs calculate the actual length of LCS based on the information received from the LU, as we describe in Section 6.5.

The first encoding approach (as we discuss in Section 6.3) improves the privacy of encoded q-grams by randomly shifting the position of the encoded q-grams in the generated encoded q-gram lists. The shifting of encoded q-grams hides their

actual positions in a string, which makes a position-based frequency analysis of q-grams more difficult and thereby prevents an adversary from identifying the string values that were encoded. This approach is useful for linking databases that require fast and accurate linkage results, such as linking the phone number of a criminal between databases to facilitate a fast response for police to take action.

In the second approach (as we describe in Section 6.4), we improve the privacy protection of q-grams by encoding q-grams into bit arrays. We hide the actual substring positions and the length of the encoded strings by adding random bit arrays at the beginning and end of the bit arrays that encode lists of q-grams. This ensures that the bit arrays of all string values in a database have the same length, and increases the difficulty for an adversary to identify the original string values that have been encoded into bit arrays. However, this approach uses more runtime than our first approach. Therefore, it can be useful for linking databases in application domains that require high degrees of privacy and accurate linkage results, but are less concerned about runtime, for example, linking credit card numbers between databases for financial crime investigations.

6.3 Shifting Hashed Q-grams based Approach

Our shifting hashed q-grams based approach consists of five steps: (1) parameter agreement (as we described in Section 6.2), (2) q-grams generation, (3) q-grams hash encoding and shifting encoded q-gram lists, (4) comparison of lists of shifting hash encoded q-gram lists by the LU, and (5) calculation of the length of the LCS by the DOs (as we describe in Section 6.5).

6.3.1 Q-grams Generation

Once the DOs have agreed on the parameters to be used in the protocol, they first add the agreed padding characters α and β to the beginning and end of their unique strings before these strings are being used to generate lists of q-grams. Let us assume the first DO has a database $\mathbf{D}_A$ and uses the padding character α, while the second DO has a database $\mathbf{D}_B$ and uses the padding character β, where $\alpha \neq \beta$. The padding characters are used to ensure the beginning and end of the compared strings are different. Due to the shifting process of encoded q-grams (as we describe in Section 6.3.2), without the padding characters, the length of LCS could be calculated incorrectly, as we discuss in Section 6.7.1.

Let us assume Σ is the alphabet of all characters in the databases $\mathbf{D}_A$ and $\mathbf{D}_B$, $\Sigma = \{a \in v : v \in \mathbf{D}_A \cup \mathbf{D}_B\}$, where a is a character in a string value v in the two databases. It needs to hold that $\alpha \notin \Sigma$ and $\beta \notin \Sigma$. Two strings $x \in \mathbf{D}_A$ and $y \in \mathbf{D}_B$ are padded as $x' = \alpha||x||\alpha$ and $y' = \beta||y||\beta$, respectively, where $||$ denotes the concatenation of strings. Once the DOs have added the padding characters to their strings, they independently generate the q-gram lists of padded strings in their databases. Each padded string in a database (let us use $\mathbf{D}_A$), consists of characters

Table 6.4: Example strings, and their corresponding q-grams and randomly shifted q-gram lists, where the patterns of where common characters occur in strings are shown in the first column (where b, m, and e represent the LCS occurring at the beginning, middle, or end of a string, respectively). The string pairs, with the minimum length of the LCS $m = 3$, are shown in the second column. Q-grams are generated using $q = 2$. The common sub-strings and the common elements are shown in bold. The random numbers used to shift each q-gram list are shown in the fourth column. *lce* refers to the length of longest common elements (LCE) between two lists in a pair.

Common patterns	String pairs	Q-gram lists (with padding characters)	Random shift	Shifted q-gram lists	*lce*	Calculated LCS, *lcs*
b-b	x = "**mary**" y = "**mars**"	[$m, **ma**, **ar**, ry, y$] [#$m, **ma**, **ar**, rs, s#]	$r_x = 0$ $r_y = 2$	[$m, **ma**, **ar**, ry, y$] [rs, s#, #$m, **ma**, **ar**]	2	$2+2-1=3$
b-m	x = "**mark**" y = "a**mary**"	[$m, **ma**, **ar**, rk, k$] [#a, am, **ma**, **ar**, ry, y#]	$r_x = 1$ $r_y = 2$	[k$, $m, **ma**, **ar**, rk] [ry, y#, #a, am, **ma**, **ar**]	2	$2+2-1=3$
b-e	x = "**mary**" y = "a**mar**"	[$m, **ma**, **ar**, ry, y$] [#a, am, **ma**, **ar**, r#]	$r_x = 2$ $r_y = 3$	[ry, y$, $m, **ma**, **ar**] [**ma**, **ar**, r#, #a, am]	2	$2+2-1=3$
m-m	x = "e**mary**" y = "a**mars**"	[$e, em, **ma**, **ar**, ry, y$] [#a, am, **ma**, **ar**, rs, s#]	$r_x = 2$ $r_y = 2$	[ry, y$, $e, em, **ma**, **ar**] [rs, s#, #a, am, **ma**, **ar**]	2	$2+2-1=3$
m-e	x = "e**mary**" y = "a**mar**"	[$e, em, **ma**, **ar**, ry, y$] [#a, am**ma**, **ar**, r#]	$r_x = 1$ $r_y = 3$	[y$, $e, em, **ma**, **ar**, ry] [**ma**, **ar**, r#, #a, am,]	2	$2+2-1=3$
e-e	x = "m**ark**" y = "c**ark**"	[$m, ma, **ar**, **rk**, k$] [#c, ca, **ar**, **rk**, k#]	$r_x = 2$ $r_y = 3$	[**rk**, k$, $m, ma, **ar**] [**ar**, **rk**, k#, #c, ca]	2	$2+2-1=3$
be-be	x = "**mary**" y = "**marry**"	[$m, **ma**, **ar**, **ry**, y$] [#$m, **ma**, **ar**, rr, **ry**, y#]	$r_x = 2$ $r_y = 2$	[**ry**, y$, $m, **ma**, **ar**] [**ry**, y#, #$m, **ma**, **ar**, rr]	2	$2+2-1=3$

and can be written as $x' = [a_0 \dots a_i \dots a_{n-1}]$, where $a_0 = a_{n-1} = \alpha$, $a_i \in \Sigma$ for $0 < i < (n-1)$, and $n = |x'|$. We define a q-gram as $q_i = a_i \dots a_{i+q-1}$, and a q-gram list as $\mathbf{q} = [q_0, \dots, q_{n-q}]$, where q is the agreed length of q-gram by the DOs. For example, we assume the agreed q-gram length is $q = 2$ and the string is $x =$ "*mary*". The DO adds the padding character $\alpha =$ "\$" to x, resulting in $x' =$ "\$*mary*\$". The DO then generates a q-gram list of the padded string x', resulting in $\mathbf{q} = [\$m, ma, ar, ry, y\$]$, as also shown in the third column in Table 6.4.

6.3.2 Hashing of Q-grams and Shifting Hash Encoded Q-gram Lists

Before the DOs send their databases to the LU, they individually encode the q-grams in each of their q-gram lists using the agreed hash function $\mathcal{H}()$. Each DO uses a salted hash encoding approach [38] to prevent a dictionary attack on the encoded q-grams. Given s is the secret salt value and $\mathcal{H}()$ is the hash function agreed by the DOs, each DO encodes each q-gram $q_i \in \mathbf{q}$, where $\mathbf{q}$ is the q-gram list and $0 \leq i < n$, with $n = |\mathbf{q}|$, into a hash value h_i as $h_i = \mathcal{H}(q_i||s)$. As a result, the hash encoded q-gram list of $\mathbf{q}$ becomes $\mathbf{h} = [h_0, h_1, ..., h_{n-1}]$.

Once the q-grams in each list $\mathbf{q}$ are hashed into a list $\mathbf{h}$, the DO generates a random number, r, for each of its hash encoded q-gram lists, $\mathbf{h}$, where $0 \leq r < |\mathbf{h}|$. The DO then shifts (rotates) the list $\mathbf{h}$ by r position(s), resulting in the shifted hash encoded q-gram list, $\mathbf{h}'$. Therefore, a given hash encoded value at position i, with

$0 \leq i < n$, is shifted to a new position $i' = ((i + r) \bmod n)$, also with $0 \leq i' < n$. This shifting process aims to hide the original positions of the hash encoded q-grams, and therefore the corresponding positions of characters in each string. Hence, the frequency distribution of shifted hash encoded q-grams will not follow Benford's [13] law because the first encoded q-grams are distributed to different positions in the shifted hash encoded q-gram lists, thereby preventing a position-based frequency attack on these encoded q-gram lists.

For example, we assume the string $x =$ "*mary*" has been padded and hashed into the hash encoded q-gram list $\mathbf{h}_x = [\mathcal{H}(\$m||s), \mathcal{H}(ma||s), \mathcal{H}(ar||s), \mathcal{H}(ry||s), \mathcal{H}(y\$||s)]$, and the generated random value for this string x is $r_x = 3$. Therefore, the hash encoded q-grams in $\mathbf{h}_x$ are shifted by three positions, resulting in the shifted list $\mathbf{h}'_x = [\mathcal{H}(ar||s), \mathcal{H}(ry||s), \mathcal{H}(y\$||s), \mathcal{H}(\$m||s), \mathcal{H}(ma||s)]$. We show examples of shifted q-gram lists in the fifth column in Table 6.4.

6.3.3 Comparison of Shifted Hash Encoded Q-gram Lists

To simplify notation, we now use $\mathbf{h}_x$ and $\mathbf{h}_y$ to represent the shifted hash encoded q-gram lists $\mathbf{h}'_x$ and $\mathbf{h}'_y$, respectively. For a pair of strings, the LU needs to find the length of the longest common (same) elements (LCE), *lce*, between the two lists $\mathbf{h}_x$ and $\mathbf{h}_y$. We propose a basic and a fast algorithm for the comparison process. The basic algorithm is a naive method to conduct the comparison by the LU which however requires longer runtimes for comparing pairs of shifted hash encoded q-gram lists. The fast algorithm is an alternative (more efficient) algorithm which allows for a faster comparison process.

6.3.3.1 Basic Shifted Hash Encoded Q-grams Comparison Algorithm

Because of the random shifting process performed by the DOs, the LU does not know the start and end positions of the shifted hash encoded q-grams in the lists $\mathbf{h}_x$ and $\mathbf{h}_y$ to be compared. Therefore, the LU needs to rotate the two lists to find the length of their LCE, *lce*, and then returns the *lce* to the DOs.

Algorithm 6.1 outlines the naive way for comparing two shifted hash encoded q-gram lists. In line 1, the LU first initialises the length of LCE, *lce*. It then checks if the two lists, $\mathbf{h}_x$ and $\mathbf{h}_y$, have any common elements in line 2. If common elements between the two lists exist, the LU then loops over each position, p_x, of the elements in the list $\mathbf{h}_x$ in line 3, where the element in $\mathbf{h}_x$ at position p_x is denoted as $\mathbf{h}_x[p_x]$.

In line 4, the LU finds all positions where the element $\mathbf{h}_x[p_x]$ occurs in the list $\mathbf{h}_y$ by using the function *getPosMatch*(). This function returns a list of positions, $\mathbf{p}_y$, where this list is empty if $\mathbf{h}_x[p_x]$ does not occur in $\mathbf{h}_y$. In line 5, the LU then generates the shifted (rotated) list, $\mathbb{h}_x$, with p_x as a starting position (first position) by concatenating the two sub-lists $\mathbf{h}_x[p_x{:}]$ and $\mathbf{h}_x[{:}p_x]$ into one rotated list $\mathbb{h}_x$. The LU then loops over each position, p_y, in the list of positions $\mathbf{p}_y$ in line 6, and in line 7 generates the rotated list, $\mathbb{h}_y$, with p_y as a starting position (first position), similar

Algorithm 6.1: ***Basic shifted hash encoded q-grams comparison process by the LU***

Input:
- $\mathbf{h}_x$: Hashed and shifted q-gram list of string x
- $\mathbf{h}_y$: Hashed and shifted q-gram list of string y

Output:
- lce: LCE of the hashed q-gram list pair

1: $lce \leftarrow 0$ // Initialise the length of LCE
2: **if** $set(\mathbf{h}_x) \cap set(\mathbf{h}_y) \neq \emptyset$ **do**: // Check if common hashed elements exist
3: **for** $0 \leq p_x < |\mathbf{h}_x|$ **do**: // Loop over each position in the list $\mathbf{h}_x$
4: $\mathbf{p}_y \leftarrow getPosMatch(\mathbf{h}_x[p_x], \mathbf{h}_y)$ // Get list of positions in $\mathbf{h}_y$ where the element $\mathbf{h}_x[p_x]$ occurs
5: $\mathbb{h}_x \leftarrow \mathbf{h}_x[p_x:] + \mathbf{h}_x[:p_x]$ // Get the current shifted (rotated) list of the list $\mathbf{h}_x$
6: **for** $p_y \in \mathbf{p}_y$ **do**: // Loop over each position in the list of positions $\mathbf{p}_y$
7: $\mathbb{h}_y \leftarrow \mathbf{h}_y[p_y:] + \mathbf{h}_x[:p_y]$ // Get the current rotated list of the list $\mathbf{h}_y$
8: $k \leftarrow 0$ // Initialise index and common count k
9: **while** $(k < min(|\mathbf{h}_x|, |\mathbf{h}_y|))$ **and** $(\mathbb{h}_x[k] = \mathbb{h}_y[k])$ **do**:
10: $k \leftarrow k + 1$ // Increment k if a common element occurs between $\mathbb{h}_x$ and $\mathbb{h}_y$
11: $lce \leftarrow max(lce, k)$ // Keep the length of the maximum length of LCE
12: return lce // Send the length of LCE to the DOs

to the list $\mathbb{h}_x$ generated in line 5. The common element at positions p_x and p_y now becomes the first element of the two rotated lists $\mathbb{h}_x$ and $\mathbb{h}_y$, respectively.

In line 8, the LU initialises an index k as a count of the number of common elements between $\mathbb{h}_x$ and $\mathbb{h}_y$. The LU loops over the rotated list that has the shorter length with the condition that the elements in the two rotated lists at position k are common (are the same), and then increases k by one in lines 9 and 10. The LU keeps the maximum k identified over all iterations of rotated lists in line 11. The LU then repeats the steps in lines 3 to 11 until the element $\mathbf{h}_x[|\mathbf{h}_x| - 1]$ becomes the first element in the rotated list $\mathbb{h}_x$. Finally, the LU returns the length of LCE, lce, to the DOs in line 12.

6.3.3.2 Fast Shifted Hash Encoded Q-grams Comparison Algorithm

Because the basic shifted hash encoded q-grams comparison algorithm (as we described above) has a high computation complexity, as we discuss in Section 6.7.2, in this fast comparison algorithm, we reduce the complexity of the comparison process by using the list concatenation technique. We use this technique because (1) the concatenated list contains the actual sequence of consecutive elements in the hash encoded q-gram list before it has been shifted, and (2) even after concatenation, the actual positions of the hash encoded q-grams are not being revealed to the LU, as we discuss in Section 6.7.3.

Let us use the q-gram list $\mathbf{q} = [\$m, ma, ar, ry, y\$]$ as an example. With the random number $r = 1$, the shifted q-gram list becomes $\mathbf{q}' = [y\$, \$m, ma, ar, ry]$. We concatenate the list $\mathbf{q}'$ with itself to generate a concatenated list $\mathbb{q}' = [y\$, \mathbf{\$m}, \mathbf{ma}, \mathbf{ar}, \mathbf{ry}, \mathbf{y\$}, \$m, ma, ar, ry]$. As can be seen from this example, the concatenated list $\mathbb{q}'$ does contain the actual sequence of consecutive elements of the q-gram list $\mathbf{q}$ (shown in bold).

Algorithm 6.2: ***Fast shifted hash encoded q-grams comparison process by the LU***

Input:
- $\mathbf{h}_x$: Shifted hashed q-gram list of string x
- $\mathbf{h}_y$: Shifted hashed q-gram list of string y

Output:
- lce: The LCE length of the pair of hashed q-gram lists

1: $lce \leftarrow 0$ // Initialise the length of LCE
2: $\mathbf{c} \leftarrow set(\mathbf{h}_x) \cap set(\mathbf{h}_y)$ // Get common hashed elements
3: **if** $|\mathbf{c}| \geq 1$ **do**: // Check if at least one common hashed element exists
4: $\mathbf{h}_s, \mathbf{h}_l \leftarrow \mathbf{h}_x, \mathbf{h}_y$ **if** $|\mathbf{h}_x| < |\mathbf{h}_y|$ **else** $\mathbf{h}_y, \mathbf{h}_x$ // Order the lists by their lengths
5: $\mathbb{h}_s \leftarrow \mathbf{h}_s + \mathbf{h}_s$ // Concatenate shorter list with a copy of itself
6: $\mathbb{h}_l \leftarrow \mathbf{h}_l + \mathbf{h}_l$ // Concatenate longer list with a copy of itself
7: $\mathbf{p}_s \leftarrow getConsecCommon(\mathbb{h}_s, \mathbf{c})$ // Get positions of consecutive common hashes in $\mathbb{h}_s$
8: **if** $|\mathbf{p}_s| = 0$ **do**: // Check if no consecutive hashed elements are found
9: $lce \leftarrow 1$ // There is no consecutive common hashed element
10: **else**: // If consecutive elements are found
11: **for** $p_s \in \mathbf{p}_s$ **do**: // Loop over each position in the list $\mathbf{p}_s$
12: $\mathbf{p}_l \leftarrow getPosMatch(\mathbb{h}_s[p_s], \mathbb{h}_l)$ // Get the list of positions in $\mathbb{h}_l$ where $\mathbb{h}_s[p_s]$ occurs
13: **for** $p_l \in \mathbf{p}_l$ **do**: // Loop over each position in the list of positions $\mathbf{p}_l$
14: $k \leftarrow 0$ // Initialise index and common count k
15: **while** $(k < min(|\mathbb{h}_s[p_s{:}p_s + |\mathbf{h}_s|]|, |\mathbb{h}_l[p_l{:}p_l + |\mathbf{h}_l|]|))$ **and** $(\mathbb{h}_s[p_s + k] = \mathbb{h}_l[p_l + k])$ **do**:
16: $k \leftarrow k + 1$ // Increment k by 1
17: $lce \leftarrow max(lce, k)$ // Keep the maximum length of LCE
18: return lce // Send the length of LCE back to the DOs

Algorithm 6.2 outlines the fast comparison between two shifted hash encoded q-gram lists. In line 1, the LU first initialises the length of the LCE, lce. It then finds the common elements between the two lists, $\mathbf{h}_x$ and $\mathbf{h}_y$, which it received from the DOs, and adds these common elements into the set $\mathbf{c}$ in line 2. If common elements occur between $\mathbf{h}_x$ and $\mathbf{h}_y$, the LU then orders the two lists by their lengths and assigns the shorter list to the list $\mathbf{h}_s$ and the longer list to the list $\mathbf{h}_l$ in lines 3 and 4. In lines 5 and 6, the LU concatenates each of $\mathbf{h}_s$ and $\mathbf{h}_l$ with a copy of itself, resulting in the two concatenated lists $\mathbb{h}_s$ and $\mathbb{h}_l$, respectively.

The LU then finds the list $\mathbf{p}_s$ of consecutive positions in $\mathbb{h}_s$ of the common elements in $\mathbf{c}$ by using the function $getConsecCommon()$ in line 7. For example, the first string pair in Table 6.4, the string $x =$ "*mary*" has the list $\mathbf{p}_s = [1, 2]$ which are the positions of q-grams *ma* and *ar*, respectively. In line 8, the LU checks if the list $\mathbf{p}_s$ is empty ($|\mathbf{p}_s| = 0$) which means there is no sequence of consecutive common elements occurring in both $\mathbf{h}_x$ and $\mathbf{h}_y$, and therefore the length of LCE is returned as $lce = 1$ in line 9. The lce is 1 because the set $\mathbf{c}$ is not empty as tested in line 3.

If the list $\mathbf{p}_s$ contains consecutive common elements, then the LU loops over each position p_s in the list $\mathbf{p}_s$ in lines 10 and 11. In line 12, the LU finds the list of positions, $\mathbf{p}_l$, in the longer concatenated list, $\mathbb{h}_l$, where the element at position p_s of the list $\mathbb{h}_s$ occurs by using the function $getPosMatch()$. The LU then loops over each position p_l in $\mathbf{p}_l$ and initialises the index k as the count of the number of common elements between the two concatenated lists, $\mathbb{h}_s$ and $\mathbb{h}_l$ in lines 13 and 14.

In line 15, the LU loops over the sub-list of the shorter concatenated list $\mathbb{h}_s$, where each sub-list is not longer than $|\mathbf{h}_s|$ and starting from position p_s, and the sub-list of the longer concatenated list $\mathbb{h}_l$, where each sub-list is not longer than the length $|\mathbf{h}_l|$ and starting from position p_l. If there are common elements in the two compared sub-lists of $\mathbb{h}_s$ and $\mathbb{h}_l$, then the LU increases the index k by one in line 16. The loop is terminated when the elements at the positions $p_s + k$ and $p_l + k$ are different. The LU then finds the maximum length of the LCE identified so far in line 17. The LU repeats the steps in lines 11 to 17 until there are no further positions to be compared. Finally, the LU returns the length of the LCE, *lce*, of the pair of shifted hash encoded q-gram lists to the DOs in line 18.

6.4 Bit Array Based Approach

While our approach based on shifted hash encoded q-grams prevents frequency attacks that are exploiting Benford's law [13], an adversary might still be able to identify the most frequent q-grams because these are encoded into hash values that will become the most frequent in the lists of hash encoded q-grams, $\mathbf{h}$. To prevent such attacks, we improve the degrees of privacy of the shifted hash encoded q-gram based approach by encoding each unique q-gram list $\mathbf{q}$ into a bit array. Each such bit array is padded at the beginning and end with random bits to ensure the bit arrays of all encoded strings have the same length even if the length of their q-gram lists differ. This approach prevents the LU from identifying sub-sequence of bits in a bit array that corresponds to each q-gram, as we discuss in Section 6.7.3.

Our bit array based approach consists of five steps: (1) parameter agreement (as we described in Section 6.2), (2) q-grams generation (as we described in Section 6.3.1), (3) generating bit arrays for strings, (4) bit arrays comparison by the LU, and (5) calculation of the length of the LCS by the DOs (as we describe in Section 6.5).

6.4.1 Generating Bit Arrays for Strings

To generate bit arrays for the strings in a database $\mathbf{D}$, each DO builds two tables of unique bit arrays. The first is the table $\mathbf{T}_\Sigma$ which contains one unique bit array for each possible q-gram that can be generated from the alphabet Σ, where Σ contains all characters that occur in the string values of the databases $\mathbf{D}_A$ and $\mathbf{D}_B$ being compared and the padding characters, α and β. The second table, $\mathbf{T}_R$, contains random bit arrays which will be used as padding of bit arrays to make the bit arrays of all q-gram lists the same length, where each DO needs to generate a unique table of such random bit arrays and these bit arrays must not be in $\mathbf{T}_\Sigma$ in order to prevent false matches.

Before building the tables $\mathbf{T}_\Sigma$ and $\mathbf{T}_R$, each DO first generates all unique q-grams that can be obtained from the alphabet Σ based on the q-gram length, q, where $\Sigma = \{a \in v : v \in \mathbf{D}_A \cup \mathbf{D}_B\} \cup \{\alpha, \beta\}$. The total number of possible q-grams we obtain is $|\Sigma|^q$. Based on the sizes of Σ, the DOs now need to calculate the q-gram bit array

Table 6.5: Minimum and estimated lengths of bit arrays, l_q^{min} and l_q^{appx} , for different sizes of Σ (includes the two padding characters, α and β) and q-gram lengths, q.

Alphabet Σ	$\lvert\Sigma\rvert$	l_q^{min}			l_q^{appx}		
		q=2	q=3	q=4	q=2	q=3	q=4
Genomics	4+2	7	10	12	10	12	16
Digits	10+2	9	13	16	12	16	20
Letters	26+2	12	17	21	14	20	24
Digits and letters	36+2	13	18	23	16	20	26

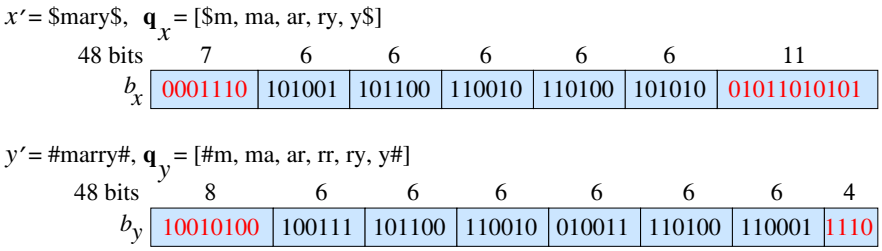

Figure 6.2: Two example bit arrays, where each is of length $l_t = 48$ bits, with different random bit arrays padded at the beginning and end (shown in red) [176].

length, l_q, to be used for generating the unique bit array for each possible q-gram. Because each DO needs to generate two tables of bit arrays ($\mathbf{T}_\Sigma$ and $\mathbf{T}_R$), where the random bit arrays $\mathbf{T}_R$ must be different between the two databases $\mathbf{D}_A$ and $\mathbf{D}_B$, the bit array length l_q must be large enough to allow at least $3|\Sigma|^q$ unique bit arrays to be generated. We can calculate a minimum length for l_q as $l_q^{min} = log_2(3|\Sigma|^q)$. This would however require every possible combination of bits to be generated, including $[0] \times l_q^{min}$ and $[1] \times l_q^{min}$. Such patterns could however reveal information as their frequencies of occurrence could be analysed by an adversary.

Therefore, to provide maximum entropy, which will make it more difficult for a frequency based attack to be performed, each DO randomly generates bit arrays where bits are set to 0 or 1 with equal probability [38, 165]. For a given bit array length l_q, the number of unique bit arrays that can be generated with half their bits set to 1 is $\binom{l_q}{l_q/2}$, where this number needs to be at least $3|\Sigma|^q$ in our case. We can calculate an approximation of this number based on Stirling's formula [44, 77] as:

$$l_q^{appx} = \left\lceil 2^{l_q} \Big/ \sqrt{\pi l_q/2} \right\rceil. \tag{6.1}$$

It holds that $l_q^{min} \leq l_q^{appx}$. Table 6.5 shows values for both l_q^{min} and l_q^{appx} for alphabets of different sizes and different q-gram lengths. In the following, and in our implementation, we assume that the value of l_q has been calculated based on Equation 6.1.

Once the DOs have calculated the q-gram bit array length, l_q, to be used, they engage in a secure protocol [166] to find the maximum length l_s that corresponds to

Algorithm 6.3: ***Bit arrays generation by a DO***

Input:
- **D**: Database as one list of q-grams per unique string value
- Σ: Alphabet
- q: Q-gram length
- l_q: Q-gram bit array length
- l_t: Final bit array length
- s: Common secret salt value
- s_d: Individual secret salt value

Output:
- **B**: Bit array inverted index

1: $\mathbf{B} \leftarrow \{\}$ // Initialise inverted index of bit arrays
2: $\mathbf{T}_\Sigma \leftarrow \{\}$ // Initialise table of all possible q-gram bit arrays
3: $\mathbf{T}_R \leftarrow \{\}$ // Initialise table of random bit arrays as a set
4: $\mathbf{Q}_\Sigma \leftarrow genQgramSet(\Sigma, q)$ // Generate set of all possible q-grams from Σ
5: **for** $q_\Sigma \in \mathbf{Q}_\Sigma$ **do**: // Loop over q-grams in $\mathbf{Q}_\Sigma$
6: $b_q \leftarrow genBitArr(\mathbf{T}_\Sigma, l_q, s, q_\Sigma)$ // Generate a unique bit array for each q-gram
7: $\mathbf{T}_\Sigma[q_\Sigma] \leftarrow b_q$ // Add q-gram bit array to table $\mathbf{T}_\Sigma$
8: **while** $|\mathbf{T}_R| \leq |\mathbf{T}_\Sigma|$ **do**: // Loop to generate table $\mathbf{T}_R$
9: $\mathbf{T}_T \leftarrow genRandBitArr(\mathbf{T}_\Sigma, l_q, s_d)$ // Generate temporal set of random bit arrays
10: $\mathbf{T}_C \leftarrow setIntersectDOs(\mathbf{T}_T)$ // Find common random bit arrays between DOs
11: $\mathbf{T}_T \leftarrow \mathbf{T}_T \setminus (\mathbf{T}_C \cup \mathbf{T}_R)$ // Remove common random bit arrays from $\mathbf{T}_T$
12: $\mathbf{T}_R.add(\mathbf{T}_T)$ Add random bit arrays to table $\mathbf{T}_R$
13: **for** $(sid, \mathbf{q}) \in \mathbf{D}$ **do**: // Loop over each q-gram list in the database
14: $b'_q \leftarrow [\,]$ // Initialise bit array for the current q-gram list
15: **for** $q_\Sigma \in \mathbf{q}$ **do**: // Loop over q-grams in the q-gram list **q**
16: $b'_q \leftarrow b'_q + \mathbf{T}_\Sigma[q_\Sigma]$ // Concatenate the bit arrays of all q-grams in **q**
17: $l_r \leftarrow l_t - |b'_q|$ // Calculate the number of random bits for padding
18: $b_f \leftarrow padRandBitArr(\mathbf{T}_R, b'_q, l_r)$ // Pad random bits at both beginning and end
19: $\mathbf{B}[sid] \leftarrow b_f$ // Add the final bit array for **q** to the inverted index **B**
20: return **B**

Function $genBitArr(\mathbf{T}_\Sigma, l_q, s, q_\Sigma)$:

21: $b_q \leftarrow [0] \times l_q$ // Initialise a bit array of 0-bits of length l_q
22: $setRandSeed(q_\Sigma || str(s))$ // Set PRNG seed value as q-gram concatenated with salt
23: **for** $0 < i \leq l_q$ **do**: // Loop over all positions of the bit array
24: $b_q[i] \leftarrow getRandChoice()$ // Choose a 0 or 1 bit value randomly with equal probability
25: **while** $b_q \in \mathbf{T}_\Sigma$ **do**: // Loop until the bit array b_q not exists in $\mathbf{T}_\Sigma$
26: $s \leftarrow s + 1$ // Increase salt value
27: $b_q \leftarrow genBitArr(\mathbf{T}_\Sigma, l_q, s, q_\Sigma)$ // Re-generate bit array b_q using increased salt value
28: return b_q

the longest q-gram list in their respective two databases, $\mathbf{D}_A$ and $\mathbf{D}_B$. To ensure all q-gram lists can be padded with random bits both at the beginning and end, the DOs add 2 to the length l_s, and then calculate the final bit array length as $l_t = (l_s + 2) \times l_q$. For example, as we illustrate in Figure 6.2, assume two padded strings are $x' =$ "$mary$" and $y' =$ "#$marry$#", and their corresponding q-gram lists generated as we described in Section 6.3.1 are $\mathbf{q}_x = [\$m, ma, ar, ry, y\$]$ and $\mathbf{q}_y = [\#m, ma, ar, rr, ry, y\#]$, respectively. As shown in Figure 6.2, the longest q-gram list is $\mathbf{q}_y$ with length $l_s = 6$, where each q-gram bit array is of length $l_q = 6$ (to simplify visualisation). Thus, the final bit array length is $l_t = (6 + 2) \times 6 = 48$ bits.

Algorithm 6.3 outlines the bit array generation by each DO. In line 1, each DO initialises the bit array inverted index, **B**, to be used for storing the generated bit

arrays of its unique strings. The DO then initialises the tables $\mathbf{T}_\Sigma$ (as an inverted index) and $\mathbf{T}_R$ (as a set), in lines 2 and 3, respectively. In line 4, the DO generates the set of all possible q-grams, $\mathbf{Q}_\Sigma$, based on the agreed alphabet, Σ, and q-gram length, q. In lines 5 to 7, the DO then loops over q-grams $q_\Sigma \in \mathbf{Q}_\Sigma$ to generate a bit array for each q_Σ by using the function $genBitArr()$. The details of this function are provided in lines 21 to 28.

In line 21, the function $genBitArr()$ first initialises a bit array of length l_q with only 0 bits. Then, the q-gram q_Σ is concatenated with the secret salt value, s, that was agreed by the two DOs. This concatenated value is used as the seed to a pseudo-random number generator (PRNG) [38]. With the same seed, a PRNG will generate the same sequence of random outputs, and therefore all DOs generate the same bit array for the same q-gram q_Σ. The loop in lines 23 and 24 will generate l_q such random bits, where the function $getRandChoice()$ returns a 0- or a 1-bit with equal probability. As a result, the bit array b_q should be filled with roughly 50% 1-bits. In line 25, the DO checks if b_q exists in $\mathbf{T}_\Sigma$ to ensure that each q-gram has a unique bit array. If the b_q does exist in $\mathbf{T}_\Sigma$, the DO increases the secret salt value and re-generates b_q for the q_Σ in lines 26 and 27. Because all DOs employ the same PRNG, they will generate the same bit arrays for the same set of all possible q-grams $\mathbf{Q}_\Sigma$. The function $genBitArr()$ returns b_q in line 28, where $b_q \notin \mathbf{T}_\Sigma$.

Back to the main program, where in line 7, each DO inserts the generated bit array b_q into the inverted index (table) $\mathbf{T}_\Sigma$, where the corresponding q-gram, q_Σ, is used as a key. Each DO repeats this process until one bit array b_q has been generated for every q-gram $q_\Sigma \in \mathbf{Q}_\Sigma$. In lines 8 to 12, the DO then generates its random bit array table, $\mathbf{T}_R$, by using the function $genRandBitArr()$. A temporary table, $\mathbf{T}_T$, is first generated in line 9, where the function $genRandBitArr()$ calls the function $genBitArr()$ (as described above) to generate each random bit array and each DO uses its individual secret salt value, s_d, as its seed. These individual secret salt values should result in different random bit arrays being generated by the DOs. However, to ensure no random bit array is generated by more than one DO, a secure set intersection protocol [53] is employed between the DOs in line 10. Any random bit array that has been generated by more than one DO will be returned by the $setIntersectDOs()$ function in the set $\mathbf{T}_C$, and only those random bit arrays unique to a DO as identified in line 11 are then added to its table (set) $\mathbf{T}_R$ in line 12. The DO repeats the steps in lines 8 to 12 until $|\mathbf{T}_R| = |\mathbf{T}_\Sigma|$.

Finally, each DO generates the final bit array, b_f, of length, l_t, for each string in its database. In line 13, each DO first loops over the q-gram lists $\mathbf{q}$ (generated as we described in Section 6.3.1) in its database, $\mathbf{D}$. For each $\mathbf{q}$, the DO initialises a q-gram bit array b'_q in line 14, and in line 15 the DO loops over each q-gram $q_\Sigma \in \mathbf{q}$. The DO then selects the q-gram bit array, b_q, that corresponds to q_Σ from $\mathbf{T}_\Sigma$ and concatenates the selected b_q to the bit array b'_q in line 16.

To ensure the generated bit arrays for all q-gram lists $\mathbf{q} \in \mathbf{D}$ are of the same length of l_t, in line 17, the DO calculates the number of random bits, l_r, that are required for padding based on the length of the generated bit array b'_q. In line 18,

Algorithm 6.4: ***Basic bit arrays comparison process by the LU***

Input:
- b_x: Bit array from the first DO
- b_y: Bit array from the second DO

Output:
- lcb: The number of bits in the longest common bit sequence

1: $lcb \leftarrow 0$ // Initialise the length of the LCB
2: $l_t \leftarrow |b_x|$ // Get the length of bit array, l_t
3: **for** $-(l_t - 1) \leq i \leq l_t - 1$ **do**: // Loop over each index position i
4: $x_s \leftarrow max(0, i)$ // Set start position of b_x to compare with b_y
5: $x_e \leftarrow min(l_t - 1, i + (l_t - 1))$ // Set end position of b_x to compare with b_y
6: $y_s \leftarrow max(-i, 0)$ // Set start position of b_y to compare with b_x
7: $y_e \leftarrow min(l_t - 1, l_t - (i + 1))$ // Set end position of b_y to compare with b_x
8: $b'_x \leftarrow b_x[x_s{:}x_e + 1]$ // Generate the bit segment b'_x of b_x for comparison
9: $b'_y \leftarrow b_y[y_s{:}y_e + 1]$ // Generate the bit segment b'_y of b_y for comparison
10: $c \leftarrow findCommon(b'_x, b'_y)$ // Find the length of the common bit array in the segment
11: $lcb \leftarrow max(lcb, c)$ // Keep the maximum length of so far lcb
12: return lcb

the DO uses the function $padRandBitArr()$ to generate the final bit array, b_f, where a random number of bits are padded both at the beginning and end of b'_q such that a total of l_r bits are padded, and where these random bits are sourced from the table of random bits arrays, $\mathbf{T}_R$. We illustrated this random padding process in Figure 6.2. In line 19, the DO then inserts the final bit array b_f into the bit array inverted index **B** which will be sent to the LU for comparison, as we describe next.

6.4.2 Comparison of Bit Arrays

For a pair of bit arrays, b_x and b_y, as the LU received from the DOs, the LU needs to find the length of the longest consecutive sequence of bits that are the same across the two bit arrays. We denote such a sequence as common bits, and the length of the longest such sequence as the length of longest common bits (LCB), lcb. We propose two algorithms for this comparison process. Similar to the comparison of shifted hash encoded q-grams described in Section 6.3.3, the first algorithm is a basic algorithm that follows a naive comparison method, while the second algorithm is a fast algorithm which substantially improves the runtime of the comparison process.

6.4.2.1 Basic Bit Arrays Comparison Algorithm

Algorithm 6.4 outlines the steps of the basic bit arrays comparison algorithm. In line 1, the LU first initialises the length of LCB to $lcb = 0$, and then obtains the length of the bit arrays b_x and b_y as $l_t = |b_x|$, where $|b_x| = |b_y|$ in line 2.

To compare the bit arrays b_x and b_y, the LU loops over each index position i in line 3, where $-(l_t - 1) \leq i \leq l_t - 1$. In lines 4 to 7, the LU then calculates the start (x_s and y_s) and end (x_e and y_e) positions for the two segments in b_x and b_y to be compared, based on the value of i, where $0 \leq x_s \leq x_e < l_t$ and $0 \leq y_s \leq y_e < l_t$.

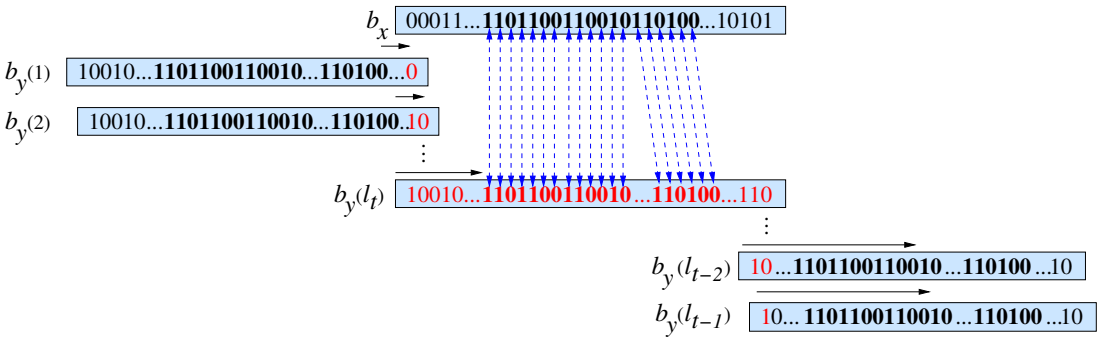

Figure 6.3: The basic bit arrays comparison of bit arrays b_x and b_y, where these bit arrays have two sequences of common bits, one 13 bits and the other 6 bits long. The numbers in brackets show the iteration number. The common bits are shown in bold and those bits that are compared between b_x and b_y are shown in red. The blue lines show the bits that are common between the two bit arrays b_x and b_y.

After that, the LU generates the corresponding two bit segments b'_x of b_x and b'_y of b_y in lines 8 and 9.

In line 10, the LU uses the function $findCommon()$ to find the length of the LCB by applying the XOR operation on b'_x and b'_y and identifying the length of the longest consecutive sequence of 0-bits, which represents the LCB between b'_x and b'_y. The LU keeps the longest length of the LCB so far identified in line 11. It repeats the steps in lines 3 to 11 for all index positions i. Finally, the LU returns the lcb to the DOs in line 12. Figure 6.3 shows an example of the basic bit arrays comparison algorithm between two bit arrays, b_x and b_y, where b_y is moved over b_x by one bit position per iteration.

6.4.2.2 Fast Bit Arrays Comparison Algorithm

In the fast bit arrays comparison algorithm, the DOs first calculate the minimum required length of LCB, l_{lcb} as $l_{lcb} = l_q \times (m - q + 1)$, where l_q is the q-gram bit array length, m is the agreed minimum length of LCS, and q is the agreed q-gram length. If two encoded strings share m consecutive characters, then they need to have a common bit sequence of at least the length of l_{lcb}.

Once the DOs have calculated the l_{lcb}, the DOs then agree on a segment length, l_γ, where $0 < l_\gamma \leq l_{lcb}$. We use the segment length l_γ because it allows the LU to compare bit arrays one segment after another, which reduces the runtime required by the LU. Furthermore, the LU will not be able to learn any information about the original bit arrays that represent individual q-grams, even if the segment length $l_\gamma = l_q$, because it does not know the length of the q-gram bit array l_q.

Algorithm 6.5 outlines the fast comparison processed by the LU and Figure 6.4 shows an example of this process on two bit arrays, b_x and b_y. As input, the LU receives b_x and b_y, and the segment length l_γ, from the DOs. In line 1, the LU

Algorithm 6.5: ***Fast bit arrays comparison process by the LU***

Input:
- b_x: Bit array from the first DO
- b_y: Bit array from the second DO
- l_γ: Segment length

Output:
- lcb: The number of bits in the longest common bit sequence

1: $lcb \leftarrow 0$ // Initialise the length of the LCB
2: $\mathbf{s}_y \leftarrow genSegment(b_y, l_\gamma)$ // Generate list of segments of bit array b_y
3: **for** $i \in (0, |\mathbf{s}_y| - 1)$ **do**: // Loop over each segment in the list of segments of $\mathbf{b}_y$
4: $\mathbf{p}_x \leftarrow getPosMatch(\mathbf{s}_y[i], b_x)$ // Get the list of positions in b_x that $\mathbf{s}_y[i]$ occurs
5: **for** $p_x \in \mathbf{p}_x$ **do**: Loop over all positions in the list $\mathbf{p}_x$
6: $c_l \leftarrow getCommonLeft(\mathbf{s}_y, b_x[:p_x])$ // Find common bits on the left of common segment
7: $c_r \leftarrow getCommonRight(\mathbf{s}_y, b_x[p_x:])$ // Find common bits on the right of common segment
8: $c \leftarrow c_l + |\mathbf{s}_y[i]| + c_r$ // Calculate length of common bit sequence
9: $lcb \leftarrow max(lcb, c)$ // Keep the maximum length of the LCB
10: return lcb

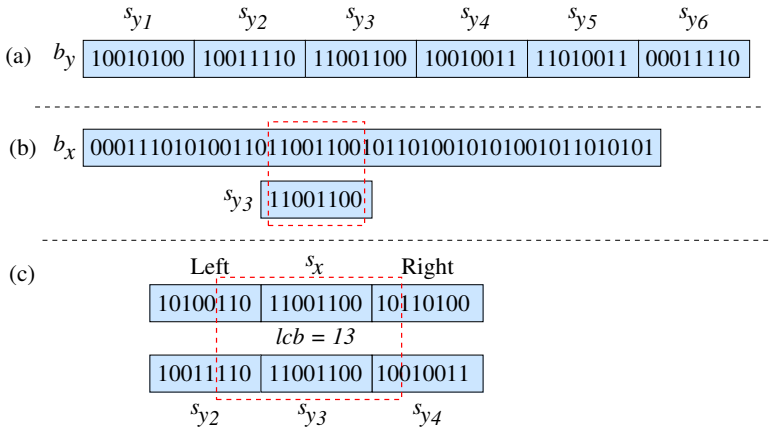

Figure 6.4: The fast bit arrays comparison between bit arrays b_x and b_y, where the bits in the red dashed boxes are the common bits, with the length of LCB, $lcb = 13$. Figure (a) illustrates the segmentation of b_y into segments of length $l_\gamma = 8$ as done by the LU, (b) shows how the LU finds the positions of the common bits, and (c) illustrates how the LU compares the segments to the left and right.

initialises the length of LCB, lcb, and in line 2 it generates a list of segments, $\mathbf{s}_y$, from b_y (as we illustrate in Figure 6.4 (a)) by using the function $genSegment()$, where each segment in $\mathbf{s}_y$ has a length of l_γ or less bits (last segment in the $\mathbf{s}_y$). In line 3, the LU then loops over the segments in the list $\mathbf{s}_y$. In line 4, for each segment, the LU finds the list of common positions, $\mathbf{p}_x$, in the bit array b_x where the segment $\mathbf{s}_y[i]$ occurs by using the function $getPosMatch()$, as we illustrate in Figure 6.4 (b).

Because each bit array contains both random and q-gram based bit arrays, a given segment can contain bits from both. For each position, p_x, in the position list $\mathbf{p}_x$, the LU therefore needs to check if there are sequences of common bits between b_x and

b_y both to the left and right of the common segment. This is because there can be further common bits in either direction, as we illustrate in Figure 6.4(c). The LU uses the functions $getCommonLeft()$ and $getCommonRight()$ in lines 6 and 7 to find the number of common bits on the left and right, respectively, between the current segment in b_y, $\mathbf{s}_y[i]$, and bits in b_x. In lines 8 and 9, the LU then calculates the current length of the common bit sequence and checks if it is a maximum length of the LCB, lcb. The LU repeats the steps in lines 3 to 9 for all segments in $\mathbf{s}_y$. Finally, the LU returns the lcb to the DOs in line 10.

6.5 LCS Length Calculation

As shown in Figure 6.1, in the last step of our approaches, the DOs calculate the length of the LCS, lcs, based on the matching results they received from the LU by using Equation 6.2. For the approach based on the shifted hash encoded q-grams (as we discussed in Section 6.3), the DOs calculate the lcs based on the length of the longest sequence of common elements, lce, while if they use the approach based on bit arrays (as we described in Section 6.4), they calculate the lcs based on the length of the longest sequence of common bits, lcb.

$$\begin{aligned} lcs &= lce + q - 1 \quad // \text{ For shifted hash encoded q-grams,} \\ lcs &= \lfloor lcb/l_q \rfloor + q - 1 \quad // \text{ For bit array encoding.} \end{aligned} \tag{6.2}$$

The DOs then only keep the string pairs that have an $lcs \geq m$, where m is the agreed minimum length of LCS. The last column in Table 6.4 on page 119 shows examples of the calculated LCS length based on the lce for different pairs of strings.

6.6 Scalability Aspect

In this section, we describe how we can scale our proposed approaches to large databases. The number of string pairs increases quadratically with the numbers of unique strings in the two databases being compared. We can improve the complexity of the comparison process by applying a privacy-preserving blocking technique [38, 58] to reduce the number of encoded string pairs that need to be compared by the LU.

We apply a q-gram based blocking approach [38] to generate blocks for each database, where each block is generated based on a permutation of q-grams in the q-gram list of each string. The DOs first agree on a secret salt value, s, a hash function, $\mathcal{H}()$, and the length of q-gram set permutations, t_q. In our approach, we calculate t_q based on the agreed minimum length of the LCS, m, and the q-gram length, q, as $t_q = m - q + 1$.

The DOs then independently generate the q-gram permutation lists of length t_q for each of their q-gram lists. Each DO concatenates the q-grams in each such list into

one string, q_s, which is used to generate a blocking key value, *bkv*, by concatenating it with the agreed secret salt value, s. This is followed by a hash encoding of this concatenated string, resulting in $bkv = \mathcal{H}(q_s \| s)$. Finally, all q-gram lists in a database that have the same *bkv* are inserted into the same block. Once the DOs have generated their blocks, they then send these blocks to the LU for conducting comparisons. The LU finds the common *bkv* between the received databases and only compares the encoded string pairs in blocks that have the same *bkv*.

For example, let us consider the DOs have agreed on $m = 3$ and $q = 2$, and therefore they calculate $t_q = 3 - 2 + 1 = 2$. We assume the two strings in the two databases, $x \in \mathbf{D}_A$ and $y \in \mathbf{D}_B$, are $x =$ "*mary*" and $y =$ "*mars*", with the q-gram lists $\mathbf{q}_x = [ma, ar, ry]$ and $\mathbf{q}_y = [ma, ar, rs]$, respectively. The DOs then individually generate the *bkv* of their strings as $\mathbf{bkv}_x = \{\mathcal{H}(maar\|s), \mathcal{H}(mary\|s), \mathcal{H}(arry\|s)\}$ and $\mathbf{bkv}_y = \{\mathcal{H}(maar\|s), \mathcal{H}(mars\|s), \mathcal{H}(arrs\|s)\}$. The encoding of strings x and y are inserted into every block with the $bkv_x \in \mathbf{bkv}_x$ and $bkv_y \in \mathbf{bkv}_y$, respectively. Once the DOs have sent their blocks to the LU, the LU can find the common $bkv = \mathcal{H}(maar\|s)$. Therefore, the two encoded strings x and y are being compared.

In the bit array based approach, we can generate blocks by applying Hamming based locality sensitivity hashing (HLSH) [58, 100]. In this approach, the LU receives two sets of bit arrays from the two DOs, it uses a set of hash functions to select certain bits. It then concatenates these bits into a bit array of fixed length, l_b, to be used as a *bkv*. The bit arrays that have the same *bkv* are then inserted into the same block. However, in our approach, the DOs individually generate blocks of bit arrays before sending them to the LU.

In our approach, the DOs first agree on the secret salt value, s, a hash function, $\mathcal{H}()$, and a bit percentage, p_b. They use p_b to calculate the length of a bit segment to be used for HLSH blocking as $l_b = (p_b \times l_{lcb})/100$, where we described l_{lcb} in Section 6.4.2. The DOs then generate segments of the selected q-gram bit arrays, b'_q, each of length of l_b. Each of these segments is then used to generate a *bkv* by concatenating them with the agreed secret salt value, s, followed by a hash encoding using the hash function $\mathcal{H}()$. The bit segments that have the same hash encoded value of *bkv* are inserted into the same block.

As the length of the b'_q is possibly not divisible by the l_b, the last segment might be shorter than l_b. To ensure every generated segment has the same length, we therefore extend any segment that is shorter than l_b by adding bits from the left segment. For example, if we assume $b'_q = 11010010$ (with $|b'_q| = 8$) and $l_b = 3$. The generated segments of this b'_q are 110, 100, and 10. Therefore, the last segment, 10, is extended with the last bit from the second segment, resulting in the last segment becoming 010. As a result, the set of *bkv* of this b'_q is $\mathbf{bkv} = \{\mathcal{H}(110\|s), \mathcal{H}(100\|s), \mathcal{H}(010\|s)\}$.

6.7 Analysis

In this section, we analyse our proposed approaches in terms of linkage quality, scalability, and privacy.

6.7.1 Linkage Quality Analysis

As mentioned in Section 6.3, we use different padding characters between databases to ensure that the calculated length of the LCS, lcs, is correct. Let us describe why the different padding characters are required by using q-gram lists without padding as an example. We assume the strings to be compared are $x =$ "*mary*" and $y =$ "*marry*". The correct LCS between these two strings is "*mar*" with the length $lcs = 3$. We assume the DOs have agreed on $q = 2$ and they use random numbers for shifting their q-gram lists $r_x = r_y = 1$ resulting in their shifted q-gram lists are $\mathbf{q}'_x = [\mathbf{ry}, \mathbf{ma}, \mathbf{ar}]$ and $\mathbf{q}'_y = [\mathbf{ry}, \mathbf{ma}, \mathbf{ar}, rr]$, respectively. When the LU compares these lists, it returns the $lce = 3$ (length of common q-grams that are shown in bold in $\mathbf{q}'_x$ and $\mathbf{q}'_y$) to the DOs. The DOs then use Equation 6.2 to calculate the lcs as $lcs = 3 + 2 - 1 = 4$. Therefore, the DOs obtain an incorrect result. As this example shows, our approach does not work when strings are not padded by different characters. Examples of correct LCS calculations are shown in Table 6.4.

Apart from the padding characters, to calculate an accurate lcs, the minimum length of the LCS, m, must be at least of length q, $m \geq q$. This is because when $m < q$, in the shifted hash encoded q-gram based approach, the LU cannot find the length of LCE, lce, between the two hash encoded q-grams lists. Let us use the two q-gram lists, $\mathbf{q}_x$ and $\mathbf{q}_y$, as an example. We assume $m = 3, q = 4$, and two padded strings are $x' =$ "$*mary*$" and $y' =$ "#*marry*#". The corresponding q-gram lists of x' and y' are $\mathbf{q}_x = [\$mar, mary, ary\$]$ and $\mathbf{q}_y = [\#mar, marr, arry, rry\#]$, respectively. There is no common q-gram between these lists, and therefore the DOs obtain the length of LCS, $lcs = 0$, while the actual LCS between this pair of x' and y' is "*mar*" with the $lcs = 3$. The same issue also occurs in the bit array based approach because each bit array is generated based on a list of q-grams.

In the bit array based approach, hash collisions [29], which occur when two or more q-grams are encoded into the same q-gram bit array, b_q, can affect the accuracy of string matching. The probability of a hash collision, P_b, that the bit can be set to 1 in this approach can be calculated by applying the conditional probability calculation [7] as:

$$P_b = \frac{l_q/2}{l_q} \times \frac{(l_q/2)-1}{l_q-1} \times \ ... \ \times \frac{1}{(l_q/2)+1}, \tag{6.3}$$

where l_q is the length of the q-gram bit array, and $l_q/2$ means 50% of l_q is set to 1.

When selecting the first bit position, there are $l_q/2$ out of l_q chances that a position is being selected to be set to 1 by two or more q-grams. The number of chances decreases by 1 once each position is selected. Finally, when selecting the last position, there remains 1 out of $l_q/2 + 1$ chances that a position can be selected. For example, assume we use $l_q = 6$, the probability that a hash collision can occur is $P_b = 3/6 \times 2/5 \times 1/4 = 0.05$ or 5% of l_q. Therefore, the probability that hash collisions can occur depends upon the length l_q. However, we avoid hash collisions in our bit array based approach by generating tables of unique q-gram bit arrays and unique random bit arrays, as we described in Section 6.4.1.

6.7.2 Scalability Analysis

In the shifted hash encoded q-gram based approach, each DO requires $O(l_h)$ for each step of the encoding process, where l_h is the length of hash encoded q-grams list corresponding to each string in its database. In the comparison process, the two shifted hash encoded q-gram lists, let us assume $\mathbf{h}_x$ and $\mathbf{h}_y$, are sent from the DOs to the LU, where these lists have the same length, l_h. In the basic comparison algorithm, the LU requires $O(l_h)$ for checking if $set(\mathbf{h}_x) \cap set(\mathbf{h}_y) \neq \emptyset$. For each common hash encoded q-gram $h_x \in \mathbf{h}_x$, the LU requires $O(l_h)$ to find the positions of h_x that occur in $\mathbf{h}_y$. It then requires $O(l_h^2)$ to find the length of the LCE, lce, between $\mathbf{h}_x$ starting from h_x (rotated $\mathbf{h}_x$) and every rotated list of $\mathbf{h}_y$. Therefore, overall the basic comparison algorithm requires $O(2l_h^2 + 2l_h)$.

In the fast comparison algorithm, the LU requires $O(l_h)$ for checking if $\mathbf{h}_x$ and $\mathbf{h}_y$ share elements. The LU then concatenates each list with itself and orders them, resulting in the shorter list $\mathbb{h}_s$ and longer list $\mathbb{h}_l$. The LU then requires $O(l_h)$ for finding the positions of consecutive common hash encoded q-grams in $\mathbb{h}_s$. For each $h_s \in \mathbb{h}_s$, it requires $O(l_h)$ to find the positions in h_s that occur in $\mathbb{h}_l$, and it requires $O(l_h)$ to find the lce between $\mathbb{h}_s$ starting from h_s and $\mathbb{h}_l$. Overall, the LU therefore requires $O(2l_h^2 + 2l_h)$. However, this is the worst-case which only occurs when $\mathbf{h}_x$ and $\mathbf{h}_y$ contain exactly the same encodings. Otherwise, the LU requires less than $O(2l_h^2)$ to find the lce between $\mathbb{h}_s$ and $\mathbb{h}_l$. Therefore, the fast comparison algorithm is faster experimentally than the basic comparison algorithm, as we will show in Section 6.8.4.

In the bit array based approach, each DO requires $O(|\Sigma|^q)$ to generate the bit array table of all possible q-grams, $\mathbf{T}_\Sigma$. To generate each random bit array, b_r, a DO checks if $b_r \notin \mathbf{T}_\Sigma \cup \mathbf{T}_R^A \cup \mathbf{T}_R^B$, where $\mathbf{T}_R^A$ and $\mathbf{T}_R^B$ are the random bit array tables of the two DOs. Each DO therefore requires a maximum $O(3|\Sigma|^q)$ to generate the random bit array table of size $|\mathbf{T}_\Sigma|$. To generate the final bit array, b_f, of each string, a DO requires $O(l_h)$ to concatenate the q-gram bit arrays, b_q, into a bit array b'_q, and $O(n_r)$ to pad each b'_q with random bit arrays, where n_r is the number of random bit arrays to be selected from $\mathbf{T}_R$.

For comparing two bit arrays, b_x and b_y, in the basic bit array comparison algorithm, the LU requires $O(l_t^2)$ to find the length of the LCB between b_x and b_y, where $l_t = |b_f|$. In the fast bit array comparison, the LU requires $O(l_t)$ to generate a list of segments, $\mathbf{s}_y$, from b_y, and for each $s_y \in \mathbf{s}_y$, it then requires $O(l_t)$ to find the positions in b_x where each s_y occurs. For each position p_x in b_x, the LU requires $O(l_t)$ to find the sequence of common bits that occur to the left and right of the bit at the position p_x in b_x. Therefore, the LU requires a total of $O(l_t + (|\mathbf{s}_y| \times (l_t + l_t^2)))$.

When the DOs apply blocking to their databases, each DO requires $O(l_h \times |\mathbf{D}|)$ to generate blocks based on q-gram based blocking [38], while with HLSH based blocking [58] the DOs require $O(b_B \times |\mathbf{D}|)$, where $b_B = \lceil l_t / l_b \rceil$ and l_b is the length of bit segments used to generate a blocking key. In the comparison process, the LU requires $O(n^2/n_b)$ block comparisons, where n is the number of shifted hash encoded q-gram lists or bit arrays in each database and n_b is the number of blocks.

6.7.3 Privacy Analysis

We now analyse what the parties involved in our approaches can learn from the data they receive from each other. We assume the DOs and the LU follow the honest-but-curious (HBC) adversary model (as we described in Section 2.2.2) where no DO colludes with the LU [116]. In our approaches, the DOs first communicate with each other to agree on parameter settings. This allows them to learn the parameters that are being used in the encoding processes but they cannot learn any sensitive information about the strings in each other's databases.

In both of our approaches, the DOs then individually encode the unique strings in their databases using the agreed parameters without learning any information from the other database. However, to generate the random bit arrays in the bit array based approach, the DOs employ a secure set intersection protocol [53] to find and exclude the common random bit arrays from their random bit arrays tables. These random bit arrays however do not represent any actual q-grams, and therefore the DOs do not learn any sensitive information from each other.

When blocking is used, the DOs apply a privacy-preserving blocking algorithm [38, 58] on their encoded strings before they are being sent to the LU. We assume such a blocking algorithm to be secure such that it does not allow the DOs to learn any sensitive information about each other's databases. The LU then receives the two encoded databases from the DOs. It first finds common blocks between them and then compares only encoded strings that are in the same blocks. In this step, the LU does learn which encodings occur in both databases, but not their actual contents.

In our shifted hash encoded q-gram based approach, the LU can learn the length of a string by guessing the length of q-gram, q (commonly used values are 2 and 3), and checking the length of the hash encoded q-gram lists. The LU can then generate q-grams from a publicly available database using the guessed q and compare the frequency of the generated q-grams and the hash encodings in a received database. However, in order to identify encoded q-grams, this public database must contain a very similar set of values with the same frequency distribution to the encoded database as otherwise the LU cannot employ a frequency analysis. Furthermore, an injection of faked values can be used to prevent such a frequency based attack [93].

As described in Section 6.3.3, in the basic encoded q-grams comparison algorithm, for each pair of shifted hash encoded q-gram lists, the LU finds the length of LCE by iteratively comparing and rotating the two lists. While the LU can keep the positions where the common hash encoded q-grams occur, it cannot learn the actual positions of these common hash encoded q-grams nor the positions of the original q-grams because (1) the common q-grams between two lists can occur at any position in the lists, and (2) the hash encoded q-grams in the two lists have been shifted by our random shifting technique. This results in the common patterns of the original string pairs are being distributed to different patterns of the encoded string pairs. In other words, the original positions where common q-grams occur in the q-gram lists have been shifted to other positions in the encoded and shifted lists.

Similarly, in the fast shifted hash encoded q-grams comparison algorithm, although the actual sequence of hash encoded q-grams is contained in the concatenation of the shifted lists, the LU still cannot learn the actual positions of where the original q-grams and the LCS occur in the two lists. This is because the LU does not know the start and end positions of the hash encoded q-grams in the compared lists.

In the bit array based approach, the LU receives bit arrays which are padded by random bits. The LU cannot learn the length of the original strings because every bit array has the same length. It also cannot learn the frequency distribution of the bits that are encoded of each q-gram because the q-gram bit arrays are shifted by a random number of bits, and therefore it cannot re-identify the original q-grams. The LU can only learn that the common bits occur in the middle of the two bit arrays (common pattern *m-m*) but it cannot learn the actual positions where the LCS occurs in the strings that correspond to a pair of bit arrays.

Once the LU has compared all encoded string pairs, it returns the length of LCE or LCB to the DOs. Each DO then calculates the length of the LCS, lcs. This allows each DO to learn the length of the LCS between a string in its database and a string in the other database, but it cannot learn the positions where the LCS occurs in strings. Therefore, each DO only learns the sub-string match.

6.8 Experimental Evaluation

We evaluated the linkage quality, scalability, and privacy of our approaches compared to Bloom filter (BF) encoding [155], tabulation based hashing (TMH) [170], and Damgård-Geisler-Krøigaard (DGK) homomorphic encryption based approximate string matching [66] (named DGK). We compared our approaches with these three baselines because BF encoding [155] is considered as a standard technique for PPRL, TMH [170] can be considered as an alternative to BF encoding, and DGK [66] is a recently proposed approach for secure string matching that encrypts strings based on their q-grams.

6.8.1 Datasets

In our evaluation, we required pairs of datasets where each pair contains different common patterns as we illustrate in Table 6.4. We used both real-world and synthetic data of types digit, letter, and mixed. In total, we evaluated our approaches and baselines on seven dataset pairs including four sets of real-world data and three sets of synthetic data. For real-world data, we extracted 100,000 strings of first names, cities, zip codes, and telephone numbers from the North Carolina Voter Registration[1] (NCVR) real-world database with snapshots from 2011 (first dataset) and 2019 (second dataset), as we described in Section 2.6.

[1] http://dl.ncsbe.gov/

Table 6.6: Lengths of the longest q-gram lists l_s, q-gram bit arrays l_q, calculated using Equation 6.1, and final bit arrays l_t, of different dataset pairs and alphabet sizes $|\Sigma|$.

Pair of datasets	Dataset	$\|\Sigma\|$	l_s	l_q	l_t
NCVR 2011-2019	First names	32	22	20	440
"	Cities	25	16	18	288
"	Zip codes	12	7	16	112
"	Telephone numbers	13	11	20	220
Synthetic-Corrupted	Credit card numbers	12	17	20	340
Synthetic-Corrupted	Barcode numbers	12	14	20	280
Synthetic-Corrupted	IBAN numbers	38	23	26	598

To generate the synthetic data, we used the Python package *Faker*[2], as we also described in Section 2.6, to create datasets of credit card, barcode, and IBAN (International Bank Account Number) numbers, where each such dataset contains 100,000 unique strings. We used each of these datasets as the first dataset in a pair. We then created the second dataset of a pair by replacing characters at different positions in each string in the first dataset with random characters of the same alphabet, where we ensured each dataset pair does contain different common patterns. However, the common pattern *b-e* (see Table 6.4) cannot occur for IBAN numbers because these numbers begin with letters and end with digits. Therefore, the beginning of the first IBAN number cannot be in common with the end of a second IBAN number in a string pair.

6.8.2 Experimental Setup

We padded strings in the two databases, $\mathbf{D}_A$ and $\mathbf{D}_B$, using the padding characters α = "$" and β = "#", respectively. We generated q-grams using $q = 3$ for first names, cities, and zip codes datasets, and $q = 4$ for telephone, credit card, barcode, and IBAN numbers. For each dataset, we used the minimum length of LCS $m = q$.

In the shifted hash encoded q-gram based approach, to generate hashed q-grams, we used the hash function $\mathcal{H}() = SHA256$ [154] and the agreed secret salt value s = "45". This salt value was also concatenated with q-grams for generating each q-gram bit array, b_q, in the bit array based approach. To generate the random bit arrays for the two DOs, DO_A and DO_B, we used the individual secret salt values, s_A = "65" and s_B = "56", respectively. We calculated the length of q-gram bit arrays, l_q, by using Equation 6.1. Table 6.6 shows the alphabet size and bit array length for each dataset. For the fast bit arrays comparison algorithm, we used segment lengths $l_\gamma = [10\%, 30\%, 50\%]$ of the minimum required length of LCB, l_{lcb}, to evaluate how different l_γ lengths influence the runtimes of the comparison process.

[2]https://pypi.org/project/Faker/

Table 6.7: Numbers of bits used for generating the *bkv* of blocks for different datasets.

Dataset	First names	Cities	Zip codes	Telephone numbers	Credit card numbers	Barcode numbers	IBAN numbers
Bit numbers	16	14	12	16	16	16	20

We compared our approaches with three baselines, which are BF [155], TMH [170], and DGK [66]. We used the same parameter settings as we used in our approaches, such as padding character α, q-gram length q, minimum length m, secret salt value s, and the hash function $\mathcal{H}()$. To generate the BF for a string, we used the random hashing [157] (as described in Section 2.3.1) to encode each q-gram set into a BF of 1,000 bits as it is a commonly used BF length for PPRL [155]. We used the optimal number of hash functions [122] for each dataset, which is 116 for first names, 87 for cities, 139 for zip codes, 87 for telephone numbers, 46 for credit card numbers, 58 for barcode numbers, and 33 for IBAN numbers.

For the TMH approach, we followed the original publication [170] and used 8 tabulation hash keys each of 64 bits to generate a bit array of length 1,000 bits to encode a string. For the DGK approach, we used keys of size 1,024 bits, and rather than using a two-party protocol as proposed in the original publication [66], we implemented a three-party protocol to be comparable with our approaches and the other two baselines by using a LU for conducting the comparison process.

To improve scalability, we applied a q-gram based blocking technique [38] for all approaches and applied HLSH based blocking [58, 100] on our bit array based, BF, and TMH approaches, as we described in Section 6.6. However, we only show results based on q-gram based blocking for BF [155] and TMH [170] as both blocking approaches (q-gram and HLSH based blockings) provide highly similar results.

For q-gram based blocking, we calculated the length of q-gram set permutations for generating blocks based on m and q, resulting in $t_q = 1$. For HLSH based blocking, in our bit array based approach, we generated blocking key values, *bkv*, by using the length of bit segments l_b calculated based on the bit percentage, $p_b = 80$ (the l_b calculation is described in Section 6.6).

For the BF [155] and TMH [170] baselines, we used the same length of bit segments l_b calculated based on $p_b = 80$ as in our approach because when using $p_b < 80$ the resulting bit segments are too short which leads to too many blocks for BFs or bit arrays of length 1,000 bits, and therefore a large number of comparisons. For example, as shown in Table 6.6, the first names dataset has the length of the q-gram bit array, $l_q = 20$. If the bit percentage p_b is less than 80, for example, $p_b = 30$, it results in the bit segments $l_b = 6$ and the number of blocks is 167 blocks, while with $p_b = 80$, it results in $l_b = 16$ (as illustrated in Table 6.7) and the number of blocks is 63. To generate each blocking key, *bkv*, we used the agreed secret salt value, $s = 45$, and the hash function $\mathcal{H}() = SHA256$.

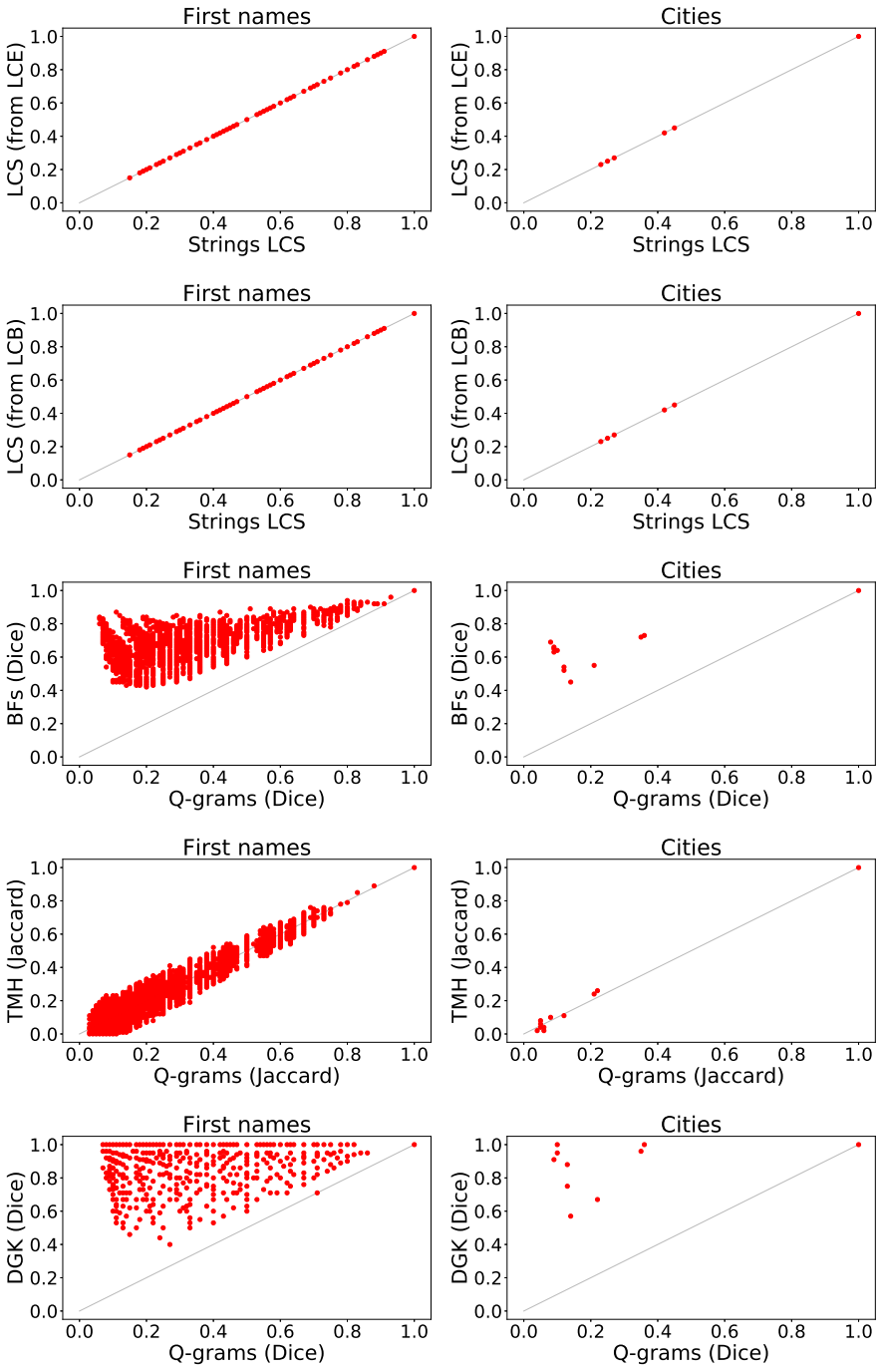

Figure 6.5: Similarity plots of shifted hash encoded q-gram based approach (first row), bit array based approach (second row), BF [155] (third row), TMH [170] (fourth row), and the DGK [66] (last row) for real-world dataset pairs of data type letter (column-wise).

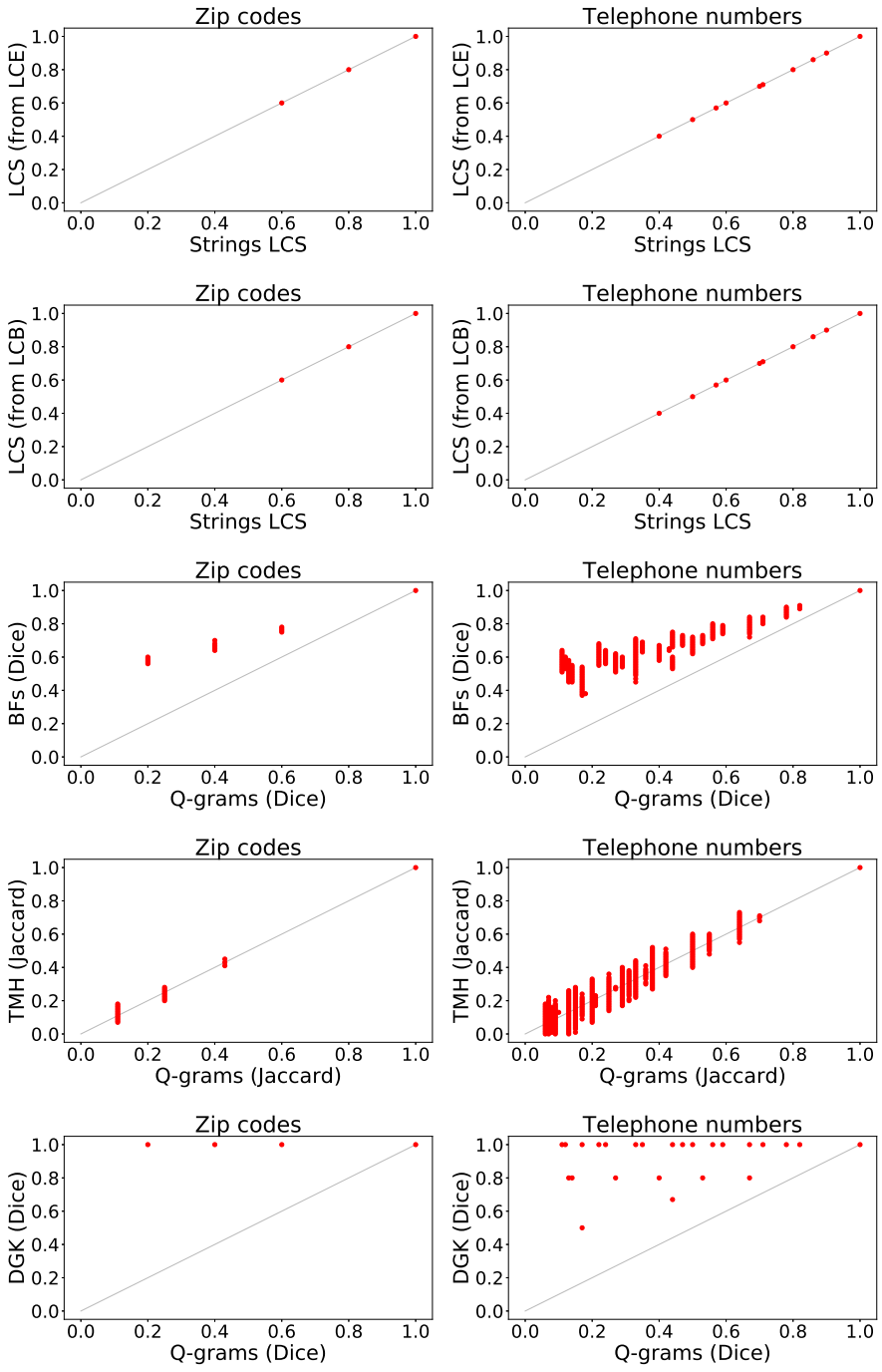

Figure 6.6: Similarity plots of shifted hash encoded q-gram based approach (first row), bit array based approach (second row), BF [155] (third row), TMH [170] (fourth row), and the DGK [66] (last row) for real-world dataset pairs of data type digit (column-wise).

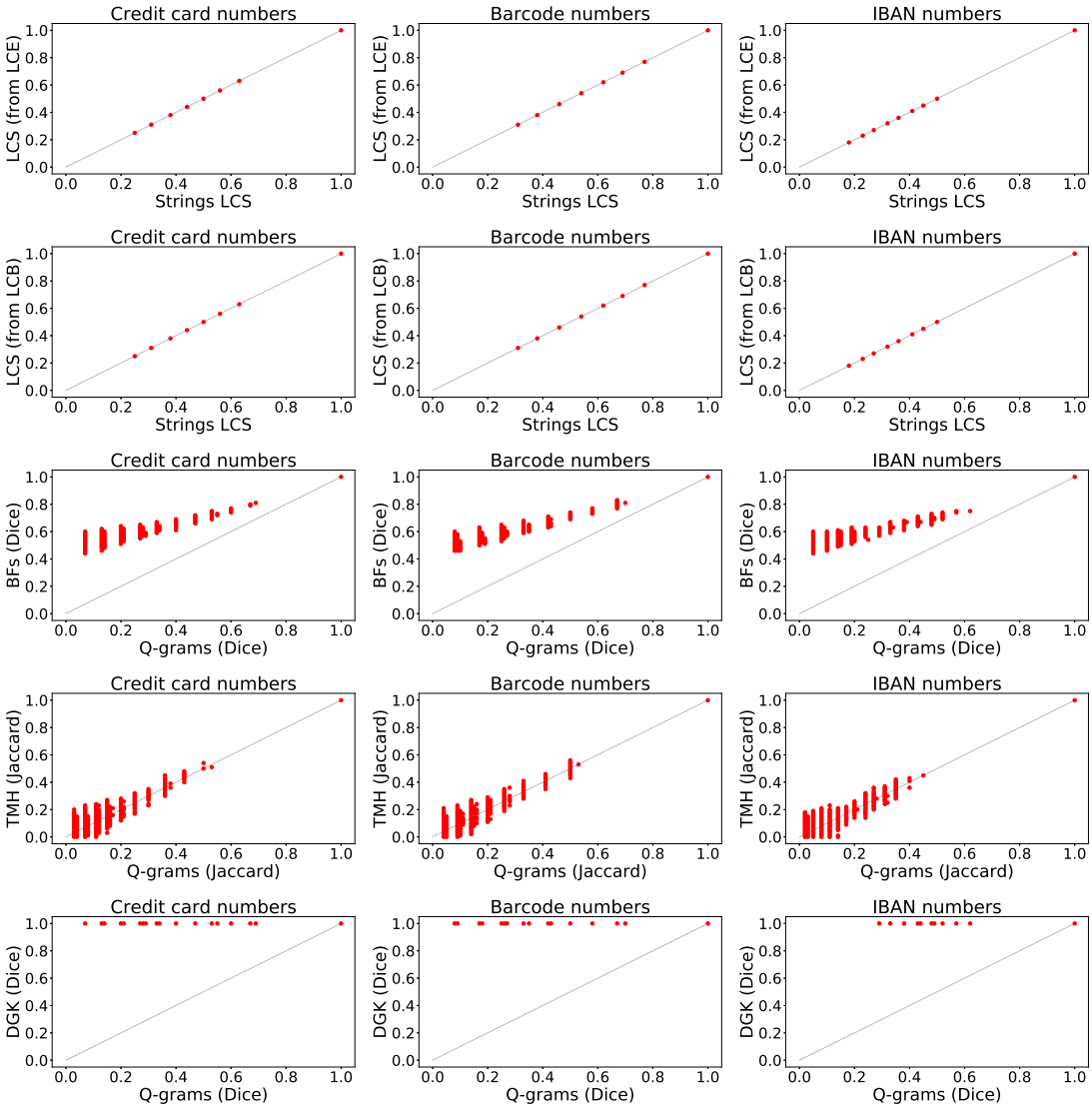

Figure 6.7: Similarity plots of shifted hash encoded q-gram based approach (first row), bit array based approach (second row), BF [155] (third row), TMH [170] (fourth row), and the DGK [66] (last row) for synthetic dataset pairs (column-wise) of data type digit (credit card and barcode numbers) and mixed (IBAN numbers).

6.8.3 Linkage Quality Results

We evaluated the accuracy of all approaches based on the correctness of similarity calculations. We compared the length of the LCS of unencoded string pairs with the calculated length of LCS, lcs, of the corresponding encoded string pairs based on Equation 6.2. To be comparable with the BF [155], TMH [170], and DGK [66] baselines, we normalised the lcs into the range [0...1] of similarity values, calculated as $sim_{lcs} = lcs/max(|x|, |y|)$, where x and y are the strings in a pair.

For the BF approach, we calculated the similarity of q-gram sets and of BFs using the Dice-coefficient similarity [155], while we used the Jaccard similarity calculation for q-gram sets and bit arrays generated by the TMH encoding technique [170]. For the DGK approach, we calculated the similarity of q-gram and encryption (ciphertext) lists using the Dice-coefficient [66].

Figures 6.5 to 6.7 show scatter plots of real-world and synthetic dataset pairs, where the horizontal axis shows unencoded (plaintext) similarities and the vertical axis shows the corresponding encoded (or encrypted) similarities. Points on the diagonal show pairs of strings where both the unencoded and the encoded similarities are the same, while any point off the diagonal shows differences in the calculated similarities between unencoded and encoded string pairs.

As can be seen, both our approaches provide accurate string similarity results, while BF [155] and TMH [170] approaches can result in inaccurate similarities. This is because of a high number of hash collisions that occur with both encoding approaches, where different q-grams are hashed into the same bit positions.

The DGK [66] approach also results in inaccurate similarities. This is because the Dice-coefficient of the ciphertexts is calculated based on the cardinality, where some ciphertexts that represent encrypted q-grams of strings in a pair are not common, although these ciphertexts are common between the two lists of all possible q-grams that were used to generate the intersection set of cardinality.

6.8.4 Scalability Results

To be comparable between our approaches and the baselines, we use a three-party protocol for all approaches [38]. We applied q-gram based blocking to our shifted hash encoded q-gram based approach and the three baselines, while we applied HLSH based blocking to our random bit array based approach and the BF [155] and TMH [170] baselines (as described in Section 6.8.2).

Table 6.8 shows the number of unique strings in each dataset in a pair ($\mathbf{D}_A$ / $\mathbf{D}_B$) and the numbers of string pair comparisons of the different dataset pairs and approaches, where we show only the numbers of comparisons based on q-gram based blocking for the three baselines and limit the maximum number of comparisons to 1,000,000 string pairs. We evaluated the runtimes of the encoding process by a DO and the comparison process by the LU, as shown in Figures 6.8 and 6.9, respectively, where we report the average times for one string or string pair in milliseconds.

As can be seen in Figure 6.8, our shifted hash encoded q-grams based approach is the fastest encoding technique while the TMH [170] and DGK [66] approaches are the slowest encoding techniques. Different sizes of the alphabets, types of strings (such as letters, digits, or mixed), and lengths of strings do not much affect any encoding approaches. However, the TMH [170] and DGK [66] approaches consume more average runtimes for one string of the dataset with a smaller size, such as city and zip code. This is because of the overhead of generating tabulation hashes in the TMH [170], and generating all possible q-grams in the DGK [66].

Table 6.8: Numbers of unique strings and string pair comparisons for different dataset pairs and using different encoding approaches, where we limit the number of comparisons to the maximum of 1,000,000 (1M) string pairs.

Dataset	Unique strings $\mathbf{D}_A$ / $\mathbf{D}_B$	Shifted hashed q-gram based	Bit array based	BF [155]	TMH [170]	DGK [66]
First names	9,154 / 11,350	929,927	1M	1M	1M	1M
Cities	20 / 13	19	20	23	23	44
Zip codes	21 / 13	60	52	96	96	96
Telephone numbers	63,475 / 63,656	1M	1M	1M	1M	1M
Credit card numbers	100,000 / 100,000	1M	1M	1M	1M	1M
Barcode numbers	100,000 / 100,000	1M	1M	1M	1M	1M
IBAN numbers	100,000 / 100,000	1M	1M	1M	1M	1M

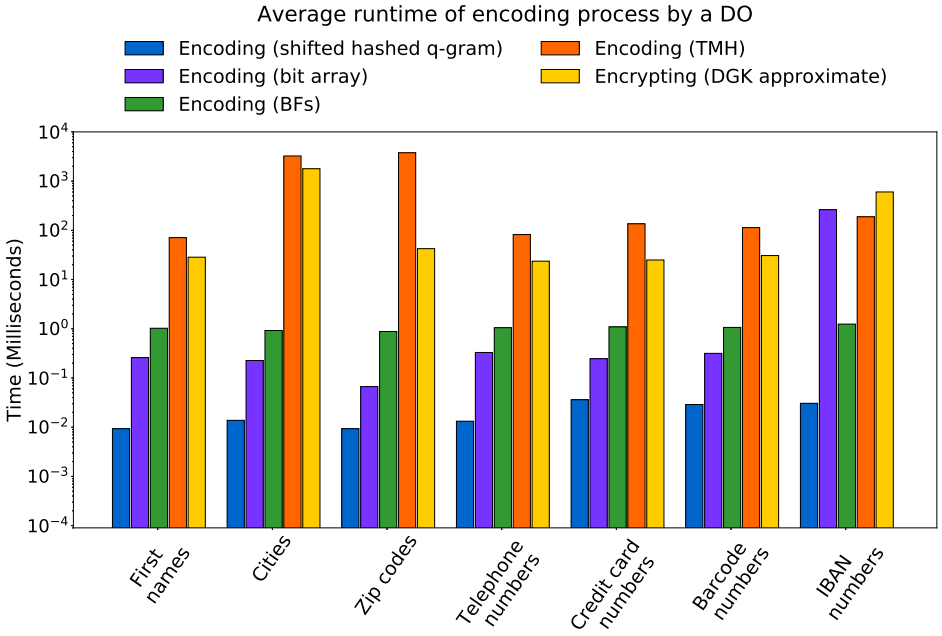

Figure 6.8: Runtimes comparison of the encoding processes by a DO between our approaches, BF [155], TMH [170], and DGK [66] approaches.

As can be seen in Figure 6.9, our approaches consume similar comparison runtimes to the DGK approach [66] and have longer runtimes than BF [155] and TMH [170] encodings, where these two baselines have similar runtimes. This is because the comparison processes of our approaches are more complicated, where we find sequences of common encodings that occur in the encoded strings pair and then find the LCS between them, while BF [155] and TMH [170] approaches calculate approximate similarities based on the set intersection of 1-bits that occur in a pair of encoded strings.

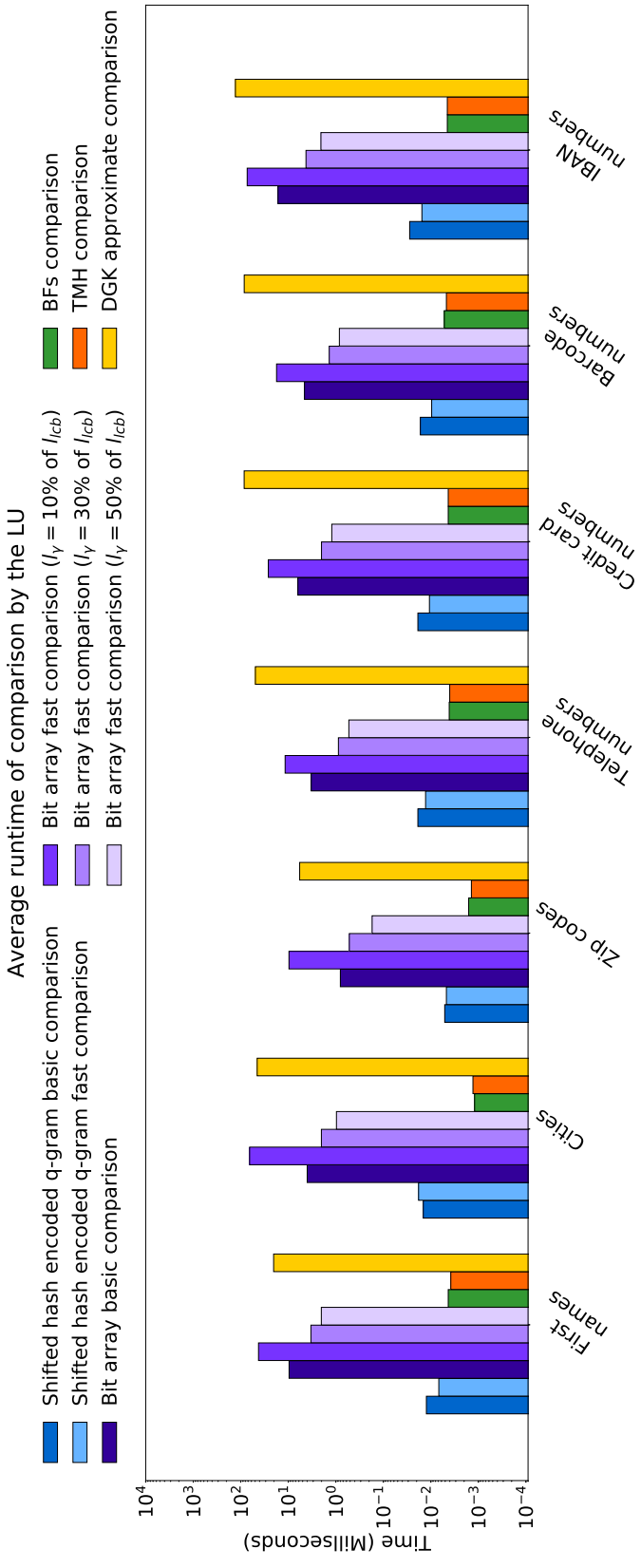

Figure 6.9: Runtimes comparison of encoded string comparison by a LU between our approaches, BF [155], TMH [170], and DGK [66] approaches.

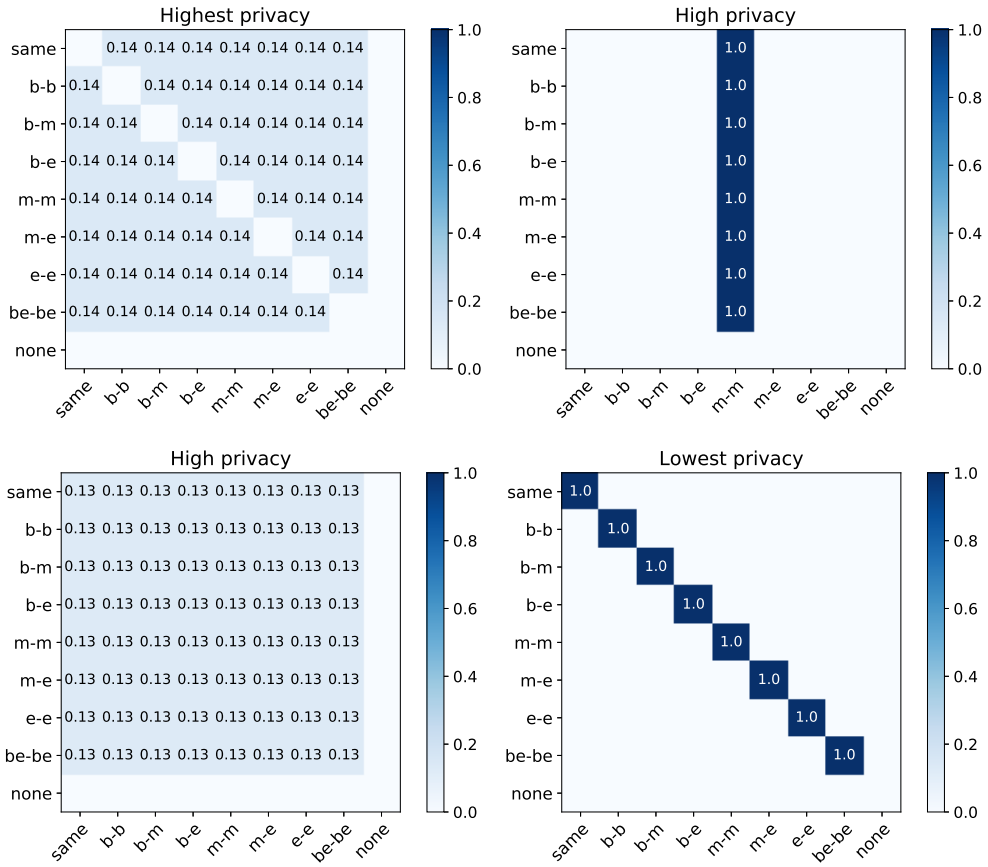

Figure 6.10: Heatmap [63] plots of different privacy levels of a PPRL approach. The highest to lowest privacy are shown from top left to bottom right.

The runtimes of our fast comparison algorithms are faster than the basic comparison algorithms. However, in the bit array based approach, the fast comparison algorithm is slower than the basic comparison algorithm when we use the segment length $l_\gamma = 10\%$ of the l_{lcb}. This is because of the overhead of the fast comparison algorithm which needs to generate segments and find the common sequences of bits to the left and right of common segments between bit arrays in a pair.

6.8.5 Privacy Results

We first evaluated the privacy of our approaches and the baselines by identifying the common patterns between unencoded (plaintext) and encoded (or encrypted) string pairs. Figure 6.10 shows example heatmap [63] plots of different levels of privacy provided by a PPRL approach. Figures 6.11 to 6.13 show heatmap [63] plots of privacy levels evaluated by using our approaches and the baselines on real-world and synthetic datasets.

In each plot, the vertical axis shows the common patterns of unencoded string pairs and the horizontal axis shows the common patterns of corresponding encoded

string pairs. The dark blue colour indicates a higher percentage of unencoded and encoded string pairs while a lower percentage of pairs are shown in light blue.

As we illustrate in Figure 6.10, an approach provides the highest privacy when there are no common patterns of unencoded and encoded string pairs (the top-left plot). In contrast to the highest privacy, an approach provides the lowest privacy when all patterns of unencoded and encoded string pairs are in commons (the bottom-right plot). An approach provides high privacy when (1) the patterns of unencoded string pairs become the *m-m* pattern when the strings in these pairs are encoded (as illustrated in the top-right plot), and (2) the patterns of unencoded string pairs become different patterns when the strings in these pairs are encoded (as illustrated in the bottom-left plot). Therefore, when common patterns of unencoded string pairs become different patterns in their corresponding encoded string pairs, it is more difficult for an adversary to identify the original q-grams and the positions where the LCS occur.

As can be seen in Figures 6.11 to 6.13, the common patterns of encoded string pairs using our approaches are different from the common patterns of unencoded string pairs, where no string pair has the *same* common pattern. With the shifted hash encoded q-grams based approach, each common pattern of unencoded string pairs is distributed to different common patterns when the strings in these pairs are encoded. The highest number of encoded string pairs in many datasets is the common pattern *m-m*, which means the encoded strings in a pair are common in the middle of the shifted hash encoded q-gram list, **h**. Similarly, in the bit array based approach, most common patterns of unencoded string pairs become the common pattern *m-m* when the strings in pairs are encoded, as we described in Section 6.7.3.

For the DGK homomorphic encryption based approximate string matching approach [66], the common patterns of unencrypted and encrypted string pairs are all the same because the common patterns of unencrypted string pairs are not distributed to other common patterns of encrypted string pairs. Each unencrypted q-gram in a pair and its corresponding ciphertext (encryption of 1) is located at the same position in the list of all possible q-grams. However, this approach still provides high degrees of privacy because of the use of the DGK homomorphic encryption [47] which results in the same string value being encrypted into different ciphertexts. Therefore, although the common patterns of unencrypted and encrypted string pairs are the same, it can be difficult to re-identify the original string values because an adversary cannot learn if a ciphertext represents a 0 or 1.

For the BF [155] and TMH [170] based encoding approaches, the common patterns of the same string values are not being distributed to other common patterns when they are encoded, and therefore the common pattern is the *same* common pattern. The encoded string pairs can have the *none* common pattern because the bits encoded of q-grams are not located in sequential order when using these approaches. The bits of common q-gram between two strings in a pair can be located next to bits that encoded of not common q-grams, and the sequence of bits in a BF or TMH bit array is then a mix of common and not common q-grams.

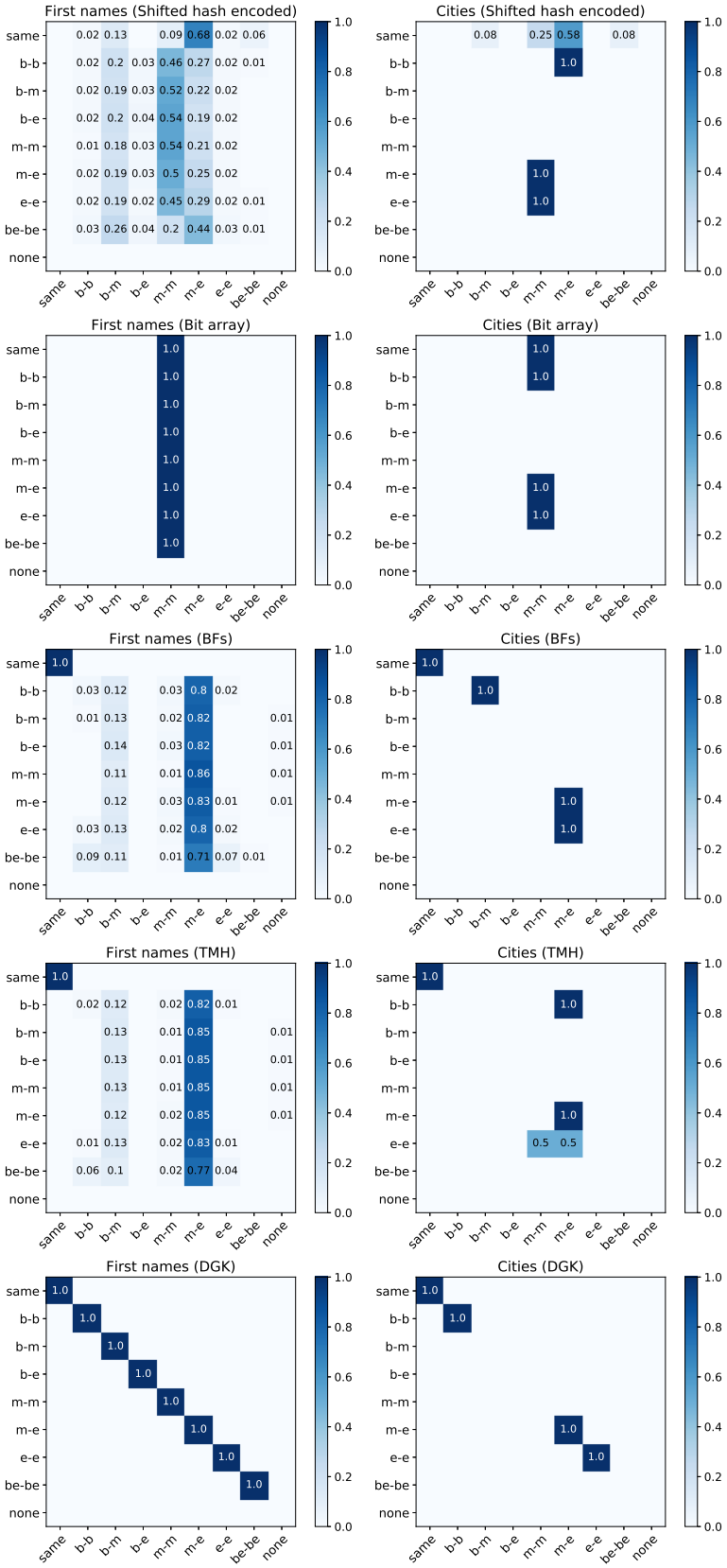

Figure 6.11: Heatmap [63] plots of real-world datasets of data type letter (column-wise) and different approaches. Each row shows shifted hash encoded q-gram, bit array, BF [155], TMH [170], and DGK [66], respectively.

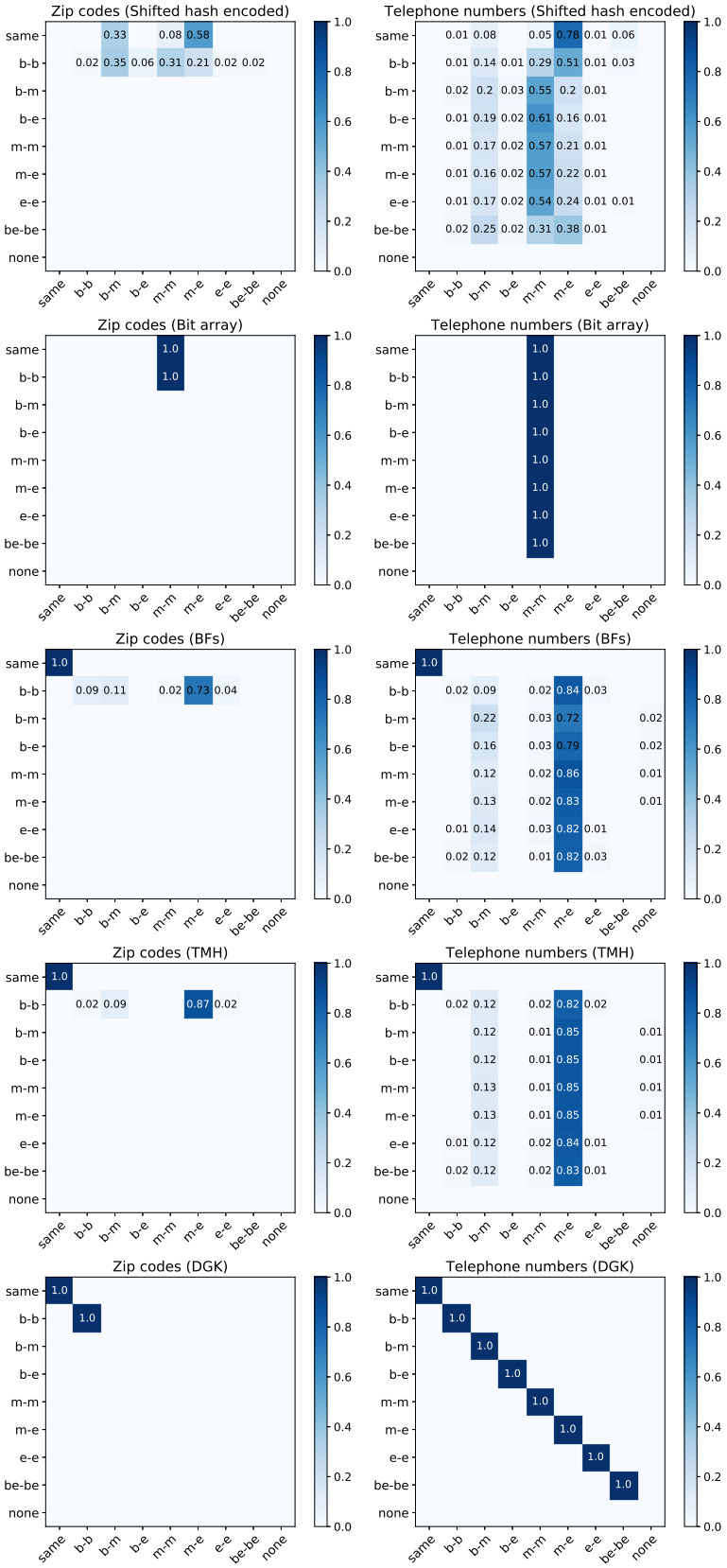

Figure 6.12: Heatmap [63] plots of different real-world datasets of data type digit (column-wise) and different approaches. Each row shows shifted hash encoded q-gram, bit array, BF [155], TMH [170], and DGK [66] approaches, respectively.

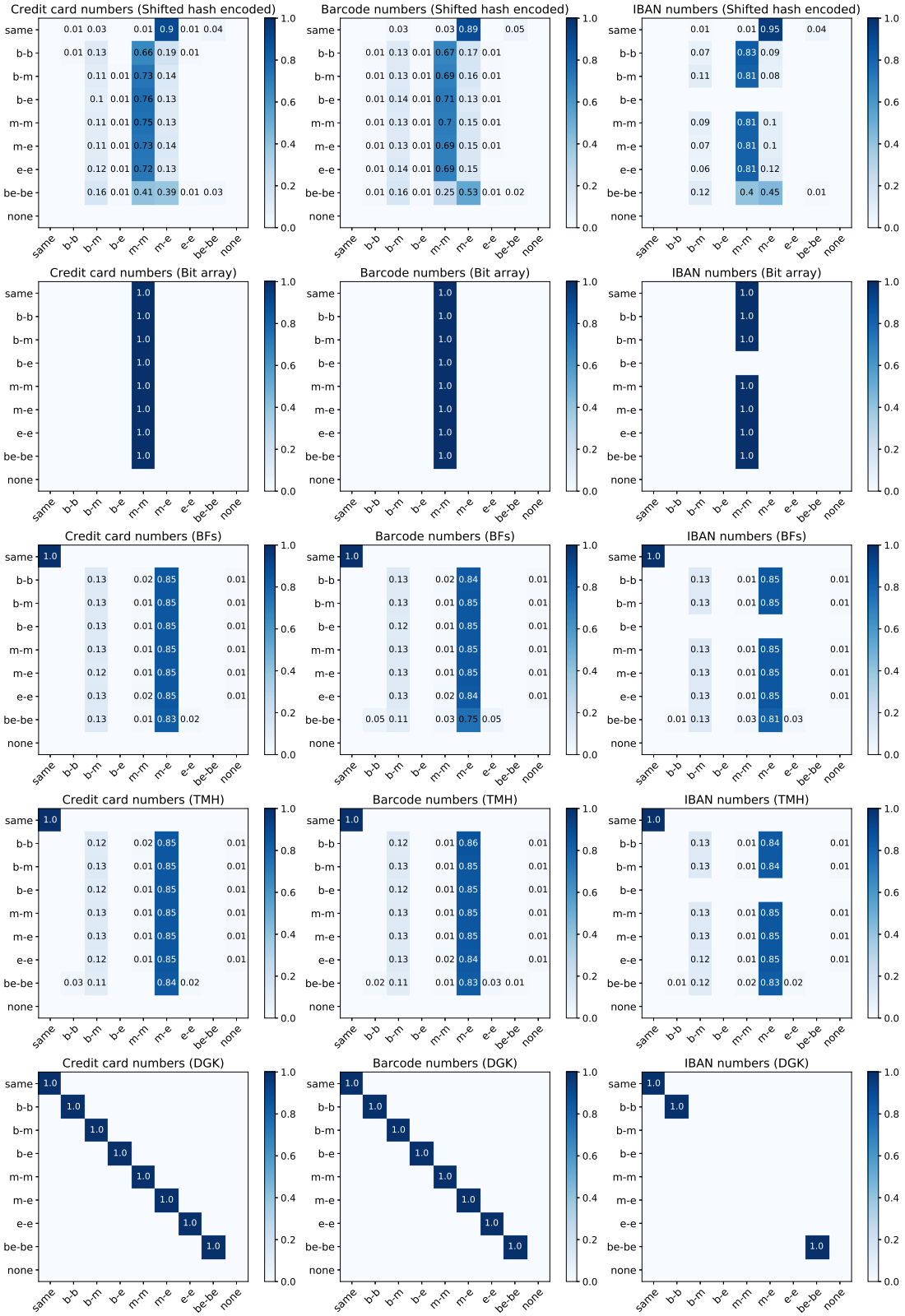

Figure 6.13: Heatmap [63] plots of different synthetic datasets (column-wise) of data type digit (credit card and barcode numbers) and mixed (IBAN numbers) and different approaches. Each row shows shifted hash encoded q-gram, bit array, BF [155], TMH [170], and DGK [66], respectively.

For example, let us assume the two BFs are $b_x = 1\mathbf{1}01000\mathbf{1}0$ and $b_y = 0\mathbf{1}00010\mathbf{1}\mathbf{1}0$. They have common bits encoding of common q-gram located at positions 1 and 8 (as shown in bold) of the BFs. These two bits are located next to the bits encodings of not common q-grams. This BF pair cannot be categorised to any of the common patterns (as illustrated in Table 6.4), and therefore the common pattern of this BFs pair is *none*.

6.9 Chapter Summary

In this chapter, we have presented two privacy-preserving record linkage (PPRL) approaches, which are shifting hash encoded q-grams based (as we described in Section 6.3) and bit array based (as we described in Section 6.4), that allow the accurate and efficient calculation of the length of the longest common sub-string (LCS) between strings. Our approaches encode sensitive string values such that no re-identification is possible, while also preventing frequency attacks on individual character encodings (as we discussed in Section 6.7).

Our experimental evaluation, as we discussed in Section 6.8, has shown that both our approaches result in the same string similarities as on the original un-encoded strings, while the commonly used Bloom filter encoding [155], tabulation based hashing [170], and DGK homomorphic encryption based approximate string matching [66] approaches will lead to potentially much higher or lower similarities between encoded strings. Our approaches are faster than the TMH [170] and DGK [66] baselines. However, our approaches use more runtimes for the comparison process than the BF [155] and TMH [170] baselines, while our bit array based approach uses similar runtimes to the DGK [66] approach. In the following chapter, we present our fast and accurate PPRL techniques proposed for string matching.

Chapter 7

Privacy-Preserving Record Linkage for Fast and High Linkage Quality

Our proposed approaches, as we discussed in Chapter 6, can provide accurate string matching. However, they consume long runtimes in the comparison process. In this chapter, we discuss our novel proposed privacy-preserving record linkage (PPRL) approaches for fast and high quality string matching based on the length of the longest common sub-string (LCS) between the strings to be compared.

In Section 7.1, we provide an introduction to our PPRL approaches, and in Section 7.2 we then provide brief descriptions of the special 1- and 0-encodings and packing encryptions that we use in these approaches. In Section 7.3, we give an overview of our PPRL protocols, and in Section 7.4 we describe our data generation process. We then discuss our proposed two approaches which are (1) using one numerical value encrypted into one ciphertext value, as we describe in Section 7.5, and (2) using multiple numerical values encrypted into one ciphertext value, as we describe in Section 7.6. Based on the results of these two approaches, we then find the length of the LCS between the compared numbers, as we discuss in Section 7.7. In Section 7.8 we analyse our approaches in terms of linkage quality, scalability, and privacy, and in Section 7.9 we discuss our experimental evaluation. Finally, in Section 7.10, we summarise this chapter.

7.1 Introduction

As we described in Chapters 1 and 3, some organisations require PPRL techniques that provide fast string matching results for data analysis [96]. For example, the linkage of crime, immigration, and airline databases for reporting criminal activities needs a fast response for a crime prevention organisation to take action proactively. Blocking techniques for record linkage have been proposed to speed up the computation time of the comparison process for large databases by reducing the comparison space between databases [96, 97]. However, these techniques can result in low linkage quality if similar string records are inserted into different blocks, and are not being compared [184].

Table 7.1: Notation and terminology used in this chapter.

$\mathbf{D}, \mathbf{D}_A, \mathbf{D}_B$	Database, database A, and database B to be encoded, respectively
$\mathbf{G}$	Publicly available global database
$\mathbf{R}$	List of reference values
$\mathbf{B}$	Inverted index of blocks
$\mathbf{T}$	Data to be encrypted
$\mathbf{E}$	Encoded database
v, x, y	String value, string value in $\mathbf{D}_A$, and string value in $\mathbf{D}_B$, respectively
s	Sub-string value
$c, \mathbf{c}$	Common sub-string and set of common sub-strings between strings in a pair
q	Length of q-gram
$\mathbf{q}$	List of q-grams
t	Threshold for generating blocking key and reference values
s_t	Similarity threshold
q_m	Minimum number of q-grams for generating blocking key and reference values
b	Binary string
p	Binary value at each position in a binary string
l	Length of a binary string
l_v, l_s, l_v^s	Length of a string, sub-string, and a sub-string in a string, respectively
n_v^s	Integer form of l_v^s
d	Number of decimal places
$HMAC()$	Hash function HMAC [107]
lcs	Length of the longest common sub-string
$\|...\|$	Size or number of values in a database or list
$\|\|$	Strings concatenation
bkv	Blocking key value
m	Minimum length of the LCS
LCS	Longest common sub-string
ShiftedHash	Shifted hash encoded q-gram based approach
BitArray	Bit array based approach

As we discussed in Chapter 6, our proposed shifted hash encoded q-grams based and bit array based approaches provide accurate linkage quality compared to Bloom filter (BF) encoding [155] and the other two baselines. However, the bit array based approach has longer runtime, while the shifted hash encoded q-gram based approach can reveal the lengths of the lists of encoded q-grams to an adversary which potentially can reveal the lengths of the original strings encoded within them [176]. In contrast, Bloom filter (BF) encoding [155] consumes less runtime for encoding and comparison processes, however it provides lower linkage quality due to false matches [38].

In this chapter, we present two PPRL approaches which provide high linkage quality and use less runtimes in the comparison process. The main idea of our proposed approaches is to compare the lengths of sub-strings in the strings (numerical values) of a pair of strings, where each length corresponds to a sub-string that can be the longest common sub-string (LCS) of the pair.

In the first approach, as we describe in Section 7.5, we encrypt a length of a sub-string in a string into one ciphertext and also encode this length into hash values of special 1- or 0-encodings [114] which will be used for comparison. In the second

approach, as we describe in Section 7.6, we encrypt multiple lengths of sub-strings in strings into a single ciphertext and compare them in a single comparison. In both approaches, the ciphertexts are sent to a third party for comparison. We then identify the lengths of the LCS of string pairs based on the compared results, as we describe in Section 7.7. In Table 7.1, we provide the notation and terminology that we use in this chapter.

7.2 Comparison of Ciphertexts

In our proposed approaches, we generate ciphertexts for the lengths of sub-strings in the strings and compare these ciphertexts based on two techniques which are special 1- and 0-encodings [114] (encoding of an integer number) and a packing based encryption method [26] (encrypting of multiple decimal numbers).

7.2.1 Comparison of Special 1- and 0-Encodings

As we discussed in Chapter 3, Lin and Tzeng [114] proposed an approach to compare encrypted numerical values (integers) based on the sets of special 1- and 0-encodings, where each encoding is a binary string. The two database owners (DOs) that participate in the protocol decide on who will generate the set of special 1- or 0-encodings and make an agreement on the length of a binary string, l, to be used. We assume the first DO (DO_A) generates a set of 1-encodings, $\mathbf{E}_x^1$, for its number x and the second DO (DO_B) generates a set of 0-encodings, $\mathbf{E}_y^0$, for its number y.

Each DO converts its integer number into a binary string, b, of length l as $b = p_1p_2...p_l$, where each p is a binary value at each position in a binary string b. The DO uses b to iteratively generate a special encoding which is then inserted into its set ($\mathbf{E}_x^1$ or $\mathbf{E}_y^0$). Each $e^1 \in \mathbf{E}_x^1$ and $e^0 \in \mathbf{E}_y^0$ are generated using Equations 7.1 and 7.2, respectively.

$$e^1 = p_1p_2...p_i \text{ if } p_i = 1, \tag{7.1}$$

$$e^0 = p_1p_2...p_i \mid p_i = 1 \text{ if } p_i = 0, \tag{7.2}$$

where p_i is a binary value at position i in a binary string b and $1 \leq i \leq l$. The position i is first initialised to $i = l$ and it is decreased by one position for each iteration until $i = 1$. Therefore, the number of iterations that a DO needs for generating encodings is l iterations.

For example, we assume DO_A has the number $x = 9$ with the binary string $b_x =$ "1001" of length $l = 4$. DO_A uses Equation 7.1 to generate its encoding e^1. In the first iteration, the p_i at the position $i = l = 4$ is 1, and therefore DO_A generates an encoding $e_1^1 =$ "1001" and inserts it into the set $\mathbf{E}_x^1$. DO_A decreases i by one and repeats the steps until $i = 1$. As a result, DO_A has the set $\mathbf{E}_x^1 = \{$"1001","1"$\}$. We assume the DO_B has the number $y = 8$ with the binary string $b_y =$ "1000". DO_B conducts the same as DO_A but it uses Equation 7.2 to generate its encodings.

Therefore, if p_i in b is 0, then DO_B replaces the p_i by 1. As a result, the DO_B has the set $\mathbf{E}_y^0 = \{\text{“1001”}, \text{“101”}, \text{“11”}\}$.

The two generated sets $\mathbf{E}_x^1$ by DO_A and $\mathbf{E}_y^0$ by DO_B are then used to conduct a comparison where:

$$comp(x,y) = \begin{cases} 1 : x > y & \text{if } \mathbf{E}_x^1 \cap \mathbf{E}_y^0 \neq \emptyset, \\ 0 : x \leq y & \text{if } \mathbf{E}_x^1 \cap \mathbf{E}_y^0 = \emptyset. \end{cases} \tag{7.3}$$

From the examples we described earlier, DO_A has $x = 9$ and the set $\mathbf{E}_x^1 = \{\text{“1001”}, \text{“1”}\}$. DO_B has $y = 8$ and the set $\mathbf{E}_y^0 = \{\text{“1001”}, \text{“101”}, \text{“11”}\}$. As can be seen, these two sets have a common encoding which is “1001” ($\mathbf{E}_x^1 \cap \mathbf{E}_y^0 \neq \emptyset$). Therefore, the comparison result between numbers x of DO_A and y of DO_B returns as 1 which means $x > y$ ($9 > 8$) based on Equation 7.3. We use these special 1- and 0-encodings and the comparison in Equation 7.3 [114] in our first approach, as we describe in Section 7.5.

7.2.2 Comparison of Packs of Encryptions

As we discussed in Chapter 3, Cheon et al. [26] proposed a homomorphic encryption scheme, called Cheon-Kim-Kim-Song (CKKS), that supports arithmetic operations, such as addition, subtraction, and multiplication, to be conducted over ciphertexts. The authors also proposed a packing method that encrypts a list of multiple numerical values (decimal numbers) into a single ciphertext. Recently, Cheon et al. [27] proposed an approach to find the minimum value between two ciphertexts that can result in errors within a $2^{-\alpha}$ bound, where α is the precision of a ciphertext. Their comparison function is based on the polynomial composition ($\circ$) function, $z = f(z) \circ g(z) = f(g(z))$, where z is initialised as $z = E(\mathbf{x}) - E(\mathbf{y})$, $E(\mathbf{x})$ is the encryption of a list of decimal numbers, $\mathbf{x}$, from the first DO, and $E(\mathbf{y})$ is the encryption of a list of decimal numbers, $\mathbf{y}$, from the second DO.

The authors suggested optimal functions $g()$ and $f()$ that result in a maximum of 2^{-4} errors. The polynomial of $g()$ and $f()$ can be written as [27]:

$$g(z) = \frac{46623}{2^{10}}z^9 - \frac{113492}{2^{10}}z^7 + \frac{97015}{2^{10}}z^5 - \frac{34974}{2^{10}}z^3 + \frac{5850}{2^{10}}z, \tag{7.4}$$

$$f(z) = \frac{35}{128}z^9 - \frac{180}{128}z^7 + \frac{378}{128}z^5 - \frac{420}{128}z^3 + \frac{315}{128}z. \tag{7.5}$$

The calculated z is then used to find a minimum value between ciphertexts $E(\mathbf{x})$ and $E(\mathbf{y})$ as:

$$min(E(\mathbf{x}), E(\mathbf{y})) = \frac{E(\mathbf{x}) + E(\mathbf{y})}{2} - \left(\frac{E(\mathbf{x}) - E(\mathbf{y})}{2} \times z\right). \tag{7.6}$$

As a result, the two packs (lists) of decimal numbers can be compared by using a single comparison.

For example, assume two DOs, DO_A and DO_B, participate in a linkage protocol, where DO_A has a list of decimal numbers $\mathbf{x} = [0.83, 1.0]$ and uses the packing method

to encrypt $\mathbf{x}$ into the ciphertext $E(\mathbf{x}) = E([0.83, 1.0])$ and DO_B has a list of decimal numbers $\mathbf{y} = [1.0, 0.5]$ and uses the packing method to encrypt $\mathbf{y}$ into the ciphertext $E(\mathbf{y}) = E([1.0, 0.5])$. To conduct a comparison based on the approach by Cheon et al. [27], we initialise z by conducting a subtraction between $E(\mathbf{x})$ and $E(\mathbf{y})$ as $z = E(\mathbf{x}) - E(\mathbf{y}) = E([-0.17, 0.5])$.

We then conduct polynomial composition function $z = f(z) \circ g(z)$ by first using the initialised z as the input to the function $g()$ (Equation 7.4) as:

$$g(z) = \frac{46623}{2^{10}}E([-0.17^9, 0.5^9]) - \frac{113492}{2^{10}}E([-0.17^7, 0.5^7]) + \frac{97015}{2^{10}}E([-0.17^5, 0.5^5]) - \frac{34974}{2^{10}}E([-0.17^3, 0.5^3]) + \frac{5850}{2^{10}}E([-0.17, 0.5]) = E([-0.81, 0.78]).$$

We then use the output of $g()$ as the input z to the function $f()$ (in Equation 7.5) as:

$$f(z) = \frac{35}{128}E([-0.81^9, 0.78^9]) - \frac{180}{128}E([-0.81^7, 0.78^7]) + \frac{378}{128}E([-0.81^5, 0.78^5]) - \frac{420}{128}E([-0.81^3, 0.78^3]) + \frac{315}{128}E([-0.81, 0.78]) = E([-0.99, 0.99]).$$

The output of the function $f()$ is the new z which is then used to calculate the minimum $min(E(\mathbf{x}), E(\mathbf{y}))$, based on Equation 7.6 as:

$$\begin{aligned} min(E(\mathbf{x}), E(\mathbf{y})) &= \frac{E([1.83,1.5])}{2} - \frac{E([-0.17,0.5])}{2} \times E([-0.99, 0.99]) \\ &= E([0.91, 0.75]) - E([-0.08, 0.25]) \times E([-0.99, 0.99]) \\ &= E([0.83, 0.5]). \end{aligned}$$

As can be seen from this example, the minimum value between $E(\mathbf{x}) = E([0.83, 1.0])$ and $E(\mathbf{y}) = E([1.0, 0.5])$ is $E([0.83, 0.5])$ which is the correct result. We use the packing method [26] and the polynomial composition function [27] in our second approach, as we describe in Section 7.6.

7.3 Protocol Overview

Our PPRL approaches involve three parties, the two database owners (DOs) and a linkage unit (LU), where we assume our approaches follow the honest-but-curious (HBC) adversary model [74, 81], as we described in Section 2.2.2. The DOs want to find the lengths of the longest common sub-string (LCS) between pairs of sensitive strings in their databases, $\mathbf{D}_A$ and $\mathbf{D}_B$, and they do not communicate with each other, except to agree on the parameters to be used in the PPRL protocol. The LU is used to compare the values sent to it by the DOs. However, the DOs do not want to reveal any sensitive information, such as the positions of sub-strings occurring in strings, the frequencies of sub-strings, and the string values in their databases to any other parties that participate in the protocol. Therefore, the DOs encrypt their sensitive values and send them to the LU such that the LU cannot learn anything about them.

In our approaches, we normalise the length of the LCS, lcs, between strings in a pair, (x, y), where $x \in \mathbf{D}_A$ and $y \in \mathbf{D}_B$, into the range $[0...1]$ to make all pairs of strings with different lengths to be matched based on one similarity threshold, s_t.

Databases	**Strings comparison**			
$\mathbf{D}_A$	$x_i \in \mathbf{D}_A$	$y_i \in \mathbf{D}_B$	LCS	Length of LCS, *lcs*
$x_1 =$ "*marry*"	$x_1 =$ "*marry*"	$y_1 =$ "*mary*"	"*mar*"	$min(\frac{3}{5}, \frac{3}{4}) =$ 0.6
$x_2 =$ "*amar*"	$x_1 =$ "*marry*"	$y_2 =$ "*amary*"	"*mar*"	$min(\frac{3}{5}, \frac{3}{5}) =$ 0.6
$x_3 =$ "*mary*"	$x_1 =$ "*marry*"	$y_3 =$ "*emary*"	"*mar*"	$min(\frac{3}{5}, \frac{3}{5}) =$ 0.6
	$x_2 =$ "*amar*"	$y_1 =$ "*mary*"	"*mar*"	$min(\frac{3}{4}, \frac{3}{4}) =$ 0.75
	$x_2 =$ "amar"	**$y_2 =$ "amary"**	**"amar"**	**$\mathbf{min}(\frac{4}{4}, \frac{4}{5}) =$ 0.8**
$\mathbf{D}_B$	$x_2 =$ "*amar*"	$y_3 =$ "*emary*"	"*mar*"	$min(\frac{3}{4}, \frac{3}{5}) =$ 0.6
$y_1 =$ "*mary*"	**$x_3 =$ "mary"**	**$y_1 =$ "mary"**	**"mary"**	**$\mathbf{min}(\frac{4}{4}, \frac{4}{4}) =$ 1.0**
$y_2 =$ "*amary*"	**$x_3 =$ "mary"**	**$y_2 =$ "amary"**	**"mary"**	**$\mathbf{min}(\frac{4}{4}, \frac{4}{5}) =$ 0.8**
$y_3 =$ "*emary*"	**$x_3 =$ "mary"**	**$y_3 =$ "emary"**	**"mary"**	**$\mathbf{min}(\frac{4}{4}, \frac{4}{5}) =$ 0.8**

Figure 7.1: Example of the *lcs* calculation (Equation 7.7) between strings $x_i \in \mathbf{D}_A$ and $y_i \in \mathbf{D}_B$, where $0 < i \leq |\mathbf{D}_A|$ and $|\mathbf{D}_A| = |\mathbf{D}_B|$. The string pairs that have an *lcs* of at least the similarity threshold $s_t = 0.8$ are classified as matches, as we show in bold. The string pairs that the DOs select as the LCS (best matches) are shown in blue bold.

The pair (x, y) is classified as a match if the *lcs* between the strings x and y is at least a similarity threshold, $lcs \geq s_t$. The *lcs* of the string pair (x, y) is calculated as:

$$lcs(x, y) = \frac{l_c}{max(l_x, l_y)} = min\left(\frac{l_c}{l_x}, \frac{l_c}{l_y}\right), \tag{7.7}$$

where l_x is the length of the string x, l_y is the length of the string y, and l_c is the length of the longest common sub-string between strings x and y. We illustrate examples of *lcs* calculation and string pair matches between strings in $\mathbf{D}_A$ and $\mathbf{D}_B$ in Figure 7.1.

However, if the DOs send the (encrypted) lengths of their strings, l_x and l_y (as used in Equation 7.7), or the (encrypted) string values x and y (as used in other PPRL approaches [155, 176]) to the LU for comparison, then it is possible that the LU can re-identify these string values x and y of the two DOs. Therefore, in our approaches, the DOs send the (encrypted) lengths of sub-strings in strings, l_v^s, in their databases ($l_x^s \in \mathbf{D}_A$ and $l_y^s \in \mathbf{D}_B$) to the LU, where each l_v^s is calculated as:

$$l_v^s = \frac{l_s}{l_v}, \tag{7.8}$$

where l_v is a length of string value v and l_s is a length of a sub-string, s, that occurs in v.

Therefore, each string value v has a list of l_v^s values, $\mathbf{l}_v^s$, where each of them corresponds to each sub-string s in v ($l_v^s \in \mathbf{l}_v^s \mathrel{\hat{=}} s \in v$). Hence, the *lcs* of the string pair (x, y) in Equation 7.7 can be written as:

$$lcs(x, y) = max(min(l_x^{si} \in \mathbf{l}_x^s, l_y^{sj} \in \mathbf{l}_y^s)), \tag{7.9}$$

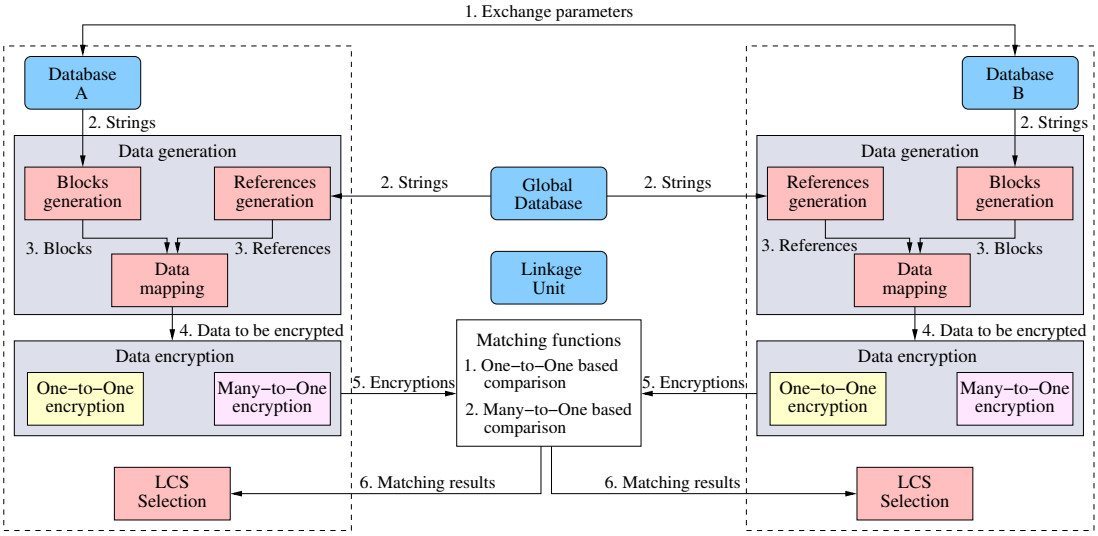

Figure 7.2: Overview protocol of our approaches. The blue boxes are the databases of the DOs, the global database as agreed by the DOs, and the LU. The steps conducted by the DOs are shown in the dashed rectangles, where the red boxes are the common steps of both of our two approaches. The shaded boxes show data generation and data encryption, where data encryption can be either one-to-one (yellow box) as we describe in Section 7.5 or many-to-one (pink box) as we describe in Section 7.6. The comparisons by the LU are shown in the white box.

where l_x^{si} and l_y^{sj} correspond to a common sub-string c in the set of all common sub-strings $\mathbf{c}$ between strings x and y. As the DOs send each of their values l_x^s and l_y^s to the LU to find $min(l_x^s, l_y^s)$ and return the result back to the DOs, the DOs can identify the lcs of a pair of strings (x, y) by finding the highest value ($max()$ in Equation 7.9).

As we illustrate in Figure 7.2, the DOs first make an agreement on the parameter settings to be used in a protocol (either our one-to-one or many-to-one approach) which are:

- The length of q-gram, q, where $q = 2, 3$ are commonly used [38]. This q is to be used for generating q-grams, as we describe in Section 7.4.
- The minimum length of the LCS, m, where $m \geq q$, to be used for selecting those string pairs that have the length of LCS (unnormalised) of at least m, as we describe in Sections 7.4 and 7.7. Similar to Chapter 6, this parameter m can be set based on the length of the strings in the linked databases. For example, linking zip codes should require $m > 1$.
- The similarity threshold, s_t, where $0 < s_t \leq 1$, to be used for generating random values (as we describe in Sections 7.4), and classifying a pair of strings as a match or non-match where the higher s_t (such as $s_t = 0.8$) means two strings in a pair are highly similar (as we describe in Section 7.7).

- The threshold, t, where $0 < t < s_t$, to be used for generating blocks and reference values and generating random values (as we describe in Section 7.4).
- The publicly available global database, **G**, to be used for generating reference values, as we describe in Section 7.4, where values in **G** should be similar to values in the databases to be linked.
- The agreement on which DO has to generate a set of special 1- or 0-encodings, as we describe in Section 7.5.
- The number of decimal places d to be used for converting a decimal number to an integer number, as we describe in Section 7.5.
- The length l to be used for generating binary strings, as we describe in Section 7.5, where l equals the length of the binary string of the number 10^d.
- The hash function [38], $HMAC()$, to be used for encoding special 1- and 0-encodings to hash values, as we describe in Section 7.5.
- The secret salt value s_v to be used for permuting lists of identifiers and l_v^s values, as we describe in Section 7.6.

The DOs then individually generate lists of sub-strings (q-grams) for their strings, where each sub-string (q-gram) is of length q. In our approaches, the LU needs to compare between l_v^s values in a pair where these values correspond to the same sub-string (as we described on page 156). Therefore, in order to allow the LU to find common sub-strings between databases, the DOs use their lists of q-grams to generate blocks, where each blocking key value, *bkv*, represents a sub-string occurring in the strings in their databases. Hence, if the DOs have the same *bkv*, then they have common sub-string(s) (corresponding to the same *bkv*).

However, the common *bkv* values can reveal frequencies of sub-strings in the databases to be linked. Therefore, the DOs use an agreed global database to generate a list of references to hide the frequencies of sub-strings in their databases where this list of references must be the same for the two DOs. Figure 7.3 a) shows examples of *bkv* of $\mathbf{D}_A$ and $\mathbf{D}_B$, and reference values generated from string values in **G**.

To insert an l_v^s value to each generated reference value, if the *bkv* of a block of a DO is the same as a reference value in the generated list of references (there is a common sub-string between a DO's database and the list of references), the DO then finds the maximum value of l_v^s in the block and inserts it into an inverted index (if the DO uses the one-to-one approach) or into a list (if the DO uses the many-to-one approach). For any reference values in the list of references that do not common to any *bkv*s (there is no common sub-string between a DO's database and the list of references), the DO generates random l_v^s values and inserts them into the inverted index or list. We describe the steps of the block and reference generation process in Section 7.4. Figure 7.3 b) shows examples of unencrypted values of $\mathbf{D}_A$ and $\mathbf{D}_B$, and their comparison results.

In our first approach (named one-to-one), each l_v^s in the inverted index is encrypted into a ciphertext, while in our second approach (named many-to-one) all

a)

$\mathbf{D}_A$

String	*bkv*	*sid* : l_v^s
x_1 =	*maar*	x_1^1 : 0.6
"marry"	*arrr*	x_1^2 : 0.6
	rrry	x_1^3 : 0.6
	maarrr	x_1^4 : 0.8
	arrrry	x_1^5 : 0.8
	maarrrry	x_1^6 : 1.0
x_2 =	*amma*	x_2^1 : 0.75
"amar"	*maar*	x_2^2 : 0.75
	ammaar	x_2^3 : 1.0
x_3 =	*maar*	x_3^1 : 0.75
"mary"	*arry*	x_3^2 : 0.75
	maarry	x_3^3 : 1.0

G

String	Reference
"mary"	*maar*
	arry
	maarry
"amar"	*amma*
	maar
	ammaar
"emar"	*emma*
	maar
	emmaar
"marr"	*maar*
	arrr
	maarrr

$\mathbf{D}_B$

String	*bkv*	*sid* : l_v^s
y_1 =	*maar*	y_1^1 : 0.75
"mary"	*arry*	y_1^2 : 0.75
	maarry	y_1^3 : 1.0
y_2 =	*amma*	y_2^1 : 0.6
"amary"	*maar*	y_2^2 : 0.6
	arry	y_2^3 : 0.6
	ammaar	y_2^4 : 0.8
	maarry	y_2^5 : 0.8
	ammaarry	y_2^6 : 1.0
y_3 =	*emma*	y_3^1 : 0.6
"emary"	*maar*	y_3^2 : 0.6
	arry	y_3^3 : 0.6
	emmaar	y_3^4 : 0.8
	maarry	y_3^5 : 0.8
	emmaarry	y_3^6 : 1.0

b)

$\mathbf{D}_A$

bkv	*sid* : l_v^s
maar	x_3^1 : 0.75
arry	x_3^2 : 0.75
maarry	x_3^3 : 1.0
amma	x_2^1 : 0.75
ammaar	x_2^3 : 1.0
emma	x_1^f : 0.6
emmaar	x_2^f : 0.7
arrr	x_1^2 : 0.75
maarrr	x_1^4 : 0.8

Comparison result

Reference	Result
maar	x_3^1, y_1^1 : 0.75
arry	x_3^2, y_1^2 : 0.75
maarry	x_3^3, y_1^3 : 1.0
amma	x_2^1, y_2^1 : 0.75
ammaar	x_2^3, y_2^4 : 0.8
emma	x_1^f, y_3^1 : 0.6
emmaar	x_2^f, y_3^4 : 0.7
arrr	x_1^2, y_1^f : 0.7
maarrr	x_1^4, y_2^f : 0.75

$\mathbf{D}_B$

bkv	*sid* : l_v^s
maar	y_1^1 : 0.75
arry	y_1^2 : 0.75
maarry	y_1^3 : 1.0
amma	y_2^1 : 0.75
ammaar	y_2^4 : 0.8
emma	y_3^1 : 0.6
emmaar	y_3^4 : 0.8
arrr	y_1^f : 0.7
maarrr	y_2^f : 0.75

c)

Match result ($lcs >= 0.8$)

Reference	String x	String y	LCS	lcs
maarry	*"mary"*	*"mary"*	*"mary"*	1.0
ammaar	*"amar"*	*"amary"*	*"amar"*	0.8

Figure 7.3: Example of comparison between (unencrypted) $\mathbf{D}_A$ and $\mathbf{D}_B$, where a) shows *bkv* and reference values generated using the threshold $t = 0.6$. b) shows the maximum l_v^s value corresponding to each reference value and their comparison results. c) shows match results where the similarity threshold $s_t = 0.8$. Red texts show random l_v^s values, blue texts show actual l_v^s values that are less than s_t, and pink texts show actual l_v^s values that are replaced with random l_v^s values to make the numbers of random and actual l_v^s values that are less than s_t equal in a database.

l_v^s values in a list are encrypted into a single ciphertext. The DO then sends its ciphertext(s) to the LU. The LU finds the minimum (encrypted) l_v^s values between the databases and returns them to the DOs. Finally, the DOs can find the (normalised) length of the LCS, *lcs*, for each pair of strings and check if the string pair should be classified as a match (if the *lcs* is at least the agreed similarity threshold). We describe the LCS length selection in Section 7.7 and we show the examples of the (unencrypted) LCS selection in Figure 7.3 c).

We use the same string examples for $\mathbf{D}_A$ and $\mathbf{D}_B$ in Figures 7.1 and 7.3. As can be seen in these figures, the comparison results using the l_v^s and reference values in Figure 7.3 c) are the same as the results of the basic comparison in Figure 7.1 where the DOs send (encrypted) strings to the LU. This implies that the string comparison using the l_v^s and reference values can provide high linkage quality. The privacy of PPRL based on our approaches can also be improved because the DOs do not need to send their (encrypted) strings to the LU. Furthermore, using the reference values (common with *bkv*s) to represent sub-strings can reduce the time complexity used in a comparison process because no position information is required. We describe the analysis of our approaches in Section 7.8.

7.4 Data Generation

Each DO first generates blocks based on sub-strings of strings in its database. Algorithm 7.1 outlines the block generation process by a DO. In line 1, each DO first initialises an inverted index **B** to be used for storing all blocks of its database. In line 2, the DO initialises a set of sub-string identifiers, **sid**, to be used to ensure that all sub-strings in the database have unique identifiers. The DO then loops over each string value v in its database **D** in line 3. For each value, v, in lines 4 and 5 the DO checks if the length of the string, $l_v = |v|$, is at least the agreed minimum length of the LCS, m.

In line 6, the DO generates the list of q-grams, **q**, of v based on the agreed length of q-gram, q, by using the function $genQgramList()$. In line 7, the DO then uses the function $calNumQgram()$ to calculate the minimum number of q-grams, q_m, to be used for generating a blocking key value, *bkv*, where the q_m is calculated as:

$$q_m = \lceil t \times l_v \rceil - q + 1, \tag{7.10}$$

where $\lceil ... \rceil$ denotes rounding to the next upper integer and t is the agreed threshold. In line 8, the DO uses the calculated q_m and the generated list of q-grams **q** as the input to the function $genBKV()$. This function generates a list **bkv** that contains pairs of (bkv, l_s), where each *bkv* represents a sub-string in the corresponding string, and l_s is the length of the sub-string that corresponds to the *bkv*.

For example, we assume the DOs agreed on $t = 0.6$ and $q = 2$. The string $v =$ "*mary*" with the length $l_v = 4$ and q-gram list $\mathbf{q} = [ma, ar, ry]$ results in $q_m = \lceil 0.6 \times 4 \rceil - 2 + 1 = 2$ and sub-lists of this string are $[ma, ar]$, $[ar, ry]$, and $[ma, ar, ry]$. Therefore, the *bkv*s *maar*, *arry*, and *maarry* of this string v represent sub-strings

Algorithm 7.1: ***Block generation process by a DO***

Input:
- **D**: Database
- m: Minimum length of the LCS
- q: Length of q-gram
- t: Threshold

Output:
- **B**: Blocks of lengths of sub-strings in strings (l_v^s)

```
1:  B ← {}                                          // Initialise an inverted index of blocks
2:  sid ← {}                                        // Initialise a set of identifiers
3:  for v ∈ D do:                                   // Loop over string values in a database
4:    l_v ← |v|                                     // Get the length of the string value
5:    if l_v ≥ m do:                                // Check if the length of string is at least m
6:      q ← genQgramList(v, q)                      // Generate q-gram list q
7:      q_m ← calNumQgram(l_v, t, q)                // Calculate minimum number of q-grams for generating a bkv
8:      bkv ← genBKV(q, q_m)                        // Generate list of bkv
9:      for (bkv, l_s) ∈ bkv do:                    // Loop over each bkv and length of sub-string, l_s, in the list
10:       l_v^s ← calLenSubstr(l_s, l_v)            // Calculate the length of a sub-string in the string
11:       sid ← genSubstrID(sid)                    // Generate sub-string identifier that is not in sid
12:       sid.add(sid)                              // Add the sid into the set
13:       if bkv ∉ B do:                            // Check if bkv not exists in B
14:         B[bkv] ← {(sid, l_v^s)}                 // Insert a pair of sid and l_v^s into B as a set
15:       else:                                     // If bkv exists in B
16:         B[bkv].add(sid, l_v^s)                  // Add a pair of sid and l_v^s into the set under the key bkv
17: return B
```

"*mar*", "*ary*", and "*mary*", respectively. As a result, the list $\mathbf{bkv} = [(maar, 3), (arry, 3), (maarry, 4)]$ will be returned from the function $genBKV()$.

In line 9, the DO then loops over each pair (bkv, l_s) in the list **bkv**. For each pair, in line 10, the DO uses the function $calLenSubstr()$ to calculate the length of the sub-string in a string, l_v^s, where this function calculates the l_v^s by using Equation 7.8. In lines 11 and 12, the DO uses the function $genSubstrID()$ to generate a sub-string identifier, *sid*. The function $genSubstrID()$ iteratively generates the *sid* and checks that the generated *sid* must not be in the list **sid** to ensure every l_v^s (sub-string) has a unique sub-string identifier. In line 13, the DO checks if the *bkv* does not exist in the inverted index **B**, it then inserts the *bkv* as a key into **B** and inserts the pair (sid, l_v^s) into a set under the key *bkv* of **B** in line 14. If the *bkv* already exists in **B**, the DO adds a pair (sid, l_v^s) into the set under the corresponding key *bkv* in lines 15 and 16. The DO repeats steps in lines 3 to 16 for every string value v in the database. We show the example of block generation for $\mathbf{D}_A$ and $\mathbf{D}_B$ using the threshold $t = 0.6$ and the length of q-gram $q = 2$ in Figure 7.3 a).

Next, the DO generates a list of references. The DO first conducts a similar process as the block generation process in Algorithm 7.1. For each string value v in the agreed global database, **G**, the DO checks if the length of string l_v is at least m. The DO then generates a list of q-grams, **q**, for this string v and calculates a minimum number of q-grams, q_m, (following Equation 7.10). After that, the DO uses the list **q** and the calculated number q_m to generate reference values. These reference values are generated in the same way as the function $genBKV()$ in Algorithm 7.1 in line 8,

Algorithm 7.2: ***Data generation process by a DO***

Input:
- **B**: Blocks of l_v^s values
- t: Threshold
- **R**: List of references
- s_t: Similarity threshold

Output:
- **T**: Database to be encoded

1: $\mathbf{T} \leftarrow \{\}$ // Initialise an inverted index of data
2: **for** $r \in \mathbf{R}$ **do**: // Loop over reference values in the list **R**
3: **if** $r \in \mathbf{B}$ **do**: // Check if r exists in **B**
4: $(sid, l_v^s) \leftarrow getMaxLSV(\mathbf{B}[r])$ // Get maximum l_v^s corresponding to $bkv = r$ and its identifier
5: **else**: // If r not exists in **B**
6: $(sid, l_v^s) \leftarrow genRand(t, s_t)$ // Generate fake identifier and random l_v^s number
7: $\mathbf{T}[r] \leftarrow (sid, l_v^s)$ // Insert a pair of sid and l_v^s into **T**
8: $\mathbf{T} \leftarrow genProbRand(\mathbf{T})$ // Ensure number of random and actual l_v^s that $l_v^s < s_t$ are equal
9: return **T**

where the length of sub-string l_s will be ignored. The DO then inserts the generated reference values into the list of references, **R**.

From the previous example, the DOs agreed on $t = 0.6$ and $q = 2$. We assume the string $v =$ "*mary*" also exists in the global database **G**. Therefore, the reference values of this string are the same as the *bkv*s in the previous example which are $maar, arry$, and $maarry$. These values are then inserted into the list **R** as $\mathbf{R} = [maar, arry, maarry]$. We show other examples of reference values generated from string values in a global database **G** in Figure 7.3 a).

Once the DO have generated blocks in **B** and a list of reference **R**, the DO generates the database to be encoded. As outlined in Algorithm 7.2, in line 1, the DO initialises a temporary inverted index **T** to store data to be encrypted. The DO loops over a list of reference **R** in line 2. In line 3, the DO then checks if the reference value $r \in \mathbf{R}$ is the same as one of the blocking key values $bkv \in \mathbf{B}$. If $r \in \mathbf{B}$, in line 4, the DO uses the function $getMaxLSV()$ to select a pair of sub-string identifier, sid, and the l_v^s, (sid, l_v^s), where the l_v^s is the maximum value in the block $\mathbf{B}[r]$. If $r \notin \mathbf{B}$, the DO uses the function $genRand()$ to generate a pair of fake sid and random l_v^s values, where the random l_v^s value must be higher than or equals the threshold t and less than the similarity threshold s_t, $t \leq l_v^s < s_t$ in lines 5 and 6. The DO then inserts the pair (sid, l_v^s) into the inverted index **T** under the key r in line 7. The DO repeats steps in lines 2 to 7 until $|\mathbf{T}| = |\mathbf{R}|$.

As we use t to calculate the number of q_m in Equation 7.10, no actual l_v^s will be less than t. In line 8, the DO uses the function $genProbRand()$ to ensure the numbers of actual l_v^s (in the range $[t...s_t]$) and random l_v^s (also in the range $[t...s_t]$) in **T** are equal. This is to make it more difficult for an adversary to identify the frequencies of sub-strings in a database, as we describe next.

As the DOs agreed to use the global database **G** that is similar to their databases for generating a list of references **R**, the number of actual l_v^s with $t \leq l_v^s < s_t$ of each database should be more than the number of random l_v^s in **T**. Therefore, the function

genProbRand() makes the numbers of random and actual l_v^s (that are less than s_t) to be equal by replacing some actual $l_v^s < s_t$ with random l_v^s. The total number of actual l_v^s that are needed to be replaced with random l_v^s is calculated as:

$$n_r = n_t - \left\lceil \frac{n_n + n_t}{2} \right\rceil, \tag{7.11}$$

where n_t is a total number of actual $l_v^s < s_t$ and n_n is a total number of random values in **T**. We show the example of data to be encrypted in Figure 7.3 b) ($\mathbf{D}_A$ and $\mathbf{D}_B$).

In our approaches, we use the threshold t and length of string value l_v to calculate the number q_m (in Equation 7.10) to be used for generating *bkv* and reference values. This is because these t and l_v help to reduce the number of *bkv* and reference values to be generated (especially for long string values) by ignoring the *bkv* and reference values that their corresponding sub-strings are too short and their l_v^s values are less than t. However, we use t instead of the similarity threshold s_t to calculate the number q_m because if we use s_t, the l_v^s values that correspond to common *bkv*s (common sub-strings) in a database will be classified as matches, therefore, an adversary will be able to find the frequencies of sub-strings. By using t and the function *genProbRand*(), there will be equal numbers of actual l_v^s and random l_v^s values in **T** that are not being classified as matches. Therefore, it is ambiguous and more difficult for the adversary to identify the frequencies of sub-strings, thus, difficult to re-identify the original string values in a database. We describe the privacy analysis in detail in Section 7.8.3.

7.5 One-to-One Approach

Our one-to-one approach is based on one l_v^s value encrypted into a ciphertext. This approach consists of five steps: (1) parameter agreement (as described in Section 7.3), (2) data generation (as described in Section 7.4), (3) one-to-one encryption, (4) one-to-one comparison by the LU, and (5) the length of the LCS selection by a DO (as we describe in Section 7.7).

7.5.1 One-to-One Encryption

In this approach, we apply the special 1- and 0-encodings [114] (as we described in Section 7.2.1) to generate the list of hash encodings for each l_v^s value. The special 1- and 0-encodings will allow the LU to conduct a comparison between two encoded values in a pair. However, the comparison function (in Equation 7.3) returns either 1 (number x of DO_A is larger than number y of DO_B) or 0 (number x is less than or equals number y).

If the LU returns a larger or smaller encoded value back to the DOs, the DOs will be unable to select the length of the LCS. Therefore, we also use homomorphic encryption such as Paillier [131], where we assume the two DOs generate the same

Algorithm 7.3: ***One-to-one encryption process by a DO***

Input:
- **T**: Data to be encrypted
- l: Length of binary string
- p_k: Public key
- enc: Agreement to generate either 1- or 0-encodings
- $HMAC()$: Hash function
- d: Number of decimal places

Output:
- **E**: Inverted index of encryptions

1: $\mathbf{E} \leftarrow \{\}$ // Initialise an inverted index of encodings and encryptions
2: **for** $(r, (sid, l_v^s)) \in \mathbf{T}$ **do**: // Loop over data in **T**
3: $rid \leftarrow genRefID(r)$ // Generate reference identifier
4: $n_v^s \leftarrow convert(l_v^s, d)$ // Convert l_v^s to an integer n_v^s
5: $E \leftarrow encrypt(n_v^s, p_k)$ // Encrypt the integer n_v^s into a ciphertext
6: $b \leftarrow genBinary(n_v^s, l)$ // Generate binary string of n_v^s
7: **if** $enc = 1$ **do**: // Check if the DO must do 1-encoding
8: $\mathbf{e}^1 \leftarrow genOneEnc(b)$ // Generate a set of special 1-encodings
9: $\mathbf{H} \leftarrow genHashList(\mathbf{e}^1, r, HMAC())$ // Generate list of hashes of $\mathbf{e}^1$
10: **else**: // If the DO must do 0-encoding
11: $\mathbf{e}^0 \leftarrow genZeroEnc(b)$ // Generate a set of special 0-encodings
12: $\mathbf{H} \leftarrow genHashList(\mathbf{e}^0, r, HMAC())$ // Generate list of hashes of $\mathbf{e}^0$
13: $eid \leftarrow encrypt(sid, p_k)$ // Encrypt the identifier of sub-string sid
14: $hid \leftarrow encrypt(rid, p_k)$ // Encrypt the identifier of reference rid
15: $\mathbf{E}[hid] \leftarrow (eid, E, \mathbf{H})$ // Insert tuple of eid, E, and **H** into a set of $\mathbf{E}[hid]$
16: return **E**

Function $genHashList(\mathbf{e}, r, HMAC())$:

17: $\mathbf{H} \leftarrow [\,]$ // Initialise a list of hashes
18: **for** $e \in \mathbf{e}$ **do**: // Loop over encodings in **e**
19: $h \leftarrow HMAC(e, r)$ // Hash encode the concatenated encoding
20: $\mathbf{H}.append(h)$ // Add hash value into the list **H**
21: return **H**

(public and private) keys and each DO uses a public key to encrypt its l_v^s value into a ciphertext. The DOs then send their lists of hash values and ciphertexts to the LU. Hence, the LU can conduct the comparison using these lists, and return ciphertexts that correspond to the minimum l_v^s values of pairs to the DOs. The DOs can then decrypt the ciphertexts using a private key and select the length of the LCS of string pairs.

Algorithm 7.3 outlines the one-to-one encryption by a DO. In line 1, the DO first initialises the inverted index of encryptions, **E**, to be sent to the LU. In line 2, the DO then loops over the temporary inverted index, **T**, generated in Algorithm 7.2 and uses the function $genRefID()$ to generate an identifier, rid, for each reference value r in line 3. The function $genRefID()$ generates random identifiers by using each reference value r as a seed to a pseudo-random number generator (PRNG) [38]. As a result, the two DOs will generate the same rid for the same reference value r. However, the DOs must ensure that there will be no duplicate rid identifiers.

In line 4, the DO converts the l_v^s (either random or actual l_v^s under the key r in **T**) to an integer value, n_v^s, by using the function $convert()$. This function first converts/rounds the value l_v^s into the agreed d decimal places, $l_v'^s$, and then converts

Table 7.2: Example of n_v^s values and their comparison results based on our one-to-one approach, where $x \in \mathbf{D}_A$ and $y \in \mathbf{D}_B$.

r	sid	n_v^s	Set of encodings, $\mathbf{e}$	Common	$min(n_x^s, n_y^s)$	Return E
maar	x_3^1	65	$\mathbf{e}^1$: {"1000001", "1"}	$\emptyset$	65	E_1
	y_1^1	75	$\mathbf{e}^0$: {"10011'", "101", "11"}			
arry	x_3^2	75	$\mathbf{e}^1$: {"1001011", "100101", "1001", "1"}	{"1001"}	70	E_2
	y_1^2	70	$\mathbf{e}^0$: {"1000111", "1001", "101", "11"}			
maarry	x_3^3	100	$\mathbf{e}^1$: {"11001", "11", "1"}	$\emptyset$	100	E_1
	y_1^3	100	$\mathbf{e}^0$: {"1100101", "110011", "1101", "111"}			
amma	x_2^1	75	$\mathbf{e}^1$: {"1001011", "100101", "1001", "1"}	{"1"}	60	E_2
	y_2^1	60	$\mathbf{e}^0$: {"0111101", "011111", "1"}			
ammaar	x_2^3	100	$\mathbf{e}^1$: {"11001", "11", "1"}	{"11"}	80	E_2
	y_2^1	80	$\mathbf{e}^0$: {"1010001", "101001", "10101", "1011", "11"}			

it to an integer by calculating $n_v^s = l_v'^s \times 10^d$. The number of decimal places d is used to ensure the two DOs will generate their integer numbers in the same range $[0...10^d]$. In line 5, the DO then encrypts the integer number n_v^s into a ciphertext E by using the function $encrypt()$ and public key p_k. We convert the l_v^s into an integer value n_v^s because we apply Paillier homomorphic encryption [131] in the function $encrypt()$ and generate a binary string for the special 1- and 0-encodings [114]. Therefore, the integer value is simple and suitable to be used in this approach.

In line 6, the DO converts the value n_v^s into a binary string b of length l by using the function $genBinary()$. Then, in lines 7 to 12, if the DO has agreed to do the 1-encoding, it uses the function $genOneEnc()$ to encode its binary string b into a set of 1-encodings, $\mathbf{e}^1$, where each $e^1 \in \mathbf{e}^1$ is generated following Equation 7.1. If the DO has agreed to do the 0-encoding, the DO uses the function $genZeroEnc()$ to encode its b into a set of 0-encodings, $\mathbf{e}^0$, where each $e^0 \in \mathbf{e}^0$ is generated following Equation 7.2. Table 7.2 (the fourth column) illustrates some examples of the sets $\mathbf{e}^1$ and $\mathbf{e}^0$ generated for $\mathbf{D}_A$ and $\mathbf{D}_B$ from Figure 7.3 b), respectively. The DO then uses the function $genHashList()$ to generate a list of hash values, $\mathbf{H}$, for its set of encodings $\mathbf{e}$ ($\mathbf{e}^1$ or $\mathbf{e}^0$). The details of this function $genHashList()$ are provided in lines 17 to 21.

The function $genHashList()$ first initialises the list of hash values, $\mathbf{H}$, in line 17, and then loops over the encodings in the set of encodings $\mathbf{e}$ in line 18. In line 19, the encoding e and the reference value r are used in the function $HMAC()$ to encode e to a hash value h. In the function $HMAC()$, the encoding e is concatenated with the reference value r into a single string e'. This r is used as a salting value [129] because it is possible that different binary strings that are in different blocks can be encoded into the same (1- or 0-) encodings, and therefore using r as a salting value can reduce the frequency distribution of each encoding e. Thus, it is more difficult for the LU to analyse the frequencies of n_v^s. The concatenated string, e', is then hash encoded by a one-way hash function (we use $SHA256$ [154]), resulting in a hash value h. This hash value h is then added to a list of hash values $\mathbf{H}$ in line 20. The steps in lines 18 to

Algorithm 7.4: ***Comparison of one-to-one encryption by the LU***

Input:
- $\mathbf{E}_1$: Inverted index of encryptions from the first DO
- $\mathbf{E}_2$: Inverted index of encryptions from the second DO

Output:
- $\mathbf{M}$: Inverted index of compared results

1: $\mathbf{M} \leftarrow \{\}$ // Initialise an inverted index of compared results
2: $\mathbf{E}_C \leftarrow \mathbf{E}_1 \cap \mathbf{E}_2$ // Find common identifiers
3: **for** $hid \in \mathbf{E}_C$ **do**: // Loop over common identifiers
4: $(eid_1, E_1, \mathbf{H}_1) \leftarrow \mathbf{E}_1[hid]$ // Get values from the first DO
5: $(eid_2, E_2, \mathbf{H}_2) \leftarrow \mathbf{E}_2[hid]$ // Get values from the second DO
6: **if** $\mathbf{H}_1 \cap \mathbf{H}_2 = \emptyset$ **do**: // Check if common hashes not exist
7: $min \leftarrow E_1$ // Select the ciphertext of the first DO as a minimum value
8: **else**: // If common hashes exist
9: $min \leftarrow E_2$ // Select the ciphertext of the second DO as a minimum value
10: $\mathbf{M}[(eid_1, eid_2)] \leftarrow min$ // Insert compared result to $\mathbf{M}$
11: return $\mathbf{M}$

20 are repeated until every encoding $e \in \mathbf{e}$ has been hashed and added to the list $\mathbf{H}$. Finally, the function returns $\mathbf{H}$ in line 21.

Back to the main program, where in line 13 the DO encrypts the *sid* (either actual or fake identifier) into a ciphertext *eid* by using the function $encrypt()$. The DO also encrypts the reference identifier *rid* into a ciphertext *hid* by using the same function $encrypt()$ in line 14. The DO then adds the encrypted identifier, *eid*, the ciphertext E, and the list of hash values $\mathbf{H}$ into the inverted index $\mathbf{E}$ under the key *hid* in line 15. The DO repeats steps in lines 2 to 15 until every reference value $r \in \mathbf{T}$ has been encoded and encrypted. Finally, in line 16, each DO sends its inverted index $\mathbf{E}$ to the LU.

7.5.2 One-to-One Comparison

As described in Section 7.3, the LU aims to find a minimum (smaller) value between ciphertexts in a pair because the minimum value is possible to be the length of the LCS between strings that contain the sub-string corresponded to the encryption of l_v^s (in this approach it is an integer n_v^s). The LU receives the inverted indexes $\mathbf{E}_1$ and $\mathbf{E}_2$ from the two DOs. As we outline in Algorithm 7.4, the LU first initialises the inverted index of compared results, $\mathbf{M}$, in line 1. The LU finds common identifiers $\mathbf{E}_C$ (encrypted identifiers of reference values) between $\mathbf{E}_1$ and $\mathbf{E}_2$ in line 2 and loops over them in line 3. The LU then extracts the corresponding identifiers (eid_1 and eid_2), ciphertexts (E_1 and E_2), and the lists of hash values ($\mathbf{H}_1$ and $\mathbf{H}_2$) from $\mathbf{E}_1$ and $\mathbf{E}_2$ for each identifier $hid \in \mathbf{E}_C$ in lines 4 and 5.

In lines 6 to 9, the LU finds common hash values between the lists $\mathbf{H}_1$ and $\mathbf{H}_2$ based on Equation 7.3, as we described in Section 7.2.1. If there are any common hashes, the LU selects the ciphertext E_2 as the minimum value min. If there is no common hash value, the LU selects the ciphertext E_1 as the minimum value min. The

Algorithm 7.5: ***Many-to-one encryption by a DO***

Input:
- **T**: Data to be encrypted
- p_k: Public key
- s_v: Secret salt value

Output:
- **eid**: List of encrypted identifiers
- E: Ciphertext

1: **eid** $\leftarrow$ [ ] // Initialise the list of identifiers
2: $\mathbf{l}_v^s \leftarrow$ [ ] // Initialise the list of l_v^s
3: $\mathbf{T}' \leftarrow sort(\mathbf{T})$ // Sort the inverted index **T** based on keys (reference values)
4: **for** $(r, (sid, l_v^s)) \in \mathbf{T}'$ **do**: // Loop over data in $\mathbf{T}'$
5: $eid \leftarrow encrypt(sid, p_k)$ // Encrypt the identifier *sid*
6: **eid**$.append(eid)$ // Add *eid* to the list
7: $\mathbf{l}_v^s.append(l_v^s)$ // Add l_v^s to the list
8: **eid**, $\mathbf{l}_v^s \leftarrow permute(\mathbf{eid}, \mathbf{l}_v^s, s_v)$ // Permute the lists **eid** and $\mathbf{l}_v^s$
9: $E \leftarrow packEncrypt(\mathbf{l}_v^s, p_k)$ // Encrypt list $\mathbf{l}_v^s$ into a ciphertext
10: return **eid**, E

LU generates a pair of identifiers (eid_1, eid_2), then inserts the generated pair as a key and inserts *min* as the value into the inverted index **M** in line 10. The LU repeats the steps in lines 3 to 10 until all encryptions corresponding to all $hid \in \mathbf{E}_C$ have been compared. Finally, the LU returns the compared results **M** to the DOs in line 11. We illustrate examples of the comparison results (the last three columns) in Table 7.2.

7.6 Many-to-One Approach

Our PPRL approach based on multiple (many) l_v^s values encrypted into one ciphertext approach consists of five steps: (1) parameter agreement (as described in Section 7.3), (2) data generation (as described in Section 7.4), (3) many-to-one encryption, (4) many-to-one comparison by the LU, and (5) the length of the LCS selection by a DO as we describe in Section 7.7.

7.6.1 Many-to-One Encryption

In this approach, we apply the packing encryption method proposed by Cheon et al. [26] to generate one ciphertext for multiple (many) l_v^s values, where we assume the two DOs agree to use the same security parameters which are used for generating the same public and private keys pair. As described in Section 7.2.2, the packing method allows multiple decimal numbers to be encrypted. Therefore, we can encrypt the l_v^s values without converting them into integer values as needed in our one-to-one approach (as we described in Section 7.5).

Algorithm 7.5 outlines the many-to-one encryption by a DO. In lines 1 and 2, the DO initialises the list of identifiers, **eid**, and list of l_v^s values, $\mathbf{l}_v^s$, respectively. The DO then sorts the inverted index **T** based on reference values (keys of **T**) in alphabetical order, resulting in the inverted index $\mathbf{T}'$ in line 3. This sorting step ensures that both

Table 7.3: Example of (unencrypted and non-permuted) lists of identifiers and l_v^s values, and their comparison results based on our many-to-one approach. The pairs with $lcs \geq s_t$, where $s_t = 0.8$, are classified as matches.

Sorted $\mathbf{R}$	$[amma, ammaar, arrr, arry, emma, emmaar, maar, maarrr, maarry]$	
$\mathbf{T}'_A$	**sid** : $[x_2^1, x_2^3, x_1^2, x_3^2, x_1^f, x_2^f, x_3^1, x_1^4, x_3^3]$	$\mathbf{l}_x^s$: $[0.75, 1.0, 0.6, 0.75, 0.6, 0.7, 0.65, 0.8, 1.0]$
$\mathbf{T}'_B$	**sid** : $[y_2^1, y_2^4, y_1^f, y_1^2, y_3^1, y_3^4, y_1^1, y_2^f, y_1^3]$	$\mathbf{l}_y^s$: $[0.6, 0.8, 0.7, 0.7, 0.6, 0.8, 0.75, 0.75, 1.0]$
Comparison result	**pid** : $[(x_2^1, y_2^1), (x_2^3, y_2^4), (x_1^2, y_1^f), (x_1^f, y_3^1), (x_2^f, y_3^4), (x_3^1, y_1^1), (x_1^4, y_2^f), (x_3^3, y_1^3)]$	min : $[0.6, 0.8, 0.6, 0.7, 0.6, 0.7, 0.65, 0.75, 1.0]$
Matches	**pid** : $[(x_2^3, y_2^4), (x_3^3, y_1^3)]$	$lcs \geq s_t$: $[0.8, 1.0]$

DOs will insert their l_v^s values that correspond to the same sub-strings in the same order, and therefore the LU will compare ciphertexts and return correct results to the DOs. In line 4, the DO loops over reference values and pairs (sid, l_v^s) in the sorted $\mathbf{T}'$. The DO then encrypts the *sid* into a ciphertext *eid* by using the function $encrypt()$ in line 5. The DO adds the *eid* into the list **eid** and adds the l_v^s into the list $\mathbf{l}_v^s$ in lines 6 and 7, respectively. The DO repeats the steps in lines 4 to 7 until the list **eid** contains numbers of *eid* equal to the number of reference values in $\mathbf{T}'$, $|\mathbf{eid}| = |\mathbf{T}'|$. We show examples of $\mathbf{T}'_A$ and $\mathbf{T}'_B$ generated from the string values in $\mathbf{D}_A$ and $\mathbf{D}_B$ from Figure 7.3 b) in Table 7.3.

In line 8, the DO permutes the lists **eid** and $\mathbf{l}_v^s$ by using the agreed secret salt value s_v as a seed to hide the original positions of values in the two lists, while each element (*eid* value and its corresponding l_v^s) in the lists still refers to the same sub-string. The DO encrypts the permuted list $\mathbf{l}_v^s$ into a ciphertext E by using the function $packEncrypt()$ [26] in line 9. Finally, in line 10, the DO sends the list **eid** and the ciphertext E to the LU for comparison.

7.6.2 Many-to-One Comparison

The LU receives the lists of identifiers ($\mathbf{eid}_1$ and $\mathbf{eid}_2$) and ciphertexts (E_1 and E_2) from the two DOs. As we outline in Algorithm 7.6, the LU first initialises the list of pairs of identifiers **pid** in line 1. The LU then loops over the indexes of the list of identifiers $\mathbf{eid}_1$, where $|\mathbf{eid}_1| = |\mathbf{eid}_2| = |\mathbf{T}|$ as generated in Algorithm 7.5. In lines 3 and 4, for each index i in the lists $\mathbf{eid}_1$ and $\mathbf{eid}_2$, the LU extracts the encrypted identifiers eid_1 and eid_2, respectively. The LU then generates a pair of identifiers, *pid*, in line 5, and inserts this pair into the list **pid** in line 6. The LU repeats the steps in lines 2 to 6 until the length of the list **pid** equals the length of $\mathbf{eid}_1$, $|\mathbf{pid}| = |\mathbf{eid}_1|$. In line 7, the LU then uses the function $compareCip()$ to find the minimum values, *min*, between ciphertexts [27]. This function conducts a comparison following Equations 7.4 to 7.6, as we described in Section 7.2.2. Finally, in line 8, the LU returns the generated list of identifier pairs, **pid**, and the compared result *min* to the DOs. We show an example of many-to-one comparison results in Table 7.3.

Algorithm 7.6: ***Comparison of many-to-one encryption by the LU***

Input:
- $\mathbf{eid}_1$: List of encrypted identifiers from the first DO
- E_1: Ciphertext from the first DO
- $\mathbf{eid}_2$: List of encrypted identifiers from the second DO
- E_2: Ciphertext from the second DO

Output:
- **pid**: List of pairs of identifiers
- *min*: Compared result

1: $\mathbf{pid} \leftarrow [\,]$ // Initialise a list of identifier pairs
2: **for** $i = 0$ to $|\mathbf{eid}_1| - 1$ **do**: // Loop over each index in the list of identifiers
3: $eid_1 \leftarrow \mathbf{eid}_1[i]$ // Get an encrypted identifier from $\mathbf{eid}_1$
4: $eid_2 \leftarrow \mathbf{eid}_2[i]$ // Get an encrypted identifier from $\mathbf{eid}_2$
5: $pid \leftarrow (eid_1, eid_2)$ // Generate pair of identifiers
6: $\mathbf{pid}.append(pid)$ // Add the generated pair of identifiers to the list
7: $min \leftarrow compareCip(E_1, E_2)$ // Compare ciphertexts
8: return $\mathbf{pid}, min$

7.7 LCS Length Selection

As shown in Figure 7.2, in the last step of our approaches, each DO selects the length of the LCS, *lcs*, based on the results they received from the LU. If the DOs follow the one-to-one approach, they will use their private keys to decrypt each ciphertext in $\mathbf{M}$ to an integer n_v^s, and then convert n_v^s to l_v^s (decimal number in the range [0...1]). If DOs follow the many-to-one approach, they will use their private keys to decrypt a ciphertext E to multiple l_v^s values.

Each DO then extracts each pair of identifiers and selects the string identifier from its database that corresponds to its *eid* in the identifier pair. Once the DO has all l_v^s values for each of its strings, the DO finds the highest l_v^s value for the string which is the length of the LCS, *lcs*, between the pair of strings (as we described in Section 7.3). Finally, the DO only keeps the string pairs (corresponding to its string identifier and the sub-string identifier of the other DO) that have an $lcs \geq s_t$ and their corresponding length of (common) sub-string of at least m as matches.

The fake identifiers and random l_v^s values will not be selected because (1) there is no string identifier in the DO's database that corresponds to the fake *eid*, and (2) the random l_v^s value was generated by ensuring it is lower than the similarity threshold, s_t, as we described in Section 7.4. Figure 7.3 c) shows examples of the string pairs that are selected as matches.

7.8 Analysis

In this section, we analyse our proposed approaches in terms of linkage quality, scalability, and privacy.

7.8.1 Linkage Quality Analysis

The publicly available global database, **G**, is the major factor to determine the linkage quality of our approaches. This is because the sub-strings of string values in **G** must be similar to the sub-strings in the databases $\mathbf{D}_A$ and $\mathbf{D}_B$ to be linked to ensure common sub-strings of the two DOs will be compared. If there are sub-strings in $\mathbf{D}_A$ and $\mathbf{D}_B$ that are not in **G**, then some common sub-strings between $\mathbf{D}_A$ and $\mathbf{D}_B$ will not be compared because these common sub-string values have no reference values and will not be encrypted and sent to the LU for comparison. Therefore, this can lead to more false non-matches (false negatives) which leads to lower recall.

As we described in Section 7.3, the two DOs aim to find the length of the longest common sub-string (LCS), *lcs*, between a pair of string values (calculated using Equation 7.7) based on the lengths of sub-strings in strings (calculated using Equation 7.8), l_v^s. When the l_v^s values in a pair correspond to the same sub-string (reference value), the minimum value of l_v^s of the pair that is found by the LU and returned to the DOs can be the length of the LCS. According to Equation 7.9, the DOs can select the length of the LCS by finding the highest value of l_v^s that correspond to all sub-strings of their strings in this pair. Therefore, the blocking key values, *bkv*s, and the reference values in the list **R** must be generated in the same manner (as we described in Section 7.4) to ensure that the l_v^s values that correspond to the same sub-strings in the database of each DO will correspond to the same reference values and will be compared.

The sorting of reference values and the same permutation of the lists **eid** and $\mathbf{l}_v^s$ are important in our many-to-one approach because they ensure that the DOs will insert their l_v^s values in the same order. Without using the sorted list of references and the same permutations, the two l_v^s values that correspond to not common sub-strings will be compared. As a result, the DOs will get incorrect match results.

For example, the two DOs agree on the similarity threshold $s_t = 0.8$. We assume the first DO has the list of references $\mathbf{R}_A = [amma, maarry, ammaar]$ and the list of l_v^s values $\mathbf{l}_x^s = [0.75, 1.0, 1.0]$. The second DO has the list of references $\mathbf{R}_B = [maarry, ammaar, amma]$ and the list of l_v^s values $\mathbf{l}_y^s = [1.0, 0.8, 0.6]$. The comparison result between $\mathbf{l}_x^s$ and $\mathbf{l}_y^s$ is $[0.75, 0.8, 0.6]$ which is incorrect because the first DO will identify that its value *maarry* has $lcs = 0.8$ and its value *ammaar* has $lcs = 0.6$ (classified as a non-match), the second DO will identify that its value *maarry* has $lcs = 0.75$ (classified as a non-match) and its value *ammaar* has $lcs = 0.8$, while the correct result between their values are $lcs = 0.6$ for *amma*, $lcs = 0.8$ for *ammaar*, and $lcs = 1.0$ for *maarry*. Thus, the values *ammaar* (of the first DO) and *maarry* (of the second DO) are false negatives. Examples of the correct results are shown in Figure 7.3 c).

The DOs add fake identifiers and random l_v^s values for values in their lists of references that differ from the sub-strings (*bkv*s) in their databases. These fake and random values do not affect the linkage quality because they are less than the similarity threshold, s_t, and their fake identifiers are not in the databases of the DOs. As a result, these random l_v^s values cannot be classified as matches.

With using homomorphic encryption [26, 131], there can be errors (noise) that are caused by any process such as the DOs adding noise to improve security, the encryption/decryption process, more operations needed to be conducted over ciphertexts, or even rounding of plaintext values before conducting the encryption process [26]. When using the packing encryption method suggested by Cheon et al. [27], errors can also be caused by conducting multiple arithmetic operations over ciphertexts. When using a comparison function that provides less errors [27], such as providing a maximum error of 2^{-5}, the comparison process will consume more runtimes. Therefore, Cheon et al. [27] suggested the optimal functions that minimise the total computation complexity and causes the maximum error of $2^{-4} \approx 6.25\%$ errors in the compared results, as we described in Section 7.2.2.

However, the errors that are caused by conducting arithmetic operations over ciphertexts often occur when two ciphertexts are the encryption of very close (or the same) numerical values [27]. Therefore, in our approaches, if the errors are not between 0.0 and 1.0 (errors are large or negative numbers [28]), the DOs can select their l_v^s values as the minimum values (compared results) because the errors imply that the values of the two DOs are very close or the same. For errors that are between 0.0 and 1.0, which can occur in both the homomorphic and packing encryption methods, these errors can lead to false matches and false non-matches. We show errors in our experimental results in Section 7.9.2.

7.8.2 Scalability Analysis

We now analyse the scalability of the main steps of our approaches. The first step performed by the DOs is the data generation step which includes the generation of blocks, the list of reference values, and the data to be encrypted. To generate the inverted index of blocks, $\mathbf{B}$, each DO requires a complexity of $O(|\mathbf{D}|)$ for extracting string values and generating the list of q-gram, $\mathbf{q}$, for each string value v, where $|\mathbf{D}|$ is the number of strings in the database $\mathbf{D}$. For each $\mathbf{q}$, the DO requires $O(|\mathbf{q}|)$ time complexity for generating the list of blocking key values, $\mathbf{bkv}$, and for each $\mathbf{bkv}$, the DO then requires $O(|\mathbf{bkv}|)$ time complexity to calculate the l_v^s values that correspond to the sub-strings of all *bkv* in the list $\mathbf{bkv}$. Therefore, overall the block generation process requires a complexity of $O(|\mathbf{D}| \times |\mathbf{q}| \times |\mathbf{bkv}|)$, where we assume $|\mathbf{q}|$ is the average length of the q-gram lists of strings in $\mathbf{D}$, and $|\mathbf{bkv}|$ is the average number of *bkv* of strings in $\mathbf{D}$.

To generate a list of references, $\mathbf{R}$, each DO requires $O(|\mathbf{G}|)$ time complexity for extracting string values from the global database $\mathbf{G}$ and generating a list of q-grams, $\mathbf{q}$. Similar to the block generation process, each DO requires $O(|\mathbf{q}|)$ time complexity to generate reference values and insert them into the list of references $\mathbf{R}$. Overall, the DO requires a maximum $O(|\mathbf{G}| \times |\mathbf{q}|)$ complexity for generating $\mathbf{R}$.

To generate the data to be encrypted, $\mathbf{T}$, the DO requires $O(|\mathbf{R}|)$ complexity to loop over the list of references $\mathbf{R}$. The DO then requires $O(|\mathbf{B}|)$ complexity for checking if a reference value r in $\mathbf{R}$ exists in $\mathbf{B}$. If $r \in \mathbf{B}$, the DO requires $O(|\mathbf{B}[r]|)$ complex-

ity to find the maximum l_v^s value in the block $\mathbf{B}[r]$. After that, the DO inserts values into $\mathbf{T}$ for which the DO requires a time complexity of $O(|\mathbf{T}|)$ to find the number k of actual l_v^s that are needed to be replaced with random l_v^s, and requires $O(k)$ complexity for the replacing value step. Overall, the DO requires $O((|\mathbf{R}| \times |\mathbf{B}| \times n) + |\mathbf{T}| + k)$, where we assume n is the average number of elements of all blocks $\mathbf{B}[r]$ in $\mathbf{B}$.

In the one-to-one encryption process by a DO, each DO first loops over $\mathbf{T}$, and therefore it requires $O(|\mathbf{T}|)$ complexity. The DO then generates 1- or 0-encodings and hash encodes the generated encodings into hash values which requires $O(|\mathbf{e}|)$, where $\mathbf{e}$ is the set of 1- or 0-encodings. Therefore, overall the DO requires a time complexity of $O(|\mathbf{T}| \times |\mathbf{e}|)$, where we assume $|\mathbf{e}|$ is the average number of elements of all $\mathbf{e}$.

In the one-to-one comparison step conducted by the LU, the LU requires $O(|\mathbf{E}_C|)$ complexity for finding the common identifiers (*hid*) between two inverted indexes, where $|\mathbf{E}_C| = |\mathbf{T}|$ because the two DOs generate their inverted indexes $\mathbf{E}_1$ and $\mathbf{E}_2$ based on $\mathbf{T}$. The LU then loops over all common $hid \in \mathbf{E}_C$ which again requires $O(|\mathbf{E}_C|)$ complexity. For a pair of lists of hash values under each key *hid* in the two inverted indexes, the LU requires a time complexity of $O(|\mathbf{H}|)$ to find common hash values, where we assume the two lists of hash values $\mathbf{H}_1$ and $\mathbf{H}_2$ have the same number of elements (hash values), $|\mathbf{H}| = |\mathbf{H}_1| = |\mathbf{H}_2|$. Overall, the LU requires a time complexity of $O(|\mathbf{E}_C| + (|\mathbf{E}_C| \times |\mathbf{H}|))$.

In the many-to-one encryption process by a DO, each DO first sorts $\mathbf{T}$ based on the reference value (keys in $\mathbf{T}$) resulting in the sorted inverted index $\mathbf{T}'$. In this step, the DO requires the worst time complexity of $O(|\mathbf{T}|^2)$, while the best time complexity is $O(|\mathbf{T}|\ log\ |\mathbf{T}|)$. The DO then loops over $\mathbf{T}'$ to generate the lists of identifiers and l_v^s values, $\mathbf{eid}$ and $\mathbf{l}_v^s$, respectively. In this step, the DO requires $O(|\mathbf{T}'|)$ complexity. The DO then requires $O(|\mathbf{eid}|)$ and $O(|\mathbf{l}_v^s|)$ for permuting the lists $\mathbf{eid}$ and $\mathbf{l}_v^s$, where these lists have the same number of elements, $|\mathbf{eid}| = |\mathbf{l}_v^s|$. Overall the DO has a worst-case complexity of $O(|\mathbf{T}|^2 + |\mathbf{T}'| + |\mathbf{eid}| + |\mathbf{l}_v^s|)$.

In the many-to-one comparison step, the LU receives the lists of identifiers $\mathbf{eid}_1$ and $\mathbf{eid}_2$, and ciphertexts E_1 and E_2 from the two DOs. Overall, the LU requires a complexity of $O(|\mathbf{eid}|)$ which is needed for generating pairs of identifiers, where $|\mathbf{eid}| = |\mathbf{eid}_1| = |\mathbf{eid}_2| = |\mathbf{T}|$. This is because the LU only requires $O(1)$ for conducting a comparison between ciphertexts E_1 and E_2.

7.8.3 Privacy Analysis

We now analyse what the parties involved in our approaches can learn from the data they receive from each other. We assume the DOs and the LU follow the honest-but-curious (HBC) adversary model (as we described in Section 2.2.2) where no DO colludes with the LU [116]. In our approaches, the DOs first communicate with each other to agree on parameter settings. This allows them to learn the parameters that are being used in the encryption processes, but they cannot learn any sensitive information about the strings in each other's databases.

In both of our approaches, the DOs individually generate the blocks of their databases. They cannot learn any information from the other database in this step. The DOs then individually generate their lists of references (**R**) based on the agreed global database **G**. This allows the DOs to know sub-strings (reference values) in the list of references **R** which are common between DOs because they generate **R** in the same manner. After that, the DOs generate the data to be encrypted, and generate random lengths of sub-strings in strings, l_v^s, values and their identifiers.

As we use the threshold t to generate reference and *bkv* values, where t is less than the similarity threshold s_t, $t < s_t$, there will be some l_v^s values that will not be classified as matches (actual l_v^s where $t \leq l_v^s < s_t$). We replace some of these actual l_v^s values with random l_v^s to make them have equal frequencies in the database to be encrypted. Therefore, the adversary cannot identify the correct frequencies of sub-strings. If the two DOs generate some common random l_v^s values, they do not affect the privacy of our approaches because the DOs do not know each other's random values.

In our approaches, as we use the list of references **R** and the l_v^s values (both actual and random values), no sensitive information such as the length of sub-strings and sub-string (and string) values will be revealed to the LU. We assume the worst-case where the LU uses any key attacks, such as the attack proposed by Li and Micciancio [112] which successfully identifies the private key used in CKKS homomorphic encryption proposed by Cheon et al. [26] (which we use Cheon et al. [26] packing method in our many-to-one approach). It is still uncertain for the LU to correctly analyse the frequencies of sub-strings and re-identify the sub-string or string values of the DOs. This is because (1) each l_v^s value that the LU receives from a DO can exist (the DO sends its actual l_v^s) or not (the DO sends random l_v^s) in the DO's database, (2) the LU does not know the lengths of the DO's strings, and therefore the LU cannot calculate and learn the length of sub-strings of the DO, and (3) the LU does not know the sub-string values and length of sub-strings or length of strings of the DO, and therefore it is difficult to analyse original string values. The LU can re-identify original string values of the DO if and only if the LU knows all parameter settings that the DO used, a database of the DO, and a global database used for generating a list of references.

In the one-to-one approach, the LU receives inverted indexes from the DOs. The LU then finds common identifiers between these inverted indexes. This allows the LU to learn the length of the list of references **R**, but it does not allow the LU to learn any sensitive information of the DOs' databases because all identifiers are in common as the DOs generated them based on the same list of references **R**. The LU does learn which lists of hash values have common hashes, but it cannot learn the original values encoded in them. Furthermore, the DOs use the function $HMAC()$, where the reference values are used as secret values, and therefore it avoids the LU to be able to conduct dictionary attacks.

The LU can count the number of lists of hashes that correspond to common sub-string values but this does not allow the LU to conduct any frequency analysis cor-

rectly because the two inverted indexes of the DOs contain both actual and random l_v^s values where the number of actual $l_v^s < s_t$ equals the number of random l_v^s values. Therefore, the LU is unable to know which list of hash values contains encodings of actual or random l_v^s values.

For the many-to-one approach, the LU receives the lists of identifiers, $\mathbf{eid}_1$ and $\mathbf{eid}_2$, and ciphertexts E_1 and E_2 from the two DOs. Similar to the one-to-one approach, although the LU can count the number of identifiers which equals the number of reference values in $\mathbf{R}$, the LU cannot learn the number of sub-strings (that corresponded to the actual l_v^s values) of the DOs because the LU does not know which *eid* is the identifier of actual or random l_v^s value.

The LU returns the compared results (ciphertexts of minimum l_v^s values) to the DOs. The DOs then decrypt the ciphertexts and select the length of the LCS of string pairs. The DOs can learn the length of the LCS between a string in their database and a string in the other database, but they cannot learn the positions where the LCS occurs in strings of the other database. In both of our approaches, each DO can count the number of its common sub-strings by excluding a number of its random l_v^s values. However, the DO cannot learn the frequency of sub-strings in the other database because the DO does not know which sub-strings of the other DO correspond to the actual or random l_v^s value as the number of these l_v^s values has the same (or almost the same) frequencies in the comparison results. This is because both DOs replace their actual $l_v^s < s_t$ by random l_v^s values to make the frequencies equal. Therefore, the comparison results will contain the same frequencies between actual ($l_v^s < s_t$) and random l_v^s.

The DO can learn some string lengths of the other DO from the compared results (minimum l_v^s values) returned from the LU. As a DO knows the common sub-string (reference value) that the l_v^s value corresponded to, it can learn the length of the sub-string, l_s, of the other DO. If the l_v^s returned from the LU is the number of the other DO, then the DO can calculate the length of the string of the other DO as l_s/l_v^s. For example, the DO_A knows that the sub-string is "*mary*" with $l_s = 4$ and the l_v^s is 0.8 where this $l_v^s = 0.8$ is not its l_v^s as DO_A has $l_v^s = 1.0$. DO_A calculates $4/0.8 = 5$. Therefore, DO_A learns that the length of the string of the other DO (DO_B) is 5 and this string contains the sub-string "*mary*".

Although DO_A would not know the actual string and where this sub-string "*mary*" occurs in the database of DO_B, DO_A can use any publicly available database and the calculated number (as we described above) to analyse the possible string of DO_B. However, it would be time consuming to do this analysis because there are many words or sequences of numbers that contain the same sub-strings, where the smaller threshold t used for generating blocks and reference values makes it more difficult to correctly identify the string of DO_A. This is because smaller t results in larger numbers of blocks and reference values.

The list of references $\mathbf{R}$ and random l_v^s values with equal frequencies as the actual $l_v^s < s_t$ also make it more difficult (increase uncertainty) for each DO to correctly re-identify the original sub-strings and strings in the database of the other DO. For

example, assume the DO_A and DO_B agreed on the similarity threshold $s_t = 0.8$. There are two common sub-strings (reference values) $x =$ "*arr*" and $y =$ "*marr*", where x is a sub-string of y. The minimum l_v^s values returned from the LU are 0.7 for the sub-string x and 0.75 for the sub-string y. Therefore, the pair of l_v^s values corresponding to the two sub-strings x and y are classified as non-matches.

We assume DO_A has the sub-string x with $l_v^s = 0.6$ but it replaces the l_v^s with random $l_v^s = 0.7$, and DO_A has the sub-string y with $l_v^s = 0.8$. We assume DO_B does not have any of x and y, it adds random $l_v^s = 0.7$ for x and $l_v^s = 0.75$ for y. The LU conducts comparison and returns $l_v^s = 0.7$ for x and $l_v^s = 0.75$ for y. DO_B cannot know if the DO_A does have or does not have the sub-strings x and y in the database. This is because it can be:

1. DO_A does not have both x and y, therefore, DO_A adds random l_v^s values.
2. DO_A has both x and y but their corresponding l_v^s values are higher than the values of DO_B, therefore, the LU returns the l_v^s values of DO_B.
3. DO_A has either x or y, while one of their corresponding l_v^s values is random and one is higher than the value of DO_B.
4. DO_A has both x and y where their corresponding l_v^s values equal the values of DO_B.
5. DO_A has both x and y but it replaces one of the l_v^s values with a random l_v^s value while the other l_v^s value equals the value of DO_B.
6. DO_A has both x and y but it replaces one of the l_v^s values with a random l_v^s value while the other l_v^s value is higher than the value of DO_B.

7.9 Experimental Evaluation

We evaluated the linkage quality, scalability, and privacy of our approaches compared to Bloom filter (BF) encoding [155], and our proposed shifted hash encoded q-gram based (named ShiftedHash) and bit array based (named BitArray) approaches [176], as we described in Chapter 6. We compared our approaches with these baselines because BF encoding [155] is considered as a standard technique for PPRL, and our ShiftedHash and BitArray approaches provide accurate string matching results.

7.9.1 Datasets and Experimental Setup

In our evaluation, we used the same real-world dataset pairs as we used in Chapter 6. For the global datasets used for generating reference values, we extracted 100,000 real-world string values of first names, cities, zip codes, and telephone numbers from the North Carolina Voter Registration[1] (NCVR) from the snapshot of 2016.

[1] http://dl.ncsbe.gov/

Table 7.4: Percentages of errors in our approaches for different datasets and blocking techniques (our technique and the original q-gram based blocking [38]).

Dataset	One-to-One		Many-to-One	
	Our technique	Christen et al. [38]	Our technique	Christen et al. [38]
First names	0.30%	0.29%	0.05%	0.05%
Cities	0%	0%	0%	0%
Zip codes	0%	0%	0%	0%
Telephone numbers	0.90%	0.95%	0.02%	0.02%

We generated q-grams using the length of q-gram $q = 3$ for first names, cities, and zip codes datasets, and $q = 4$ for telephone numbers. For each dataset, we used the minimum length of the LCS, $m = q$. We used the threshold $t = 0.6$ for generating blocks and reference values and used similarity threshold $s_t = [0.8, 0.9, 1.0]$ for classifying a string pair as a match.

In the one-to-one approach, we used decimal place $d = 2$ for converting the length of sub-string in string l_v^s (decimal number) to n_v^s (integer number). We used the one-way hash function $\mathcal{H}() = SHA256$ [154] for the $HMAC()$ function to generate hash values of 1- and 0-encodings [114], where we let the first DO generates 1-encodings and the second DO generates 0-encodings. We then used Paillier homomorphic encryption [131] to generate a ciphertext (encryption) of each length of a sub-string in a string (n_v^s). In the many-to-one approach, we used secret salt value $s_v = 45$ for permuting the lists of identifiers and l_v^s values.

For the baselines, we used the same parameter settings as our proposed approaches such as the length of q-gram q, minimum length of the LCS m, similarity threshold $s_t = [0.8, 0.9, 1.0]$, and the hash function $\mathcal{H}() = SHA256$. For other parameters, we used the same settings as we described in Chapter 6.

To compare approaches in terms of scalability, we also used a q-gram based blocking technique [38] (as we described in Section 2.1) to evaluate the performance of our block generation based on Equation 7.10. For the q-gram based blocking [38], we used the same $t = 0.6$ for calculating the number of q-grams to generate blocks.

7.9.2 Linkage Quality Results

As described in Section 7.8, the ciphertexts can contain noise (errors). In Table 7.4, we show the number of errors in our approaches that cause to lower linkage quality. For the linkage quality evaluation, we used precision and recall [33, 182], as we described in Section 2.5.1.

We first compared the normalised length of the LCS of string pairs (act_{lcs}) with the selected normalised length of the LCS, lcs, of the corresponding encrypted l_v^s of the string pair based on Equation 7.9 for our one-to-one and many-to-one approaches, and based on Equation 6.2 for the ShiftedHash and BitArray approaches to make them comparable with the Dice-coefficient of q-gram sets (Equation 2.1) and of BFs

Table 7.5: Precision and recall for different datasets and approaches, where the blocks and references are generated using Equation 7.10 as described in Section 7.4. The worst precision or recall results for each dataset are shown in red bold.

Dataset	Approach	$s_t = 0.8$		$s_t = 0.9$		$s_t = 1.0$	
		prec	*rec*	*prec*	*rec*	*prec*	*rec*
First names	One-to-One	1.0	1.0	1.0	0.93	1.0	0.93
	Many-to-One	0.99	1.0	0.99	0.95	0.99	0.95
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0
	BF	**0.49**	1.0	**0.81**	1.0	1.0	1.0
Cities	One-to-One	1.0	1.0	1.0	1.0	1.0	1.0
	Many-to-One	1.0	1.0	1.0	1.0	1.0	1.0
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0
	BF	1.0	1.0	1.0	1.0	1.0	1.0
Zip codes	One-to-One	1.0	1.0	1.0	1.0	1.0	1.0
	Many-to-One	1.0	1.0	1.0	1.0	1.0	1.0
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0
	BF	**0.67**	1.0	1.0	1.0	1.0	1.0
Telephone numbers	One-to-One	1.0	1.0	1.0	0.94	1.0	0.8
	Many-to-One	1.0	1.0	1.0	0.98	1.0	0.9
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0
	BF	**0.21**	1.0	**0.4**	1.0	1.0	1.0

(Equation 2.6) for the BF [155] approach. For a pair of l_v^s values or strings, we calculate the length of LCS, act_{lcs}, and Dice-coefficient, sim_D^q, between plaintext values of a pair, and use them as the true similarity. For a given threshold, s_t, we classified the corresponding encrypted pair with its lcs (sim_D^b for BF encoding) as:

- A pair is classified as true positive (TP) if $act_{lcs} \geq s_t$ and $lcs \geq s_t$.
- A pair is classified as false positive (FP) if $act_{lcs} < s_t$ and $lcs \geq s_t$.
- A pair is classified as true negative (TN) if $act_{lcs} < s_t$ and $lcs < s_t$.
- A pair is classified as false negative (FN) if $act_{lcs} \geq s_t$ and $lcs < s_t$.

We then used these classified values to calculate precision by using Equation 2.13 and recall by using Equation 2.14. We illustrate the precision and recall results of our approaches compared to ShiftedHash, BitArray, and BF approaches for different datasets and blocking techniques, in Tables 7.5 and 7.6. However, the results for the baselines are from a total of 1,000,000 sub-string (string) pairs because the total numbers of comparisons are very large, as we show in Table 7.8, and therefore we limit the comparison to 1,000,000 pairs.

As can be seen in Tables 7.4 to 7.6, in our approaches, the errors of ciphertexts cause to lower precision, while common sub-strings of the two DOs with no reference values cause to lower recall because they are not compared (as we discussed

Table 7.6: Precision and recall for different datasets and approaches, where the blocks and references are generated by using the q-gram based blocking technique proposed by Christen et al. [38]. The worst precision or recall results for each dataset are shown in red bold.

Dataset	Approach	$s_t = 0.8$		$s_t = 0.9$		$s_t = 1.0$	
		prec	*rec*	*prec*	*rec*	*prec*	*rec*
First names	One-to-One	1.0	1.0	1.0	0.93	1.0	0.93
	Many-to-One	0.99	1.0	0.99	0.95	0.99	0.95
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0
	BF	**0.49**	1.0	**0.81**	1.0	1.0	1.0
Cities	One-to-One	1.0	1.0	1.0	1.0	1.0	1.0
	Many-to-One	1.0	1.0	1.0	1.0	1.0	1.0
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0
	BF	1.0	1.0	1.0	1.0	1.0	1.0
Zip codes	One-to-One	1.0	1.0	1.0	1.0	1.0	1.0
	Many-to-One	1.0	1.0	1.0	1.0	1.0	1.0
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0
	BF	**0.67**	1.0	1.0	1.0	1.0	1.0
Telephone numbers	One-to-One	1.0	1.0	1.0	0.95	1.0	0.8
	Many-to-One	1.0	1.0	1.0	0.98	1.0	0.9
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0
	BF	**0.2**	1.0	**0.39**	1.0	1.0	1.0

in Section 7.8.1). Overall, the ShiftedHash and BitArray approaches provide the best precision and recall values, while the BF approach provides the worst precision values because it results in many false positives which can be caused by hash collisions in BFs [38, 155] as we described in Section 2.3. Our one-to-one and many-to-one approaches provide high linkage quality (provide precision and recall from 0.8 to 1.0). The blocks generated using different techniques (Equation 7.10 and q-gram based blocking [38]) do not seem to affect the linkage quality of any approaches.

7.9.3 Scalability Results

Table 7.7 shows the numbers of references generated from different global datasets and the numbers of blocks of different datasets. These reference values and blocks were generated based on our approach using Equation 7.10 as we described in Section 7.4 and the q-gram based blocking [38]. The number of blocks of a dataset in each pair is the same for all approaches. As can be seen in Table 7.7, the string of longer length (longer l_v) and larger q-grams (q) result in larger numbers of references and blocks.

For example, the telephone numbers dataset ($l_v = 10$) was generated by using $q = 4$ and resulted in larger numbers of references and blocks than first names,

Table 7.7: Numbers of blocks and references generated by using our Equation 7.10 as described in Section 7.4 compares to the numbers of blocks and references generated by using the original q-gram based blocking [38]. The number of blocks of a dataset in each pair is the same for different approaches (our approaches and baselines).

Dataset	Number of references		Number of blocks	
	Equation 7.10	Christen et al. [38]	Equation 7.10	Christen et al. [38]
First names	41,526	41,526	41,526	41,526
Cities	224	224	224	224
Zip codes	92	92	92	92
Telephone numbers	460,561	372,291	460,561	372,291

Table 7.8: Numbers of comparisons for different dataset pairs and approaches, where blocks are generated using different blocking techniques (our Equation 7.10 and original q-gram based blocking [38]). The numbers of comparisons for the three baselines (ShiftedHash, BitArray, and BF) are same.

Dataset	One-to-One		Many-to-One		Baselines	
	Equation 7.10	Christen et al. [38]	Equation 7.10	Christen et al. [38]	Equation 7.10	Christen et al. [38]
First names	41,526	41,526	1	1	119,919	119,919
Cities	112	112	1	1	12	12
Zip codes	78	78	1	1	80	80
Telephone numbers	460,561	372,291	1	1	58,961,256	7,101,016

cities, and zip codes which have shorter strings and were generated by using $q = 3$. This is because a larger q results in more unique sub-strings where the longer l_v results in a large number of references (and *bkv*s) to be generated for each string.

Table 7.8 shows the numbers of comparisons of different dataset pairs and approaches, where the numbers for the baselines are all the same. As can be seen, the numbers of comparisons of our one-to-one approach are the same as the number of references in Table 7.7. This is because the DOs generate and encrypt datasets based on the reference values in the list of references before sending them to the LU for conducting a comparison. For our many-to-one approach, the numbers of comparisons are 1 because all lengths of sub-strings in strings of each DO are inserted into one list before being encrypted into a single ciphertext and sent to the LU for conducting a comparison.

The numbers of comparisons of the baseline approaches for first names and telephone numbers are much larger than the numbers of their blocks because each block contains more than one encoding. The number of comparisons of telephone numbers for blocks that are generated using Equation 7.10 (number of q-grams $q_m = 3$) is larger than using the original q-gram based blocking [38] (number of q-grams $q_m = 4$) because the calculated number q_m of Equation 7.10 is smaller. This results in more records inserted into the same blocks, thus, a larger number of comparisons.

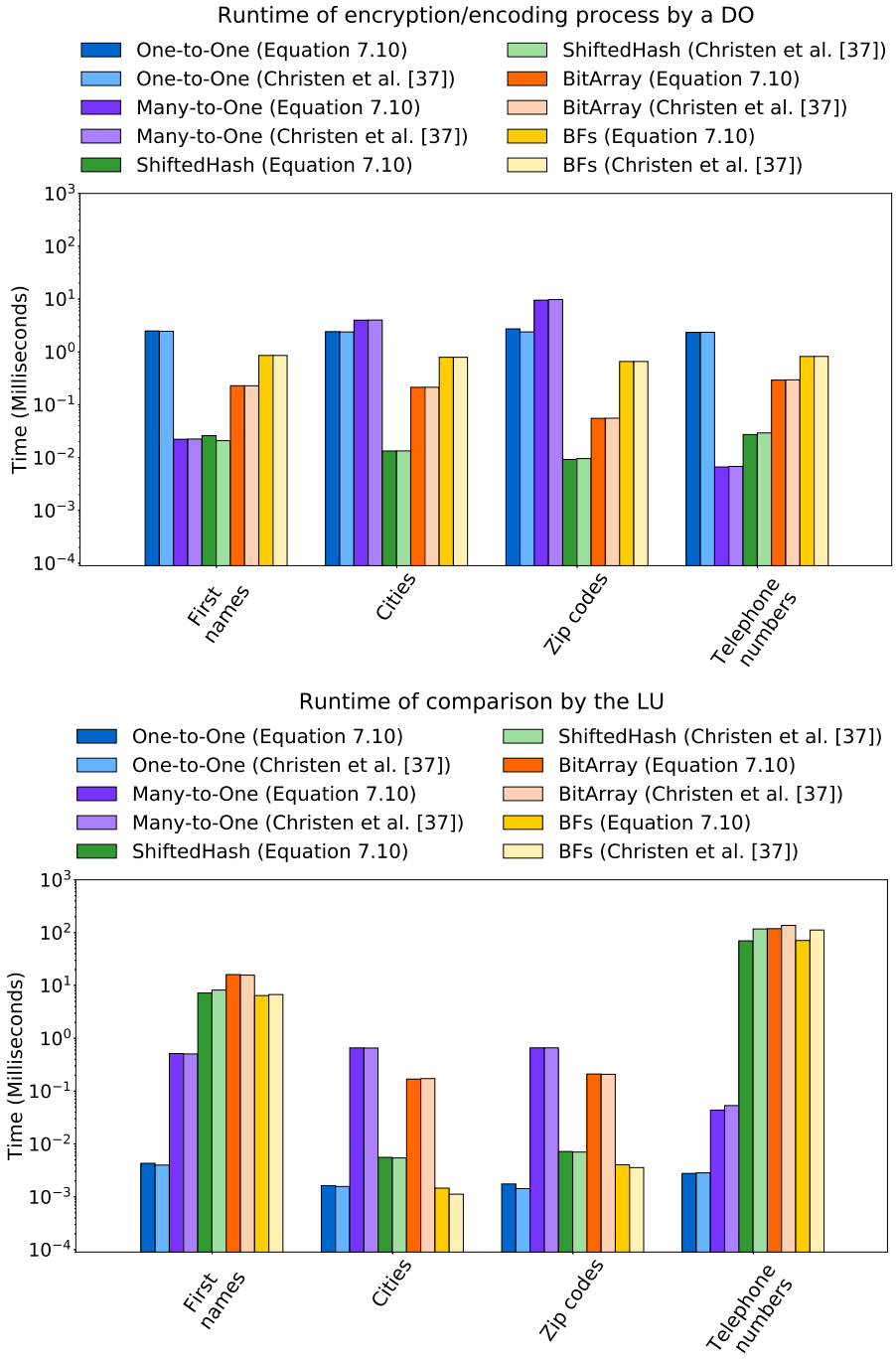

Figure 7.4: Runtimes per sub-string (string)/pair for the encryption/encoding (top) and comparison (bottom) processes by a DO and the LU, respectively. These results are compared between our approaches, ShiftedHash, BitArray, and BF approaches, where the blocks (and references) were generated using Equation 7.10 and the original q-gram based blocking [38].

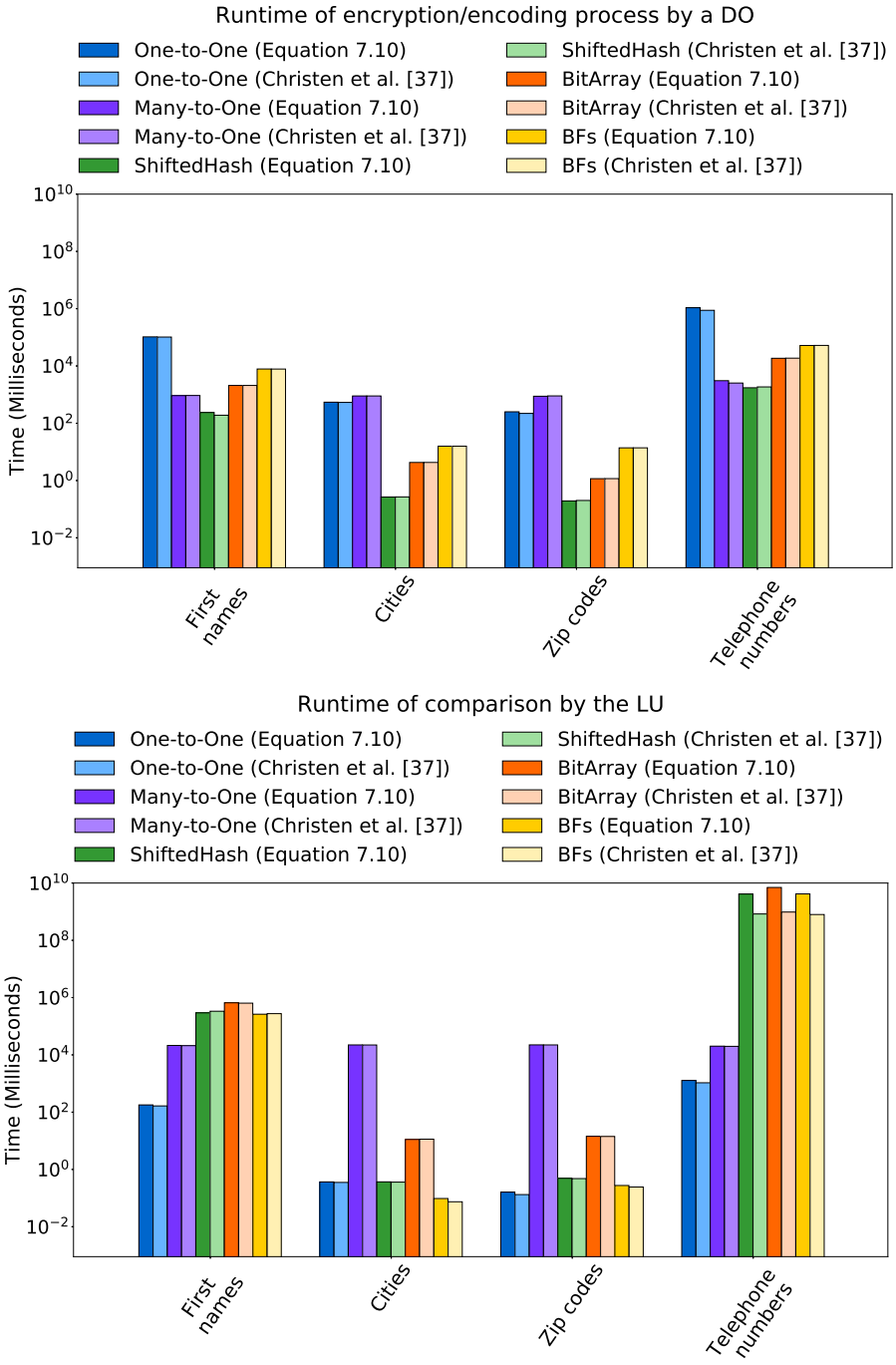

Figure 7.5: Runtimes of the whole dataset/pair for the encryption/encoding (top) and comparison (bottom) processes by a DO and the LU, respectively. These results are compared between our approaches, ShiftedHash, BitArray, and BF approaches, where the blocks (and references) were generated using Equation 7.10 and the original q-gram based blocking [38].

We evaluated the runtimes of the encryption process by a DO and the comparison process by the LU, as shown in Figures 7.4 and 7.5, where we report the times of encryption/encoding and comparison in milliseconds. Runtimes for each pair of sub-strings (or strings) are shown in Figure 7.4 and runtimes for the whole datasets are shown in Figure 7.5. For the ShiftedHash and BitArray approaches, we use the fast comparison algorithms described in Sections 6.3.3 and 6.4.2 for conducting a comparison. For BitArray approach, we set the segment length to 50% of the required length of the longest common bit which is very fast compared to the basic comparison algorithm (as we described in Section 6.4.2). However, as the number of comparisons for the baselines is very large (for telephone numbers), we limit the number of comparisons to 1,000,000 pairs. Therefore, the runtimes for (telephone numbers) dataset pair of baselines were estimated based on the runtimes per pair and the numbers of their corresponding comparisons.

As can be seen from the process by a DO in Figures 7.4 and 7.5 (top), the ShiftedHash encoding approach is the fastest encoding technique while our approaches are the slowest technique for most of the datasets. This is because the ShiftedHash approach encodes a list of q-grams into hash values and then shifts these hash values in the list. The process is very fast and requires only $O(|\mathbf{q}|)$, where $|\mathbf{q}|$ is the length of a list of q-grams. Our approaches use longer runtimes because of multiple steps of encryptions, as we described in Sections 7.5 and 7.6. The runtimes of the BitArray approach for all datasets are less than the BF encoding. Overall, the runtimes for the whole datasets, as shown in Figure 7.5, for the baselines are depended upon the number of blocks and strings to be encoded in these blocks. The runtimes for our one-to-one approach are depended upon the number of reference values, while the numbers of reference values do not affect the runtimes for our many-to-one because it encrypts a list of multiple values in a single step.

As can be seen from the comparison by the LU in Figures 7.4 and 7.5 (bottom), our one-to-one approach provides the fastest comparison. Our many-to-one approach consumes longer runtime for comparison than the one-to-one, and the three baselines when the number of comparisons is smaller because of the complicated comparison process and multiple arithmetic operations that have to be conducted over the ciphertexts (Equations 7.4 to 7.6). For a larger number of comparisons such as for first names and telephone numbers, our many-to-one provides faster comparison runtimes than the baselines although the BF encoding provides a very fast comparison and the ShiftedHash and BitArray approaches conduct the comparison process by using the fast comparison algorithms (as we described in Chapter 6).

Comparing the runtimes of the process by a DO based on blocks and references generated using our Equation 7.10 (as we described in Section 7.4) and the original q-gram based blocking [38], the runtimes seem to be not much different. However, the runtimes of the comparison process by the LU for the baselines depend upon the number of comparisons, and therefore the runtimes based on blocks generated by using our Equation 7.10 is longer than the original q-gram based blocking [38] for the telephone numbers dataset pair.

Table 7.9: Privacy measures for different datasets and approaches, where the blocks and references generated using our Equation 7.10 as described in Section 7.4. The best results are shown in blue bold and the worst results are shown in red italic texts.

Dataset	Approach	$H(\mathbf{D})$	$H(\mathbf{D}\|\mathbf{E})$	IG	RIG	DR_{max}	DR_{mean}	DR_{med}	DR_{mark}
First names	One-to-One	15.34	0.0	15.34	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
	Many-to-One	15.34	0.0	15.34	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	14.97	2.57	12.40	0.83	1.0	0.03	0.17	0.0
	BitArray	14.97	2.57	12.40	0.83	1.0	0.03	0.17	0.0
	BF	14.97	2.57	12.40	0.83	1.0	0.03	0.17	0.0
Cities	One-to-One	7.80	0.0	7.80	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
	Many-to-One	7.80	0.0	7.80	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	7.81	3.61	4.19	0.54	0.33	0.01	0.10	0.0
	BitArray	7.81	3.61	4.19	0.54	0.33	0.01	0.10	0.0
	BF	7.81	3.61	4.19	0.54	0.33	0.01	0.10	0.0
Zip codes	One-to-One	6.52	0.0	6.52	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
	Many-to-One	6.52	0.0	6.52	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	6.23	2.03	4.19	0.67	0.16	0.03	0.16	0.0
	BitArray	6.23	2.03	4.19	0.67	0.16	0.03	0.16	0.0
	BF	6.23	2.03	4.19	0.67	0.16	0.03	0.16	0.0
Telephone numbers	One-to-One	18.81	0.0	18.81	*1.0*	*1.0*	*1.0*	*0.0*	*1.0*
	Many-to-One	18.81	0.0	18.81	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	17.38	1.66	15.72	0.90	1.0	0.01	0.10	0.0
	BitArray	17.38	1.66	15.72	0.90	1.0	0.01	0.10	0.0
	BF	17.38	1.66	15.72	0.90	1.0	0.01	0.10	0.0

7.9.4 Privacy Results

We evaluated privacy by using information gain (IG) [184], relative information gain (RIG) [93] and different disclosure risks [182] (as we described in Section 2.5). We assume the worst-case scenario where an adversary uses the same parameter settings, global database, and the same database to be encoded/encrypted as a DO.

We illustrate the results of our approaches compared to ShiftedHash, BitArray, and BF approaches for different datasets and blocking techniques, in Tables 7.9 and 7.10. However, for our one-to-one approach in these tables, we show only evaluation results of the list of hash values because the evaluation on ciphertexts in this approach provides the same result as our many-to-one approach.

As can be seen in Tables 7.9 and 7.10, our one-to-one approach provides the weakest privacy protections because the relationship between the reference value and the list of hash values is a one-to-one relationship, and therefore there is no uncertain and high possibility that the adversary can infer the plaintext value in the database. The approach provides high values of IG, RIG, and DR because we assume the adversary uses the same databases (global database and database to be encrypted) and parameter settings as a DO. If the adversary uses different databases from a DO, the IG, RIG, and DR values will be lower depending upon how many strings in its guess databases and DO's databases are in common. However, the adversary can still infer some plaintext values based on those common strings.

Table 7.10: Privacy measures for different datasets and approaches, where the blocks and references generated using the original q-gram based blocking technique proposed by Christen et al. [38]. The best results are shown in blue bold and the worst results are shown in red italic texts.

Dataset	Approach	$H(\mathbf{D})$	$H(\mathbf{D}\|\mathbf{E})$	IG	RIG	DR_{max}	DR_{mean}	DR_{med}	DR_{mark}
First names	One-to-One	15.34	0.0	15.34	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
	Many-to-One	15.34	0.0	15.34	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	14.97	2.57	12.40	0.83	1.0	0.03	0.17	0.0
	BitArray	14.97	2.57	12.40	0.83	1.0	0.03	0.17	0.0
	BF	14.97	2.57	12.40	0.83	1.0	0.03	0.17	0.0
Cities	One-to-One	6.52	0.0	6.52	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
	Many-to-One	6.52	0.0	6.52	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	7.81	3.61	4.19	0.54	0.33	0.01	0.10	0.0
	BitArr	7.81	3.61	4.19	0.54	0.33	0.01	0.10	0.0
	BF	7.81	3.61	4.19	0.54	0.33	0.01	0.10	0.0
Zip codes	One-to-One	6.52	0.0	6.52	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
	Many-to-One	6.52	0.0	6.52	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	6.23	2.03	4.19	0.67	0.16	0.03	0.16	0.0
	BitArray	6.23	2.03	4.19	0.67	0.16	0.03	0.16	0.0
	BF	6.23	2.03	4.19	0.67	0.16	0.03	0.16	0.0
Telephone numbers	One-to-One	18.51	0.0	18.51	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
	Many-to-One	18.51	0.0	18.51	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	17.62	1.66	15.95	0.91	1.0	0.01	0.10	0.0
	BitArray	17.62	1.66	15.95	0.91	1.0	0.01	0.10	0.0
	BF	17.62	1.66	15.95	0.91	1.0	0.01	0.10	0.0

Our many-to-one approach provides the strongest privacy protections because multiple values are encrypted into a single ciphertext. The RIG value is 1.0 for this approach because the database $\mathbf{D}$ given $\mathbf{E}$, $H(\mathbf{D}|\mathbf{E})$ results in 0.0, thus, the RIG calculation follows Equation 2.23, where IG calculated using Equation 2.22 results in 1.0. However, as can be seen from different DR measures of this approach, it provides all 0.0 for different datasets. These $DR = 0.0$ implies that there is no risk of disclosure for the approach. In a case where the adversary can correctly guess the private key and is able to decrypt the ciphertext(s) [112], it is still high uncertainty for the adversary to re-identify the plaintext values as we already described in Section 7.8.3.

For the baselines, many encodings can correspond to each sub-string (string), where all baselines provide the same number of encodings for each sub-string (string). As we assume the adversary uses the same parameters and database as a DO, for BF encoding [155], the adversary can generate the same BFs as an encoded database of the DO. For the ShiftedHash approach, although the list of hash values has been shifted (as we described in Section 6.3.2) the adversary can rotate the list until it finds the same list of hash values as a list of a DO. Similarly for the BitArray approach, as the adversary uses the same parameters and database, it knows the q-gram bit array (as we described in Section 6.4.1). Therefore, the adversary will be able to find the bit array of a string before it has been padded with random bits.

As also be seen in Tables 7.9 and 7.10, the longer strings such as first names and telephone numbers result in larger numbers of entropy $H(\mathbf{D})$ than shorter strings

such as cities and zip codes. However, the shorter strings provide higher *DR*. This is because there are more numbers of the probability of suspicion with the value of 1.0 which means that a value in a database of a DO and a value in the database used for the privacy evaluation (which we assume they are the same) have a one-to-one relation. For different blocking techniques, the results are again not much different.

7.10 Chapter Summary

In this chapter, we have presented two privacy-preserving record linkage (PPRL) approaches, which are one-to-one (as we described in Section 7.5) and many-to-one (as we described in Section 7.6) approaches, that allow fast comparison and provide high linkage quality string matching. Our one-to-one approach encodes and encrypts each length of a sub-string in a string (in an integer form) into a list of hash values and a ciphertext, while our many-to-one approach encrypts multiple lengths of sub-strings in strings of a database into a single ciphertext.

Our experimental evaluation, as discussed in Section 7.9, has shown that both our approaches result in high linkage quality. However, the one-to-one approach performs better in terms of both linkage quality and scalability, while in the many-to-one approach no sensitive information is being revealed. We recommend using our one-to-one approach for linking large databases that require fast and high linkage results where privacy is less concerned. We recommend using our many-to-one approach for linking large databases that require fast, high linkage quality, and high degrees of privacy.

Comparing our approaches with ShiftedHash [176], BitArray [176], and BF encoding [155] approaches, our one-to-one approach provides the fastest comparison results, while our many-to-one approach provides the strongest privacy protection. The ShiftedHash and BitArray provide the best linkage quality and the BF approach provides the lowest linkage quality. In the following chapter, we present the conclusion of this thesis and future research directions.

Chapter 8

Conclusion and Future Research Directions

In this thesis, we have proposed privacy-preserving record linkage (PPRL) techniques for providing high linkage quality of matching results, while consideration of scalability and privacy of the linkage process is also of high importance. In Section 8.1, we first discuss and summarise the research problems of this thesis. We then provide a summary of our contributions in Section 8.2. In Section 8.3, we discuss directions for future research. Finally, in Section 8.4, we conclude this thesis.

8.1 Summary of Research Problems

As we described in Chapter 1, there are three challenges of privacy-preserving record linkage (PPRL) which are linkage quality, scalability, and privacy. Linkage quality is the effectiveness of a linking process where it should provide high linkage quality to allow data scientists to accurately analyse data [38]. Scalability is the efficiency of a linking process that should provide matching results quickly [96]. Privacy is the most challenging aspect of PPRL because no sensitive information should be revealed to any parties, both that do participate and those that do not participate, in a linking process [33, 184].

As we described in Chapter 2, data collected by organisations are increasing in size every day [150]. These large volumes of data (Big data) need efficient management for data processing, linking, and analysis [90] and leading to four challenges (known as the four Vs) which are: (1) volume which refers to large amounts of data that needs high performance techniques, large data storage, and high computing resources [55, 90, 198], (2) velocity which refers to the speed of data creation and analysis [90, 185, 198], (3) variety which refers to the heterogeneous nature of data, such as different data formats and inconsistent data semantics [55, 90], and (4) veracity which refers to different levels of data quality, where data might be missing and/or contain erroneous values [33, 185].

In this thesis, we mainly focus on the linkage quality aspect of the three challenges of PPRL, where scalability and privacy are also considered. However, as the

four Vs of Big data relate to the three challenges of PPRL (as we discussed in Chapters 1 and 2), we also need to consider them to develop efficient and effective PPRL techniques that can be applied to large volumes of data.

Based on the objective we described above and the existing works we discussed in Chapter 3, we proposed PPRL techniques to address four problems: (1) Low privacy of Bloom filter (BF) encoding [155] and low linkage quality of the existing BLoom and FlIP (BLIP) hardening technique [5, 157]. (2) Missing values have not seen much attention in the record linkage and PPRL contexts, and therefore existing works provide low linkage quality when linking databases that contain missing values. (3) Loss of positional information of strings to be compared which can lead to inaccurate string matching. (4) Higher computational efforts are required to improve the linkage quality and privacy of PPRL, and therefore slow down the linkage process. We provide a summary of our contributions to address these problems in the next section.

8.2 Summary of Contributions

In this section, we provide a summary of our contributions to address the research problems (as we described in Chapter 1) and summarised in the previous section.

1. **Reference values based hardening for Bloom filter based PPRL**
   In Chapter 4, we presented a PPRL technique for improving the privacy of BF encoding [155] and the linkage quality of the BLIP hardening technique [5, 157]. Our approach selects reference values from a large publicly available global database and uses these values to modify the BLIP approach such that similar record values generate similar BLIP hardened BFs. As a result, similar record values will be classified as matches.

   The experimental evaluation on a real-world database showed that our approach provides higher linkage quality and privacy than the original BLIP approaches [5, 157]. The cryptanalysis attack [40] on our approach results in mostly wrong and no guesses which imply that the attack is not able to re-identify the original record values correctly. However, with using more reference values per record, the runtime of our approach increases, and therefore it uses a longer runtime than the original BLIP approaches.

2. **Accurate PPRL for databases with missing values**
   In Chapter 5, we presented a novel PPRL technique for linking databases that contain missing values based on a lattice structure [82] and record-level BF (RBF) encoding [59]. Each attribute value to be linked is first encoded into an attribute-level BF (ABF) [155], where the attribute with missing value results in the ABF containing only zero bits. We then generate missingness patterns and form a lattice structure of a database. The lattice structure is used to find common missingness patterns between the databases to be linked.

We generate record BF by using common missingness patterns and RBF encodings [59]. Each record BF is then inserted into one or more partitions based on a grouping method, which ensures records with different missingness patterns will be appropriately compared.

We also proposed two linkage methods which are an iterative and a batch method. The iterative linkage method compares record BFs of the two database owners (DOs) partition by partition, where once each partition has been compared, matched record BFs are removed from the next partition to be compared to prevent redundant comparison. The batch linkage method compares record BFs in all partitions of the two DOs at one time. The DOs then individually select the best match for each record pair.

We conducted an extensive analysis and experimental evaluation of our approach compared to the k-nearest neighbour based PPRL approach for missing values [30] (k-NN) and record-level BF [59] (RBF) approaches on databases of different sizes and with different patterns and numbers of missing values. The results showed that our approach can achieve high linkage quality where it substantially outperforms baseline approaches while providing the strongest privacy protection against cryptanalysis attacks [39]. The iterative linkage method uses less runtime than the batch method and the baselines, while the batch linkage method consumes the longest runtime.

3. **Accurate and efficient privacy-preserving string matching**
In Chapter 6, we presented two new privacy-preserving string matching techniques that allow the accurate and efficient calculation of the length of the longest common sub-string (LCS) between strings in a pair. Our first approach is the shifted hash encoded q-grams based approach, where we encode each q-gram in a list of q-grams and shift the hash values in the list by random positions. Our second approach is the bit array based approach, where we improve the degrees of privacy of our first approach by encoding q-grams into a bit array of fixed length. We pad a record bit array by random bits to hide the original bit positions. Both of our approaches prevent frequency attacks on individual character encodings, and therefore no re-identification is possible. For record pair comparison, we proposed the basic and fast comparison methods for both our approaches, where the basic comparison is a naive way of conducting comparison and the fast comparison is a way to conduct comparison that uses less runtime.

   Our experimental evaluation has shown that both our approaches result in the same string similarities as on the original unencoded strings, while commonly used BF encoding [155], tabulation based hashing [170] (TMH), and the Damgård-Geisler-Krøigaard (DGK) approximate string matching [66] approaches will lead to potentially much higher or lower similarities between encoded strings. Our approaches provide higher privacy than the BF and TMH approaches. For the DGK approach, as it is based on homomorphic encryp-

tion, where the same values can be encrypted into different ciphertexts, and therefore the DGK is considered as a secure approach.

For the experimental results in terms of scalability, our shifted hash encoded q-grams based approach is the fastest encoding technique, while both our approaches consume similar runtimes for comparison as the DGK approach. However, our fast comparison for the bit array based approach consumes longer runtime when the segment length used for conducting comparison is shorter. This is because of the overhead of segment generation and the left and right bits comparison.

4. **PPRL for fast and high linkage quality**
   In Chapter 7, we presented two new PPRL techniques for reducing the time complexity of string matching. The first approach is a one-to-one approach that encrypts a numerical value (that represents a length of a sub-string in a string) into one ciphertext based on the special 1- and 0-encodings [114]. The second approach is a many-to-one approach that encrypts multiple numerical values into a single ciphertext by using the packing encryption method [26].

   We evaluated our approaches compare to BF encoding [155], our shifted hash encoded q-grams, and our bit array approaches. The experimental evaluation showed that our approaches provide higher linkage quality than the BFs. Our one-to-one approach is the fastest comparison. Our many-to-one approach uses similar runtimes for different processes and dataset pairs. However, both of our approaches are the slowest encryption techniques because of multiple encryption steps. In terms of privacy evaluation, our one-to-one approach provides the weakest privacy protection while our many-to-one approach provides the strongest privacy protection compared to the baselines.

8.3 Future Research Directions

In this section, we discuss future research directions for PPRL based on the open research questions that we can make from the relevant works (as we described in Chapter 3) and our contributions from Chapters 4 to 7.

1. **Suitable reference database**
   Different PPRL approaches [91, 124, 132, 183] including our reference values based hardening for BF encoding described in Chapter 4 and our PPRL approach for fast and high linkage quality described in Chapter 7 are using a public global database as a reference database to improve linkage quality of PPRL and generating blocks. The limitations of using a global database as a reference database are (1) the plaintext values in the global database must be very similar to the values in the databases to be linked, and (2) it can lower the privacy of the linkage process if the adversary correctly guesses similar global values and uses them for re-identifying the original plaintext values of

the linked databases. Therefore, selecting or generating a suitable reference database is still an open challenge for developing PPRL techniques that aim to use reference values for linking records.

2. **Hardening techniques for BF encoding based PPRL**
   Although our reference values based hardening for BF encoding (as we described in Chapter 4) can improve the degrees of privacy of BF encoding [155] and linkage quality of the BLIP hardening technique [5, 157], it is still worth to develop novel hardening techniques for BF encoding. This is because BF encoding is widely used in the PPRL context and existing hardening techniques [5, 157, 158, 159, 161] for BF encoding including our approach cannot provide high precision of matching results.

3. **PPRL techniques for databases with missing values**
   As we described in Chapters 3 and 5, record linkage and PPRL techniques for databases that contain missing values have seen limited attention. It is worth to develop alternative linkage techniques for missing values. This is because (1) our approaches cannot work when none of the missingness patterns is in common between databases and (2) most of the proposed techniques for linking databases that contain missing values use the imputation method where it is impossible to impute values in a database that contains missing completely at random (MCAR) and structurally missing data [117].

4. **PPRL techniques for real-time and accurate string matching**
   Our shifted hash encoded q-grams and bit array based approaches (as we described in Chapter 6) provide accurate string matching results, but they consume long runtimes for the comparison process. Our one-to-one and many-to-one approaches (as we described in Chapter 7) use less time in the comparison process, but they do not provide accurate string matching results for some databases. Although various PPRL techniques have been proposed for fast or real-time string matching [12, 97, 168] (as we described in Chapter 3), some of these techniques reveal sensitive information. For example, the lengths of strings are revealed when the techniques apply suffix trees (and arrays) for linking string values [22, 134, 190, 200]. Therefore, new PPRL techniques for real-time and accurate string matching are still required.

5. **PPRL techniques based on homomorphic encryption**
   Some PPRL techniques proposed to use homomorphic encryptions to encrypt sensitive values for the linkage process [66, 167, 201]. However, homomorphic encryption can result in noise/errors when arithmetic operations are conducted over ciphertexts and when the ciphertexts are decrypted [26, 27, 47, 64, 131]. Such noise can lead to lower accuracy of ciphertexts and result in lower linkage quality. Therefore, developing homomorphic encryption as well as PPRL based on homomorphic encryption techniques to minimise noise/errors in ciphertexts is required.

8.4 Conclusion

This thesis aimed to develop PPRL techniques for high linkage quality, where the scalability and privacy challenges of PPRL, and volume, velocity, variety, and veracity challenges of Big data, were under consideration. We proposed different PPRL techniques which are (1) reference values based hardening for BF to improve the degrees of privacy of BF and linkage quality of BLIP hardening technique by using reference values, (2) a PPRL technique for databases with missing values to improve linkage quality of matching results by using a lattice structure and record-level BF , (3) PPRL techniques for accurate and efficient privacy-preserving string matching by using the longest common sub-string and shifting technique to improve both linkage quality and privacy of PPRL, and (4) PPRL techniques for fast and high linkage quality where our techniques can provide fast and high linkage quality of string matching.

The experimental evaluations have shown that our PPRL techniques provide higher linkage quality and privacy compared to the baselines used. Our string matching techniques provide accurate linkage results and outperform some baselines with regard to scalability for linking larger databases. We believe that our proposed techniques are applicable for linking large databases and practical to be used for linking real-world databases. However, there are still various open research questions for developing PPRL techniques that are needed to be addressed.

References

1. Acar, A., Aksu, H., Uluagac, A. S., and Conti, M. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur) 51*, 4 (2018), 1–35. (cited on page 19)

2. Agrawal, R., and Srikant, R. Privacy-preserving data mining. In *ACM Special Interest Group on Management of Data (SIGMOD) International Conference on Management of Data* (2000), vol. 29, ACM, pp. 439–450. (cited on page 21)

3. Al-Lawati, A., Lee, D., and McDaniel, P. Blocking-aware private record linkage. In *Proceedings of the 2nd International Workshop on Information Quality in Information Systems* (2005), pp. 59–68. (cited on page 34)

4. Alaggan, M., Cunche, M., and Gambs, S. Privacy-preserving wi-fi analytics. *Proceedings on Privacy Enhancing Technologies*, 2 (2018), 4 – 26. (cited on page 30)

5. Alaggan, M., Gambs, S., and Kermarrec, A.-M. BLIP: non-interactive differentially-private similarity computation on Bloom filters. In *Symposium on Self-Stabilizing Systems* (2012), Springer, pp. 202–216. (cited on pages 4, 5, 6, 30, 60, 61, 62, 63, 64, 65, 66, 67, 71, 77, 79, 188, and 191)

6. Anindya, I. C., Kantarcioglu, M., and Malin, B. Determining the impact of missing values on blocking in record linkage. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)* (Melbourne, 2019), pp. 262–274. (cited on pages 54, 57, 79, and 97)

7. Ash, R. B. *Basic probability theory*. Courier Corporation, 2008. (cited on page 132)

8. Babenko, M., and Starikovskaya, T. Computing longest common substrings via suffix arrays. In *International Computer Science Symposium in Russia (ICSSR)* (Moskow, 2008), Springer, pp. 64–75. (cited on page 44)

9. Bellare, M., Canetti, R., and Krawczyk, H. Keying hash functions for message authentication. In *Annual International Cryptology Conference* (1996), Springer, pp. 1–15. (cited on page 42)

10. Benaloh, J. Dense probabilistic encryption. In *Proceedings of the Workshop on Selected Areas of Cryptography* (1994), pp. 120–128. (cited on page 19)

11. Benaloh, J. D. C. *Verifiable secret-ballot elections*. PhD thesis, Yale University, 1987. (cited on page 19)

12. Beneventano, D., Bergamaschi, S., Gagliardelli, L., and Simonini, G. Blast2: An efficient technique for loose schema information extraction from heterogeneous big data sources. *Journal of Data and Information Quality (JDIQ) 12*, 4 (2020), 1–22. (cited on pages 47 and 191)

13. Benford, F. The law of anomalous numbers. *Proceedings of the American Philosophical Society* (1938), 551–572. (cited on pages 117, 120, and 123)

14. Bezawada, B., Liu, A. X., Jayaraman, B., Wang, A. L., and Li, R. Privacy-preserving string matching for cloud computing. In *IEEE 35th International Conference on Distributed Computing Systems* (2015), IEEE, pp. 609–618. (cited on pages 43 and 114)

15. Bleiholder, J., and Naumann, F. Data fusion. *ACM computing surveys (CSUR) 41*, 1 (2009), 1–41. (cited on page 1)

16. Bloom, B. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426. (cited on page 59)

17. Boneh, D., Goh, E.-J., and Nissim, K. Evaluating 2-DNF formulas on ciphertexts. In *Theory of Cryptography Conference* (2005), Springer, pp. 325–341. (cited on page 19)

18. Bonomi, L., Xiong, L., Chen, R., and Fung, B. C. Frequent grams based embedding for privacy-preserving record linkage. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management (CIKM)* (2012), pp. 1597–1601. (cited on pages 50 and 114)

19. Broder, A. Z. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES (Cat. No. 97TB100171)* (1997), IEEE, pp. 21–29. (cited on pages 42 and 114)

20. Brooks, S. Markov chain Monte Carlo method and its application. *Journal of the Royal Statistical Society: Series D (the Statistician) 47*, 1 (1998), 69–100. (cited on page 56)

21. Burrows, M., and Wheeler, D. A block-sorting lossless data compression algorithm. In *Digital SRC Research Report* (1994), Citeseer. (cited on page 44)

22. Chan, S., Kao, B., Yip, C., et al. Mining emerging substrings. In *International Conference on Database Systems for Advanced Applications (DASFAA)* (Kyoto, 2003), IEEE, pp. 119–126. (cited on pages 48 and 191)

23. Chase, M., and Shen, E. Pattern matching encryption. *International Association for Cryptologic Research (IACR) Cryptology ePrint Archive* (2014), 638. (cited on pages 43, 44, 48, and 114)

24. Chen, E., Gomez, I., Saavedra, B., and Yucra, J. Cocoon: Encrypted substring search, 2015. (cited on page 44)

25. Chen, F., Wang, D., Li, R., Chen, J., Ming, Z., Liu, A. X., Duan, H., Wang, C., and Qin, J. Secure hashing-based verifiable pattern matching. *IEEE Transactions on Information Forensics and Security 13*, 11 (2018), 2677–2690. (cited on pages 44, 48, and 114)

26. Cheon, J. H., Kim, A., Kim, M., and Song, Y. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security* (2017), Springer, pp. 409–437. (cited on pages 7, 52, 153, 154, 155, 167, 168, 171, 173, 190, and 191)

27. Cheon, J. H., Kim, D., and Kim, D. Efficient homomorphic comparison methods with optimal complexity. In *International Conference on the Theory and Application of Cryptology and Information Security* (2020), Springer, pp. 221–256. (cited on pages 52, 154, 155, 168, 171, and 191)

28. Cheon, J. H., Kim, D., Kim, D., Lee, H. H., and Lee, K. Numerical method for comparison on homomorphically encrypted numbers. In *International Conference on the Theory and Application of Cryptology and Information Security* (2019), Springer, pp. 415–445. (cited on pages 50, 52, and 171)

29. Chi, L., and Zhu, X. Hashing techniques: A survey and taxonomy. *ACM Computing Surveys (Csur) 50*, 1 (2017), 1–36. (cited on page 132)

30. Chi, Y., Hong, J., Jurek, A., Liu, W., and O'Reilly, D. Privacy-preserving record linkage in the presence of missing values. *Information Systems 71* (2017), 199–210. (cited on pages 5, 34, 54, 56, 79, 80, 97, 99, 103, 104, 107, 111, and 189)

31. Christen, P. Febrl-an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *Proceedings of the 14th ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) International Conference on Knowledge Discovery and Data mining* (2008), pp. 1065–1068. (cited on page 14)

32. Christen, P. A survey of indexing techniques for scalable record linkage and deduplication. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 24*, 9 (2011), 1537–1555. (cited on pages 13, 14, and 57)

33. Christen, P. *Data matching – Concepts and techniques for record linkage, entity resolution, and duplicate detection*. Springer, 2012. (cited on pages 1, 2, 3, 4, 11, 12, 13, 14, 15, 16, 17, 18, 23, 34, 35, 36, 45, 50, 53, 57, 67, 84, 113, 114, 176, and 187)

34. Christen, P. Preparation of a real voter data set for record linkage and duplicate detection research. *Australian National University, Canberra, Australia* (2013). (cited on page 114)

35. Christen, P. Privacy aspects in Big data integration: Challenges and opportunities. In *Proceedings of the First International Workshop on Privacy and Secuirty of Big Data* (2014), ACM, pp. 1–1. (cited on pages 1 and 2)

36. Christen, P., and Goiser, K. Quality and complexity measures for data linkage and deduplication. In *Quality Measures in Data Mining*. Springer, 2007, pp. 127–151. (cited on page 2)

37. Christen, P., Hegland, M., Roberts, S., Nielsen, O. M., Churches, T., and Lim, K. Parallel computing techniques for high-performance probabilistic record linkage. *Data Mining Group, Australian National University, Epidemiology and Surveillance Branch, Project web page: http://datamining.anu.edu.au/linkage.html* (2002), 1–11. (cited on page 3)

38. Christen, P., Ranbaduge, T., and Schnell, R. *Linking Sensitive Data: Methods and Techniques for Practical Privacy-Preserving Information Sharing*. Springer International Publishing AG, 2020. (cited on pages 1, 2, 3, 5, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 24, 25, 27, 28, 29, 30, 31, 32, 36, 37, 38, 39, 57, 63, 81, 83, 94, 97, 99, 113, 117, 119, 124, 126, 130, 133, 134, 137, 141, 152, 157, 158, 164, 176, 178, 179, 180, 181, 182, 184, and 187)

39. Christen, P., Ranbaduge, T., Vatsalan, D., and Schnell, R. Precise and fast cryptanalysis for Bloom filter based privacy-preserving record linkage. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 31*, 11 (2018), 2164–2177. (cited on pages 5, 28, 29, 57, 80, 81, 94, 97, 99, 101, 109, 110, 111, and 189)

40. Christen, P., Schnell, R., Vatsalan, D., and Ranbaduge, T. Efficient cryptanalysis of Bloom filters for privacy-preserving record linkage. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)* (2017). (cited on pages 4, 5, 23, 28, 29, 30, 48, 59, 66, 72, 77, and 188)

41. Christen, P., and Vatsalan, D. Flexible and extensible generation and corruption of personal data. In *ACM International Conference on Information and Knowledge Management (CIKM)* (San Francisco, 2013), pp. 1165–1168. (cited on page 101)

42. Christen, P., Vidanage, A., Ranbaduge, T., and Schnell, R. Pattern-mining based cryptanalysis of Bloom filters for privacy-preserving record linkage. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)* (2018). (cited on pages 4, 5, 23, 28, and 59)

43. Clifton, C., Kantarcioglu, M., Vaidya, J., Lin, X., and Zhu, M. Y. Tools for privacy-preserving distributed data mining. *ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) Explorations Newsletter 4*, 2 (2002), 28–34. (cited on pages 17, 19, 20, and 21)

44. Conrad, K. Stirling's formula. *Available in http://www.math.uconn.edu/kconrad/blurbs/analysis/stirling.pdf* (2016). (cited on page 124)

45. Cox, M., and Ellsworth, D. Application-controlled demand paging for out-of-core visualization. In *Proceedings. Visualization (Cat. No. 97CB36155)* (1997), IEEE, pp. 235–244. (cited on page 33)

46. Culnane, C., Rubinstein, B. I., and Teague, V. Options for encoding names for data linking at the Australian Bureau of Statistics. *arXiv preprint arXiv:1802.07975* (2018). (cited on page 114)

47. Damgård, I., Geisler, M., and Krøigaard, M. Efficient and secure comparison for on-line auctions. In *Australasian Conference on Information Security and Privacy* (2007), Springer, pp. 416–430. (cited on pages 45, 145, and 191)

48. Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry* (2004), pp. 253–262. (cited on page 22)

49. De Cristofaro, E., and Tsudik, G. Practical private set intersection protocols with linear complexity. In *International Conference on Financial Cryptography and Data Security* (2010), Springer, pp. 143–159. (cited on page 6)

50. Dempster, A. P., Laird, N. M., and Rubin, D. B. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological) 39*, 1 (1977), 1–22. (cited on pages 27, 55, and 56)

51. Denning, J. Saving all the bits, the science of computing. *Research Institute for Advanced Computer Science at the NASA Ames Research Center* (1990), 1–4. (cited on page 33)

52. Dillinger, P. C., and Manolios, P. Fast and accurate bitstate verification for spin. In *International SPIN Workshop on Model Checking of Software* (2004), Springer, pp. 57–75. (cited on page 24)

53. Dong, C., Chen, L., and Wen, Z. When private set intersection meets Big data: an efficient and scalable protocol. In *Proceedings of ACM Special Interest Group on Security, Audit and Control (SIGSAC) Conference on Computer and Communications Security* (2013), pp. 789–800. (cited on pages 6, 98, 99, 126, and 134)

54. Dong, X. L., and Srivastava, D. Big data integration. In *IEEE 29th International Conference on Data Engineering (ICDE)* (2013), IEEE, pp. 1245–1248. (cited on pages 2, 33, 84, and 87)

55. Dong, X. L., and Srivastava, D. Big data integration. *Synthesis Lectures on Data Management 7*, 1 (2015), 1–198. (cited on pages 1, 2, 33, 34, and 187)

56. Dunn, H. L. Record linkage. *American Journal of Public Health and the Nations Health 36*, 12 (1946), 1412–1416. (cited on page 11)

57. Durham, E., Xue, Y., Kantarcioglu, M., and Malin, B. Quantifying the correctness, computational complexity, and security of privacy-preserving string comparators for record linkage. *Information Fusion 13*, 4 (2012), 245–259. (cited on page 2)

58. Durham, E. A. *A framework for accurate, efficient private record linkage*. PhD thesis, 2012. (cited on pages 22, 23, 24, 26, 27, 28, 130, 131, 133, 134, and 137)

59. Durham, E. A., Kantarcioglu, M., Xue, Y., Toth, C., Kuzu, M., and Malin, B. Composite Bloom filters for secure record linkage. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 26*, 12 (2013), 2956–2968. (cited on pages 6, 23, 24, 26, 27, 28, 80, 94, 97, 99, 101, 103, 104, 107, 111, 188, and 189)

60. Dwork, C. Differential privacy. *International Colloquium on Automata, Languages and Programming (ICALP)* (2006), 1–12. (cited on pages 48, 50, 60, and 61)

61. Dwork, C. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation* (2008), Springer, pp. 1–19. (cited on pages 21 and 22)

62. Dwork, C., McSherry, F., Nissim, K., and Smith, A. Calibrating noise to sensitivity in private data analysis. In *Theory of Cryptography Conference* (2006), Springer, pp. 265–284. (cited on pages 21, 22, 30, and 48)

63. Eisen, M. B., Spellman, P. T., Brown, P. O., and Botstein, D. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences 95*, 25 (1998), 14863–14868. (cited on pages 144, 146, 147, and 148)

64. ElGamal, T. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory 31*, 4 (1985), 469–472. (cited on pages 19, 45, and 191)

65. Erlingsson, Ú., Pihur, V., and Korolova, A. Rappor: Randomized aggregatable privacy-preserving ordinal response. In *Proceedings of the ACM Special Interest Group on Security, Audit and Control (SIGSAC) Conference on Computer and Communications Security* (2014), ACM, pp. 1054–1067. (cited on pages 30 and 61)

66. Essex, A. Secure approximate string matching for privacy-preserving record linkage. *IEEE Transactions on Information Forensics and Security 14*, 10 (2019), 2623–2632. (cited on pages 44, 45, 114, 135, 137, 138, 139, 140, 141, 142, 143, 145, 146, 147, 148, 149, 189, and 191)

67. Fellegi, I. P., and Sunter, A. B. A theory for record linkage. *Journal of the American Statistical Association 64*, 328 (1969), 1183–1210. (cited on pages 11, 15, 27, 41, 42, 54, 55, 56, 91, 97, 103, 104, and 106)

68. Ferguson, J., Hannigan, A., and Stack, A. A new computationally efficient algorithm for record linkage with field dependency and missing data imputation. *Elsevier International Journal of Medical Informatics (IJMI) 109* (2018), 70–75. (cited on pages 56, 79, and 97)

69. Ferragina, P., and Manzini, G. Opportunistic data structures with applications. In *Proceedings 41st Annual Symposium on Foundations of Computer Science* (2000), IEEE, pp. 390–398. (cited on page 46)

70. Ferrer, J. D. I. A new privacy homomorphism and applications. *Information Processing Letters 60*, 5 (1996), 277–282. (cited on page 45)

71. Franke, M., Sehili, Z., and Rahm, E. Parallel privacy-preserving record linkage using LSH-based blocking. In *3rd International Conference on Internet of Things Big Data and Security (IoTBDS)* (2018), pp. 195–203. (cited on page 34)

72. Fun, T. S., and Samsudin, A. A survey of homomorphic encryption for outsourced Big data computation. *Korean Society for Internet Information (KSII) Transactions on Internet and Information Systems (TIIS) 10*, 8 (2016), 3826–3851. (cited on page 19)

73. Gentry, C. *A fully homomorphic encryption scheme*. Stanford university, 2009. (cited on page 19)

74. Goldreich, O. Secure multi-party computation. Tech. rep., Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Israel, 2002. (cited on pages 83, 116, and 155)

75. Goldreich, O., Micali, S., and Wigderson, A. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing* (1987), ACM, pp. 218–229. (cited on page 19)

76. Goldstein, H., and Harron, K. Record linkage: a missing data problem. *Methodological Developments in Data Linkage* (2015), 109–124. (cited on pages 3, 6, 56, 79, and 97)

77. Graham, R. L., Knuth, D. E., Patashnik, O., and Liu, S. Concrete mathematics: a foundation for computer science. *Computers in Physics 3*, 5 (1989), 106–107. (cited on page 124)

78. Gravano, L., Ipeirotis, P. G., Jagadish, H. V., Koudas, N., Muthukrishnan, S., Srivastava, D., et al. Approximate string joins in a database (almost) for free. In *27th International Conference on Very Large Databases (VLDB)* (2001), vol. 1, pp. 491–500. (cited on pages 34 and 49)

79. Hahn, F., Loza, N., and Kerschbaum, F. Practical and secure substring search. In *Proceedings of the International Conference on Management of Data* (2018), ACM, pp. 163–176. (cited on pages 6, 50, 113, and 114)

80. Halachev, M., Shiri, N., and Thamildurai, A. Exact match search in sequence data using suffix trees. In *Proceedings of the 14th ACM International Conference on Information and Knowledge Management (CIKM)* (2005), pp. 123–130. (cited on pages 6 and 113)

81. Hall, R., and Fienberg, S. E. Privacy-preserving record linkage. In *International Conference on Privacy in Statistical Databases* (2010), Springer, pp. 269–283. (cited on pages 83, 116, and 155)

82. Han, J., Pei, J., and Kamber, M. *Data mining: concepts and techniques*. Elsevier, 2011. (cited on pages 6, 16, 82, and 188)

83. Hand, D., and Christen, P. A note on using the F-measure for evaluating record linkage algorithms. *Statistics and Computing 28*, 3 (2018), 539–547. (cited on pages 35, 36, and 104)

84. Hastie, T., Tibshirani, R., Sherlock, G., Eisen, M., Brown, P., and Botstein, D. Imputing missing data for gene expression arrays, 1999. (cited on page 4)

85. Hernández, M. A., and Stolfo, S. J. The merge/purge problem for large databases. *ACM Special Interest Group on Management of Data (SIGMOD) Record 24*, 2 (1995), 127–138. (cited on page 13)

86. Hernández, M. A., and Stolfo, S. J. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery 2*, 1 (1998), 9–37. (cited on page 53)

87. Herzog, T. N., Scheuren, F. J., and Winkler, W. E. *Data Quality and Record Linkage Techniques*. Springer, New York, 2007. (cited on pages 81 and 91)

88. Juels, A., and Sudan, M. A fuzzy vault scheme. *Designs, Codes and Cryptography 38*, 2 (2006), 237–257. (cited on pages 53 and 54)

89. Jurczyk, P., Lu, J. J., Xiong, L., Cragan, J. D., and Correa, A. Fine-grained record integration and linkage tool. *Birth Defects Research Part A: Clinical and Molecular Teratology 82*, 11 (2008), 822–829. (cited on page 54)

90. Kaisler, S., Armour, F., Espinosa, J. A., and Money, W. Big data: Issues and challenges moving forward. In *46th Hawaii International Conference on System Sciences (HICSS)* (2013), IEEE, pp. 995–1004. (cited on pages 1, 33, 34, and 187)

91. Karakasidis, A., and Verykios, V. S. Reference table based k-anonymous private blocking. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing* (2012), pp. 859–864. (cited on pages 53 and 190)

92. Karakasidis, A., and Verykios, V. S. A sorted neighborhood approach to multidimensional privacy-preserving blocking. In *IEEE 12th International Conference on Data Mining Workshops (ICDMW)* (2012), IEEE, pp. 937–944. (cited on page 34)

93. Karakasidis, A., Verykios, V. S., and Christen, P. Fake injection strategies for private phonetic matching. In *Data Privacy Management and Autonomous Spontaneus Security*. Springer, 2011, pp. 9–24. (cited on pages 37, 38, 134, and 183)

94. Karapiperis, D., Gkoulalas-Divanis, A., and Verykios, V. S. Distance-aware encoding of numerical values for privacy-preserving record linkage. In *IEEE 33rd International Conference on Data Engineering (ICDE)* (2017), IEEE, pp. 135–138. (cited on page 51)

95. Karapiperis, D., Gkoulalas-Divanis, A., and Verykios, V. S. Federal: A framework for distance-aware privacy-preserving record linkage. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 30*, 2 (2017), 292–304. (cited on page 23)

96. Karapiperis, D., Gkoulalas-Divanis, A., and Verykios, V. S. Fast schemes for online record linkage. *Data Mining and Knowledge Discovery 32* (2018), 1229–1250. (cited on pages 3, 5, 7, 34, 46, 151, and 187)

97. Karapiperis, D., Gkoulalas-Divanis, A., and Verykios, V. S. Efficient record linkage in data streams. In *IEEE International Conference on Big Data (Big Data)* (2020), IEEE, pp. 523–532. (cited on pages 5, 46, 47, 151, and 191)

98. Karapiperis, D., Vatsalan, D., Verykios, V. S., and Christen, P. Efficient record linkage using a compact hamming space. In *19th International Conference on Extending Database Technology (EDBT)* (2016), pp. 209–220. (cited on page 22)

99. Karapiperis, D., and Verykios, V. S. An LSH-based blocking approach with a homomorphic matching technique for privacy-preserving record linkage. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 27*, 4 (2015), 909–921. (cited on page 96)

100. Karapiperis, D., and Verykios, V. S. A fast and efficient hamming LSH-based scheme for accurate linkage. *Knowledge and Information Systems 49*, 3 (2016), 861–884. (cited on pages 131 and 137)

101. Karmel, R., Anderson, P., Gibson, D., Peut, A., Duckett, S., and Wells, Y. Empirical aspects of record linkage across multiple data sets using statistical linkage keys: the experience of the PIAC cohort study. *BMC Health Services Research 10*, 1 (2010), 1–13. (cited on page 41)

102. Kerschbaum, F. Frequency-hiding order-preserving encryption. In *Proceedings of the 22nd ACM Special Interest Group on Security, Audit and Control (SIGSAC) Conference on Computer and Communications Security* (2015), pp. 656–667. (cited on page 50)

103. Kim, M.-S., Whang, K.-Y., and Lee, J.-G. n-Gram/2L-approximation: a two-level n-gram inverted index structure for approximate string matching. *Computer Systems Science and Engineering 22*, 6 (2007), 365. (cited on page 49)

104. Kimura, M., Takasu, A., and Adachi, J. FPI: a novel indexing method using frequent patterns for approximate string searches. In *Proceedings of the Joint Extending Database Technology/International Conference on Database Theory (EDBT/ICDT) Workshops* (Genoa, 2013), pp. 397–403. (cited on page 44)

105. Kirsch, A., and Mitzenmacher, M. Less hashing, same performance: Building a better Bloom filter. In *European Symposium on Algorithms* (2006), Springer, pp. 456–467. (cited on pages 24, 25, and 26)

106. Kissner, L., and Song, D. Privacy-preserving set operations. In *Annual International Cryptology Conference* (2005), Springer, pp. 241–257. (cited on page 19)

107. Krawczyk, H., Bellare, M., and Canetti, R. HMAC: Keyed-hashing for message authentication, 1997. (cited on pages 22 and 152)

108. Kroll, M., and Steinmetzer, S. Automated cryptanalysis of Bloom filter encryptions of databases with several personal identifiers. In *International Joint Conference on Biomedical Engineering Systems and Technologies* (2015), Springer, pp. 341–356. (cited on pages 4, 5, 23, 28, 59, 81, and 94)

109. Kuzu, M., Kantarcioglu, M., Durham, E., and Malin, B. A constraint satisfaction cryptanalysis of Bloom filters in private record linkage. In *International Symposium on Privacy Enhancing Technologies Symposium* (2011), Springer, pp. 226–245. (cited on pages 4, 5, 23, 28, 57, 59, 80, 81, 94, 99, and 109)

110. Kuzu, M., Kantarcioglu, M., Inan, A., Bertino, E., Durham, E., and Malin, B. Efficient privacy-aware record integration. In *Proceedings of the 16th International Conference on Extending Database Technology (EDBT)* (2013), pp. 167–178. (cited on page 48)

111. Leskovec, J., Rajaraman, A., and Ullman, J. Mining of massive datasets cambridge university press, 2014. (cited on page 47)

112. Li, B., and Micciancio, D. On the security of homomorphic encryption on approximate numbers. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2021), Springer, pp. 648–677. (cited on pages 173 and 184)

113. Li, B.-H., Liu, Y., Zhang, A.-M., Wang, W.-H., and Wan, S. A survey on blocking technology of entity resolution. *Journal of Computer Science and Technology 35*, 4 (2020), 769–793. (cited on page 14)

114. Lin, H.-Y., and Tzeng, W.-G. An efficient solution to the millionaires' problem based on homomorphic encryption. In *International Conference on Applied Cryptography and Network Security* (2005), Springer, pp. 456–466. (cited on pages 7, 50, 51, 152, 153, 154, 163, 165, 176, and 190)

115. Lindell, Y. Secure multiparty computation for privacy-preserving data mining. In *Encyclopedia of Data Warehousing and Mining*. IGI Global, 2005, pp. 1005–1009. (cited on pages 2 and 3)

116. Lindell, Y., and Pinkas, B. Secure multiparty computation for privacy-preserving data mining. (cited on pages 17, 18, 99, 134, and 172)

117. Little, R. J. A., and Rubin, D. B. *Statistical Analysis with Missing Data*, 3 ed. Wiley, Hoboken, 2020. (cited on pages 81 and 191)

118. Manning, C., and Schutze, H. *Foundations of statistical natural language processing*. MIT press, 1999. (cited on pages 31 and 32)

119. Mashey, J. R. Big data and the next wave of InfraStress problems, solutions, opportunities. In *Computer Science Division Seminar, University of California, Berkeley* (1997). (cited on page 33)

120. McCallum, A., Nigam, K., and Ungar, L. H. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the 6th ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD) International Conference on Knowledge Discovery and Data Mining* (2000), Citeseer, pp. 169–178. (cited on page 46)

121. McCreight, E. M. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM) 23*, 2 (1976), 262–272. (cited on page 43)

122. Mitzenmacher, M., and Upfal, E. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. CUP, 2005. (cited on pages 63, 67, 83, and 137)

123. Monge, A. E. Matching algorithms within a duplicate detection system. *IEEE Data Eng. Bull. 23*, 4 (2000), 14–20. (cited on page 16)

124. Mullaymeri, X., and Karakasidis, A. A two-party private string matching fuzzy vault scheme. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing* (2021), pp. 340–343. (cited on pages 53 and 190)

125. Nakagawa, Y., Ohata, S., and Shimizu, K. Efficient privacy-preserving variable-length substring match for genome sequence. In *21st International Workshop on Algorithms in Bioinformatics (WABI)* (2021), Schloss Dagstuhl-Leibniz-Zentrum für Informatik. (cited on pages 46 and 114)

126. Naumann, F., and Herschel, M. An introduction to duplicate detection. *Synthesis Lectures on Data Management 2*, 1 (2010), 1–87. (cited on pages 16 and 35)

127. Navarro, G. A guided tour to approximate string matching. *ACM Computing Surveys (Csur) 33*, 1 (2001), 31–88. (cited on pages 14, 52, and 54)

128. Newcombe, H. B., Kennedy, J. M., Axford, S., and James, A. P. Automatic linkage of vital records. *Science 130*, 3381 (1959), 954–959. (cited on page 11)

129. Niedermeyer, F., Steinmetzer, S., Kroll, M., and Schnell, R. Cryptanalysis of basic Bloom filters used for privacy-preserving record linkage. *German Record Linkage Center, Working Paper Series, No. WP-GRLC-2014-04* (2014). (cited on pages 5, 24, 29, 57, 67, 80, 99, 103, 109, and 165)

130. Ong, T. C., Mannino, M. V., Schilling, L. M., and Kahn, M. G. Improving record linkage performance in the presence of missing linkage data. *Journal of Biomedical Informatics 52* (2014), 43–54. (cited on pages 54, 55, 79, 80, and 93)

131. Paillier, P. Public-key cryptosystems based on composite degree residuosity classes. In *International Conference on the Theory and Applications of Cryptographic Techniques* (1999), Springer, pp. 223–238. (cited on pages 7, 19, 163, 165, 171, 176, and 191)

132. Pang, C., Gu, L., Hansen, D., and Maeder, A. Privacy-preserving fuzzy matching using a public reference table. In *Intelligent Patient Management*. Springer, 2009, pp. 71–89. (cited on pages 6, 52, and 190)

133. Patil, M., Cai, X., Thankachan, S., Shah, R., Park, S., and Foltz, D. Approximate string matching by position restricted alignment. In *Joint Extending Database Technology/International Conference on Database Theory (EDBT/ICDT) Workshops* (Genoa, 2013), pp. 384–391. (cited on page 48)

134. Pei, J., Wu, W., and Yeh, M. On shortest unique substring queries. In *IEEE 29th International Conference on Data Engineering (ICDE)* (Brisbane, 2013), pp. 937–948. (cited on pages 48 and 191)

135. Pyvovar, O. Big O notation as one of landau's symbol. *BBK 81.2 T65* (2020), 123. (cited on pages 36 and 37)

136. Quantin, C., Bouzelat, H., Allaert, F., Benhamiche, A., Faivre, J., and Dusserre, L. Automatic record hash coding and linkage for epidemiological follow-up data confidentiality. *Methods of Information in Medicine 37*, 03 (1998), 271–277. (cited on page 22)

137. Rahm, E., and Do, H. H. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull. 23*, 4 (2000), 3–13. (cited on page 12)

138. Ramadan, B., Christen, P., Liang, H., and Gayler, R. W. Dynamic sorted neighborhood indexing for real-time entity resolution. *Journal of Data and Information Quality (JDIQ) 6*, 4 (2015), 15. (cited on page 5)

139. Ranbaduge, T., Christen, P., and Schnell, R. Secure and accurate two-step hash encoding for privacy-preserving record linkage. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)* (2020), Springer, pp. 139–151. (cited on pages 22, 41, and 42)

140. Ranbaduge, T., Christen, P., Vatsalan, D., et al. Tree based scalable indexing for multi-party privacy-preserving record linkage. (cited on page 41)

141. Ranbaduge, T., and Schnell, R. Securing Bloom filters for privacy-preserving record linkage. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management (CIKM)* (2020), pp. 2185–2188. (cited on pages 32 and 33)

142. Ranbaduge, T., Vatsalan, D., and Christen, P. Clustering-based scalable indexing for multi-party privacy-preserving record linkage. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)* (2015), Springer, pp. 549–561. (cited on page 34)

143. Ranbaduge, T., Vatsalan, D., and Christen, P. Secure multi-party summation protocols: Are they secure enough under collusion? *Transactions on Data Privacy 13*, 1 (2020), 25–60. (cited on page 20)

144. Ranbaduge, T., Vatsalan, D., Christen, P., and Verykios, V. Hashing-based distributed multi-party blocking for privacy-preserving record linkage. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)* (2016), Springer, pp. 415–427. (cited on page 34)

145. Randall, S., Brown, A. P., Ferrante, A. M., and Boyd, J. H. Privacy-preserving linkage using multiple match-keys. *International Journal of Population Data Science 4*, 1 (2019). (cited on pages 41 and 42)

146. Randall, S., Wichmann, H., Brown, A., Boyd, J., Eitelhuber, T., Merchant, A., and Ferrante, A. A blinded evaluation of privacy-preserving record linkage with Bloom filters. *BMC Medical Research Methodology 22*, 1 (2022), 1–7. (cited on page 113)

147. Randall, S. M., Ferrante, A. M., Boyd, J. H., Bauer, J. K., and Semmens, J. B. Privacy-preserving record linkage on large real world datasets. *Journal of Biomedical Informatics 50* (2014), 205–212. (cited on pages 41 and 113)

148. Randall, S. M., Ferrante, A. M., Boyd, J. H., Brown, A. P., and Semmens, J. B. Limited privacy protection and poor sensitivity: Is it time to move on from the statistical linkage key-581? *Health Information Management Journal 45*, 2 (2016), 71–79. (cited on page 41)

149. Randall, S. M., Ferrante, A. M., Boyd, J. H., and Semmens, J. B. The effect of data cleaning on record linkage quality. *BMC Medical Informatics and Decision Making 13*, 1 (2013), 1–10. (cited on page 12)

150. Rider, F., et al. Scholar and the future of the research library. (cited on pages 1, 33, and 187)

151. Rivest, R. L., Shamir, A., and Adleman, L. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21*, 2 (1978), 120–126. (cited on page 19)

152. Ross, S. M. *Introduction to probability models*. Academic press, 2014. (cited on page 30)

153. Scannapieco, M., Figotin, I., Bertino, E., and Elmagarmid, A. K. Privacy-preserving schema and data matching. In *Proceedings of the ACM Special Interest*

Group on Management of Data (SIGMOD) International Conference on Management of Data (2007), pp. 653–664. (cited on page 22)

154. Schneier, B., et al. Applied cryptography-protocols, algorithms, and source code in c, 1996. (cited on pages 63, 83, 117, 136, 165, and 176)

155. Schnell, R., Bachteler, T., and Reiher, J. Privacy-preserving record linkage using Bloom filters. *BMC Medical Informatics and Decision Making 9*, 1 (2009). (cited on pages 4, 5, 6, 15, 22, 23, 24, 25, 26, 29, 30, 41, 42, 43, 48, 50, 56, 59, 62, 67, 79, 80, 84, 97, 99, 113, 114, 135, 137, 138, 139, 140, 141, 142, 143, 145, 146, 147, 148, 149, 152, 156, 175, 177, 178, 184, 185, 188, 189, 190, and 191)

156. Schnell, R., Bachteler, T., and Reiher, J. A novel error-tolerant anonymous linking code. *Available at SSRN 3549247* (2011). (cited on pages 23, 24, 25, 26, and 28)

157. Schnell, R., and Borgs, C. Randomized response and balanced Bloom filters for privacy-preserving record linkage. In *IEEE 16th International Conference on Data Mining Workshops (ICDMW)* (2016), IEEE, pp. 218–224. (cited on pages 4, 5, 6, 23, 24, 29, 30, 59, 60, 61, 62, 63, 64, 65, 66, 67, 70, 77, 79, 103, 137, 188, and 191)

158. Schnell, R., and Borgs, C. XOR-folding for Bloom filter-based encryptions for privacy-preserving record linkage. *Working paper, German Record Linkage Center* (2016). (cited on pages 4, 5, 29, and 191)

159. Schnell, R., and Borgs, C. Hardening encrypted patient names against cryptographic attacks using cellular automata. In *IEEE International Conference on Data Mining Workshops (ICDMW)* (2018), IEEE, pp. 518–522. (cited on pages 31, 59, and 191)

160. Schnell, R., and Borgs, C. Protecting record linkage identifiers using a language model for patient names. *Studies in Health Technology and Informatics 253* (2018), 91–95. (cited on page 31)

161. Schnell, R., and Borgs, C. Encoding hierarchical classification codes for privacy-preserving record linkage using Bloom filters. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases* (2019), Springer, pp. 142–156. (cited on pages 23 and 191)

162. Schnell, R., and Borgs, C. Encoding diagnostic codes for privacy-preserving record linkage. *International Journal of Population Data Science 5*, 5 (2020). (cited on page 23)

163. Schnell, R., Klingwort, J., and Farrow, J. M. Locational privacy-preserving distance computations with intersecting sets of randomly labeled grid points. *International Journal of Health Geographics 20*, 1 (2021), 1–16. (cited on page 23)

164. Sehili, Z., Kolb, L., Borgs, C., Schnell, R., and Rahm, E. Privacy-preserving record linkage with PPJoin. *Datenbanksysteme für Business, Technologie und Web (BTW 2015)*. (cited on page 34)

165. Shannon, C. E. A mathematical theory of communication. *The Bell System Technical Journal 27*, 3 (1948), 379–423. (cited on page 124)

166. Sheikh, R., and Mishra, D. K. Protocols for getting maximum value for multi-party computations. In *Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation* (2010), IEEE, pp. 597–600. (cited on page 124)

167. Shimizu, K., Nuida, K., and Rätsch, G. Efficient privacy-preserving string search and an application in genomics. *Bioinformatics 32*, 11 (2016), 1652–1661. (cited on pages 43, 44, 45, 46, 114, and 191)

168. Shin, M., Kim, J., Kim, J., Seo, D., Park, C., Yu, S. J., and Park, S. CATS: a big network clustering algorithm based on triangle structures. In *Proceedings of the Symposium on Applied Computing* (2017), ACM, pp. 1590–1592. (cited on pages 46, 47, and 191)

169. Slaney, M., and Casey, M. Locality-sensitive hashing for finding nearest neighbors. *IEEE Signal Processing Magazine 25*, 2 (2008), 128. (cited on page 22)

170. Smith, D. Secure pseudonymisation for privacy-preserving probabilistic record linkage. *Journal of Information Security and Applications 34* (2017), 271–279. (cited on pages 22, 41, 42, 43, 114, 135, 137, 138, 139, 140, 141, 142, 143, 145, 146, 147, 148, 149, and 189)

171. Sudo, H., Jimbo, M., Nuida, K., and Shimizu, K. Secure wavelet matrix: Alphabet-friendly privacy-preserving string search for bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics 16*, 5 (2018), 1675–1684. (cited on page 46)

172. Tilly, C. The old new social history and the new old social history. (cited on page 33)

173. Troyanskaya, O., Cantor, M., Sherlock, G., Brown, P., Hastie, T., Tibshirani, R., Botstein, D., and Altman, R. B. Missing value estimation methods for DNA microarrays. *Bioinformatics 17*, 6 (2001), 520–525. (cited on page 4)

174. Ukkonen, E. Approximate string-matching over suffix trees. In *Annual Symposium on Combinatorial Pattern Matching* (1993), Springer, pp. 228–242. (cited on page 43)

175. Vaiwsri, S., Ranbaduge, T., and Christen, P. Reference values based hardening for Bloom filters based privacy-preserving record linkage. In *Australasian Conference on Data Mining* (2018), Springer, pp. 189–202. (cited on pages 23, 29, and 68)

176. Vaiwsri, S., Ranbaduge, T., and Christen, P. Accurate and efficient privacy-preserving string matching. *International Journal of Data Science and Analytics* (2022), 1–25. (cited on pages 114, 124, 152, 156, 175, and 185)

177. Vaiwsri, S., Ranbaduge, T., Christen, P., and Schnell, R. Accurate privacy-preserving record linkage for databases with missing values. *Information Systems 106* (2022), 101959. (cited on page 88)

178. Vatsalan, D., and Christen, P. An iterative two-party protocol for scalable privacy-preserving record linkage. In *Proceedings of the Tenth Australasian Data Mining Conference-Volume 134* (2012), Australian Computer Society, Inc., pp. 127–138. (cited on pages 17 and 46)

179. Vatsalan, D., and Christen, P. Sorted nearest neighborhood clustering for efficient private blocking. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)* (2013), Springer, pp. 341–352. (cited on page 53)

180. Vatsalan, D., and Christen, P. Scalable privacy-preserving record linkage for multiple databases. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management (CIKM)* (Shanghai, 2014), pp. 1795–1798. (cited on page 85)

181. Vatsalan, D., and Christen, P. Privacy-preserving matching of similar patients. *Journal of Biomedical Informatics 59* (2016), 285–298. (cited on pages 3, 23, 50, 51, 97, and 103)

182. Vatsalan, D., Christen, P., O'Keefe, C. M., and Verykios, V. S. An evaluation framework for privacy-preserving record linkage. *Journal of Privacy and Confidentiality 6*, 1 (2014). (cited on pages 16, 19, 21, 24, 28, 35, 36, 37, 38, 39, 48, 176, and 183)

183. Vatsalan, D., Christen, P., and Verykios, V. S. Efficient two-party private blocking based on sorted nearest neighborhood clustering. In *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management (CIKM)* (2013), ACM, pp. 1949–1958. (cited on pages 34, 53, and 190)

184. Vatsalan, D., Christen, P., and Verykios, V. S. A taxonomy of privacy-preserving record linkage techniques. *Information Systems 38*, 6 (2013), 946–969. (cited on pages 2, 3, 4, 12, 13, 14, 15, 16, 17, 21, 22, 35, 36, 37, 38, 96, 151, 183, and 187)

185. Vatsalan, D., Sehili, Z., Christen, P., and Rahm, E. Privacy-preserving record linkage for Big data: Current approaches and research challenges. In *Handbook of Big Data Technologies*. Springer, 2017, pp. 851–895. (cited on pages 1, 2, 3, 16, 21, 23, 33, 34, 62, and 187)

186. Verykios, V. S., Moustakides, G. V., and Elfeky, M. G. A bayesian decision model for cost optimal record matching. *The International Journal on Very Large Databases (VLDB) 12*, 1 (2003), 28–40. (cited on page 16)

187. Vidanage, A., Ranbaduge, T., Christen, P., and Randall, S. A privacy attack on multiple dynamic match-key based privacy-preserving record linkage. *International Journal of Population Data Science 5*, 1 (2020). (cited on page 42)

188. Vidanage, A., Ranbaduge, T., Christen, P., and Schnell, R. Efficient pattern mining based cryptanalysis for privacy-preserving record linkage. In *IEEE 35th International Conference on Data Engineering (ICDE)* (Macau, 2019), pp. 1698–1701. (cited on page 99)

189. Wandelt, S., Deng, D., Gerdjikov, S., Mishra, S., Mitankin, P., Patil, M., Siragusa, E., Tiskin, A., Wang, W., Wang, J., et al. State-of-the-art in string similarity search and join. *ACM Special Interest Group on Management of Data (SIGMOD) Record 43*, 1 (2014), 64–76. (cited on page 43)

190. Wang, J., and Li, R. A new cluster merging algorithm of suffix tree clustering. Springer US, pp. 197–203. (cited on pages 48 and 191)

191. Wang, J., Yang, X., Wang, B., and Liu, C. An adaptive approach of approximate substring matching. In *International Conference on Database Systems for Advanced Applications (DASFAA)* (2016), Springer, pp. 501–516. (cited on page 43)

192. Wang, W., Hu, Y., Chen, L., Huang, X., and Sunar, B. Exploring the feasibility of fully homomorphic encryption. *IEEE Transactions on Computers 64*, 3 (2013), 698–706. (cited on page 19)

193. Winkler, W. E. *Using the EM algorithm for weight computation in the Fellegi-Sunter model of record linkage*. US Bureau of the Census Washington, DC, 2000. (cited on pages 15 and 42)

194. Witten, I. H., Witten, I. H., Moffat, A., Bell, T. C., Bell, T. C., Fox, E., and Bell, T. C. *Managing gigabytes: compressing and indexing documents and images*. Morgan Kaufmann, 1999. (cited on page 48)

195. Wolfram, S. A new kind of science, wolfram media, inc. *Champaign, IL* (2002). (cited on page 31)

196. Wu, N., Vatsalan, D., Verma, S., and Kaafar, M. A. Fairness and cost constrained privacy-aware record linkage. *IEEE Transactions on Information Forensics and Security 17* (2022), 2644–2656. (cited on page 48)

197. Xu, H., Li, X., and Grannis, S. A simple two-step procedure using the Fellegi-Sunter model for frequency-based record linkage. *Journal of Applied Statistics* (2021), 1–16. (cited on page 11)

198. Yang, C., Huang, Q., Li, Z., Liu, K., and Hu, F. Big data and cloud computing: innovation opportunities and challenges. *International Journal of Digital Earth 10*, 1 (2017), 13–53. (cited on pages 1, 34, and 187)

199. Yao, A. C.-C. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science* (1986), IEEE, pp. 162–167. (cited on page 19)

200. Yu, M., Chai, C., and Yu, G. A tree-based indexing approach for diverse textual similarity search. *IEEE Access 9* (2020), 8866–8876. (cited on pages 49 and 191)

201. Zarezadeh, M., Mala, H., and Ladani, B. T. Efficient secure pattern matching with malicious adversaries. *IEEE Transactions on Dependable and Secure Computing* (2020). (cited on pages 44, 45, 114, and 191)

202. Zipf, G. *Human Behavior and the Principle of Least Effort*. Addison-Wesley Press, 1949. (cited on page 117)