Accurate privacy-preserving record linkage for databases with missing values

Sirintra Vaiwsri, Thilina Ranbaduge, Peter Christen

School of Computing, The Australian National University, Canberra, Australia

Rainer Schnell

University of Duisburg-Essen, Duisburg, Germany

Abstract

Privacy-preserving record linkage is the process of matching records that refer to the same entity across sensitive databases held by different organisations. This process is often challenging because no unique entity identifiers, such as social security numbers, are available in the databases to be linked. Therefore, quasi-identifying attributes such as names and addresses are required to identify records that are similar and likely refer to the same entity. Such quasi-identifiers are however often not allowed to be shared between organisations due to privacy and confidentiality concerns. Besides variations and errors in the values used for linking, quasi-identifiers can have missing values. A popular approach to link sensitive data in a privacy-preserving way is to encode quasi-identifying values into Bloom filters, bit vectors that allow approximate similarities between values to be calculated. However, with existing Bloom filter encoding approaches, missing values can lead to missed true matches because they affect the similarities calculated between Bloom filters. We propose a novel approach to consider missing values in privacy-preserving record linkage by adapting Bloom filter encoding based on the patterns of missingness identified in the databases being linked. We build a lattice of missingness patterns, and then generate partitions of Bloom filters over this lattice. In each partition the non-missing quasi-identifying attributes are assigned different weights during the Bloom filter generation process. This results in more accurate similarity calculations between Bloom filters and leads to better linkage quality. To prevent dictionary and frequency based attacks, in our approach each partition is encoded independently. We evaluate our approach on large real databases that contain different amounts and patterns of missing values, showing that our approach can substantially outperform both Bloom filter encoding that does not consider missing values, and an earlier Bloom filter based approach for linking sensitive databases that do contain missing values.

Keywords: Missing data, Privacy, Entity resolution, Data linkage, Bloom filter encoding

1. Introduction

Organisations in many domains increasingly collect large databases containing millions of records, where these records contain detailed information about people, such as customers, patients, tax payers, or travellers. Often such databases need to be shared and integrated to facilitate advanced analytics and processing [1]. Record linkage [2] is one major task that needs to be conducted when databases are to be integrated. Record linkage aims to identify and match records that refer to the same entities in different databases [1]. In many application domains, unique entity identifiers (such as social security numbers) are not available in the databases to be linked, and therefore the linkage needs to be conducted based on the available quasi-identifying attributes. These commonly contain personal identifying information, such as names, addresses, dates of birth, and so on. Data quality aspects such as typographical errors, variations, and changes of values over time, are common in most quasi-identifying attributes used for linkage. As a result, approximate comparison functions are required when records are being compared [1].

One major data quality aspect that so far has only seen limited attention in record linkage is missing values [3, 4, 5, 6, 7]. Missing values can occur for a variety of reasons and they can have different characteristics, as

Email addresses: sirintra.vaiwsri@anu.edu.au (Sirintra Vaiwsri), thilina.ranbaduge@anu.edu.au (Thilina Ranbaduge), peter.christen@anu.edu.au (Peter Christen), rainer.schnell@uni-due.de (Rainer Schnell)

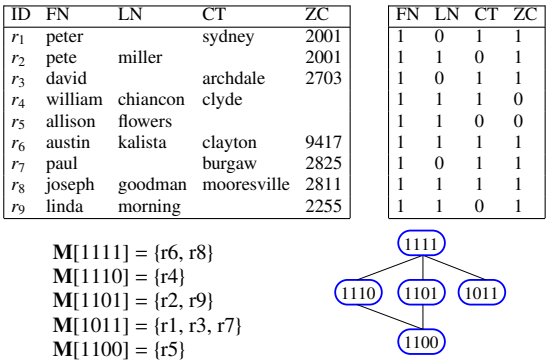

Figure 1: A small example database with nine records, where some have missing values, the resulting missingness patterns (top right, where 0 indicates a missing value) and missing pattern table, **M** (bottom left), and the resulting lattice of missingness patterns (bottom right) connecting patterns with Hamming distance 1. ID refers to the record identifier, and FN, LN, CT, and ZC to the quasi-identifying attributes first name, last name, city, and zipcode, respectively.

we discuss in Section 3. If records have missing values in the quasi-identifying 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 quasi-identifying attributes that have missing values are not used for a linkage, then linkage quality will again likely suffer [6].

Besides linkage quality, privacy is increasingly also of concern when databases that contain personal information are to be linked across organisations [8]. Certain types of linkages might not be possible if the databases contain sensitive information about individuals (such as patients or tax payers) that cannot be shared with other organisations. Privacy-preserving record linkage (PPRL) is concerned with the development of techniques that facilitate the linkage of sensitive databases across organisations without the need of any sensitive plaintext data to be exchanged or shared [9].

PPRL techniques generally encode the values in the sensitive quasi-identifying attributes used for linkage before they are being sent to the organisation that conducts the linkage [10]. The outcome of a PPRL protocol will only be the set of linked record identifiers, but no sensitive identifying information of these records. No database owner involved in a PPRL protocol should be able to learn anything about the attribute values in the records of any of the other databases being linked, and the organisation conducting the linkage or any external party (even a malicious adversary) must not be able to learn anything about the databases being linked [9].

One popular PPRL technique that is now being used in practical applications world-wide [11, 12, 13] is Bloom filter (BF) encoding [14]. As we describe in Section 3, BFs are bit vectors into which elements of a set are hashed [15] to facilitate efficient similarity calculations between values.

To the best of our knowledge, only one approach has so far investigated how to overcome the challenge of missing values in the PPRL process. The basic idea of the approach by Chi et al. [4] is to find the records that are closest (k-nearest neighbours) to a given record that has a missing value, and to use similarities calculated between the available attribute values to estimate the similarity a missing value would have had with the corresponding value in another record. This approach works best if there are groups of records that have similar attribute values, such as people in a household that have the same last name and address. While this approach has shown to be able to improve linkage quality in the presence of missing values in such situations, its drawbacks are that it is sensitive with regard to the blocking technique employed, and the use of attribute-level BFs makes it vulnerable to cryptanalysis attacks [16, 17, 18], as we show in Section 9.3.

Contributions: We propose a novel PPRL protocol based on BF encoding for linking databases that contain missing values in the quasi-identifying attributes being encoded. We first identify the patterns of occurrences of missing values across all records in a database, and use these patterns to generate a lattice structure that represents the different patterns of missingness, as shown in Fig. 1. Using a secure set intersection protocol [19] we then find the common missingness patterns in the databases to be linked, and group lattice nodes to generate partitions of BFs that need to be compared.

We then present two methods in a three-party scenario to link these partitions in a privacy preserving way: (1) an iterative method that requires multiple communication steps but that 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. To improve the privacy of our approach, we employ bit sampling from BFs based on matching weights calculated for the quasi-identifying attributes being compared [20], and different random permutations for the different partitions of BFs.

Using an extensive evaluation on large real-world databases we show that our approach can achieve high linkage quality even with significant amounts of missing values. Our approach substantially outperforms both traditional BF encoding [20] (that does not consider missing values) as well as the k-nearest neighbour missingness approach for PPRL proposed by Chi et al. [4].

2. Related work

First computer based techniques for record linkage were proposed by Newcombe et al. [21] who compared records and classified record pairs into matches and non-matches using probabilities calculated based on the likelihood that two records refer to the same person. These ideas were formalised by Fellegi and Sunter in 1969 [22], and their method to calculate match and non-match weights is still in use today [2].

Missing data has always been a challenge for linking records across databases because missing values in quasi-identifying attributes can result in lower linkage quality. Ong et al. [7] introduced three methods to improve the accuracy of linking records with missing values in the context of probabilistic record linkage [22]. These methods either redistribute the matching weights of attributes that contain missing values to non-missing attributes, estimate the similarity between attributes when values are missing in a record, or use a set of additional quasi-identifier attributes to calculate similarities if a primary quasi-identifier is missing.

Goldstein and Harron [6] developed an approach to link attributes of interest for a statistical model between an administrative (primary) database and one or more other (secondary) databases that contain the attributes of interest. The approach exploits relationships between attributes that are spread across different databases, and treats the linkage between these databases as a missing data problem. The approach uses multiple imputation to combine information from records in the different databases belonging to the same individual, and incorporates match weights to correct selection bias.

Ferguson et al. [5] proposed an approach to improve record linkage when the databases to be linked contain missing values by using a modification of the Expectation-Maximisation (EM) algorithm [23] that considers both the imputation of missing values as well as correlations between values. An evaluation of this approach on real databases containing nearly 800,000 patient records showed that it can improve linkage quality compared to when missing values are not considered.

Aninya et al. [3] analysed how different blocking techniques are affected by missing values in the attributes used for blocking [1]. Six blocking techniques were evaluated experimentally on large voter databases. Results showed that those blocking techniques that insert each record into multiple blocks, such as canopy clustering and suffix array indexing [1], performed best when the attributes used for blocking contained missing values, with suffix array indexing being the overall best performing technique.

The first PPRL approach was developed by French health researchers in the 1990s using one way hashing [24], where quasi-identifying values were hashed such that matching hashcodes mean the corresponding quasi-identifying values were the same. However, this approach only allowed for exact matching of values. Since the early 2000s different PPRL techniques have been developed, some based on secure multi-party computation (SMC) [25] while others applied perturbation to the quasi-identifying values to be compared. For details we refer the reader to the surveys by Vatsalan et al. [9] and Gkoulalas-Divanis et al. [10].

The idea behind SMC based techniques is to encrypt quasi-identifying attribute values in such a way that encrypted values can be compared and similarities between records can be calculated securely [26]. While SMC based techniques are provably secure, they generally have much higher computational and/or communication costs or they are only capable of exact matching of values [9], making them less applicable for the linking of large real-world databases.

Perturbation based techniques, on the other hand, are often more efficient and they allow approximate similarities to be calculated between encoded values [27], such as string similarities for names and addresses [1]. However, these advantages often come at a cost of some information leakage, or the vulnerability of perturbation based techniques with regard to attacks that aim to reidentify the sensitive values contained in an encoded database [16, 17, 18, 28]. The currently most widely used perturbation technique for PPRL is Bloom Filter (BF) encoding [14], which we use in our approach and formally describe in Section 3.3.

As we discussed in Section 1, only the approach by Chi et al. [4] can deal with missing values in the context of PPRL. For a record with a missing value in a quasi-identifying attribute, this approach considers the most similar k-nearest neighbouring records in the same database to calculate similarity estimates. The approach is based on attribute-level BFs [8] which have shown to be vulnerable to several cryptanalysis attacks [16, 17, 18], and it requires groups of records with high similarities, such as families and households in census records. We evaluate this approach as a baseline in Section 8 and further discuss its advantages and weaknesses as compared to our approach.

3. Background

In this section we first describe different types of missing data, and introduce the concept of a lattice structure which is a key component of our approach.

We then describe Bloom filter encoding as it has been used in the context of PPRL.

For notation, throughout this paper we use italics type letters for integers, strings, and BFs; bold lowercase letters for lists and sets; and uppercase bold letters for lattices, and for lists and sets of lists and sets. Lists are shown with square and sets with curly brackets, where lists have an order while sets do not.

3.1. 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 deletion of values due to inconsistencies, or even the refusal of individuals to provide information for example when filling in Web forms or answering surveys [8]. Missing data can be categorised into different types [29].

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. With data *missing at random* (MAR) one can assume that a missing value can be predicted by other data in the same record and/or database. As an example, for a record of a surgeon in an employment database, her salary could be predicted by averaging the salaries of all other surgeons in that database. Data *missing not at random* (MNAR) do occur for some specific reasons, for example if a patient in a medical study suffered from a stroke and therefore did not return to re-examinations then there will be missing values for this patient in the study's database. Finally, *structurally missing* data are values that are missing because they should not exist, and missing is their correct value. For example, young children should not have an occupation.

Any of these types of missing data can occur in the quasi-identifying attributes used to link databases. While data imputation can be applied with the aim to fill such missing values before the databases are linked [2, 29], in our work we assume that not all missing values can be imputed. Specifically, 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 is accessible where a complete record of that person is available).

3.2. Lattice structure to represent missingness patterns

A lattice is an abstract structure used in various application areas, including database management, data mining, data warehousing, and information retrieval, to represent multi-dimensional data [30]. As shown in Fig. 1, a lattice 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.

In our approach we build a lattice from the missingness patterns of the quasi-identifying attributes used for a linkage as identified in a database. Each such pattern is represented as a bit vector that becomes a node in the lattice, where a 0 represents a missing value while a 1 represents a non-missing value in a certain attribute. The bit vector 1111 in Fig. 1, for example, represents all records that have no missing values in four quasi-identifying attributes.

These missingness bit patterns are arranged into a lattice 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 distance is up to a threshold $d_t \geq 1$ (they differ by up to d_t bit values). The lattice generated from a database is potentially incomplete if not all possible missingness patterns occur in a database, as we discuss further in Section 5.3 and illustrate in Figs. 5 and 6. These lattices allow us to identify common missingness patterns that occur in both databases to be linked, and they are the basis of how we generate partitions of BFs.

3.3. Bloom filter encoding

The Bloom filter (BF) technique was introduced by Bloom in 1970 [15] as an efficient method for checking the element membership in a set. In 2009, Schnell et al. [14] proposed to use BFs as an encoding technique for PPRL because it allows for efficient approximate matching between the encoded quasi-identifying attribute values from different databases thereby preserving the privacy of sensitive values.

A BF, b, is a bit vector of length $l = |b|$ bits that can encode the elements of a set $\mathbf{s}$ using k hash functions that each maps $s \in \mathbf{s}$ to a bit position in the range $[1, \ldots, l]$. Initially all bits of a BF are set to zero. In PPRL, the sets to be encoded are generated from the quasi-identifying attribute values to be compared, such as the names, addresses, and dates of birth of individuals [14]. Most of these values are strings, which are converted into character substrings of length q (known as q-grams), as shown in Fig. 2. Techniques to encode numerical values [31, 32] as well as hierarchical codes [33] have also been proposed, and our approach to handle missing values in PPRL works on any of these encoding techniques.

To encode an element $s \in \mathbf{s}$ into a BF b, the element is hashed using the k hash functions h_i, with $1 \leq i \leq k$, and

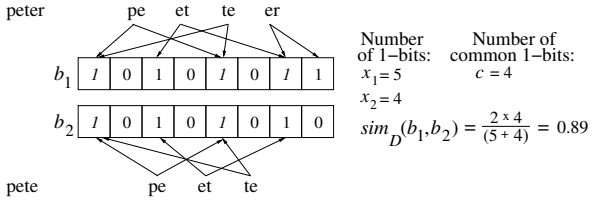

Figure 2: Example Dice coefficient similarity calculation between two BFs. The names 'peter' and 'pete' are converted into bigrams ($q = 2$) and encoded into two Bloom filters b_1 and b_2 of length $l = 8$ bits using $k = 2$ hash functions. Hash collisions occur in b_1 at bit positions 1, 5, and 7, and in b_2 at positions 1 and 5, as shown in italics.

the bit positions where the element is hashed into are set to 1: $b[h_i(s)] = 1$, with $1 \leq h_i(s) \leq l$. To calculate an approximate similarity between two BFs, a set-based similarity function such as the Dice coefficient can be used [1, 14]. The Dice coefficient similarity, sim_D, between two BFs b_1 and b_2 is calculated as:

$$sim_D(b_1, b_2) = \frac{2 \times c}{(x_1 + x_2)}, \quad (1)$$

where c is the number of 1-bits in common (at the same positions) between the two BFs, and x_1 and x_2 are the total number of 1-bits in b_1 and b_2, respectively. As also shown in Fig. 2, hash collisions can occur when different elements are hashed into the same bit position in a BF by different hash functions. While such hash collisions affect the similarity calculations between BFs, they also provide privacy because there is no one-to-one mapping of the hashed elements to bit positions [8].

While BF encoding was shown to have weaknesses that can be exploited by cryptanalysis attacks [16, 17, 18, 28], hardening techniques are being developed to make BF encoding more secure and resilient with regard to such attacks [8, 34].

In a PPRL protocol, the database owners (DOs) encode the values in their quasi-identifying attributes into BFs, and then send these BFs to a linkage unit (LU) [9], the organisation that is undertaking the linkage. Alternatively, the DOs can exchange their BFs among themselves to identify pairs of BFs with high similarities.

There are different ways of how quasi-identifying attribute values can be encoded into BFs. One approach is to use attribute-level BF (ABF) [14], where one BF is generated per quasi-identifying attribute used for the linkage (for example, one BF for first name, one for last name, and so on). One advantage of this approach is that individual similarities can be calculated per attribute which allows more detailed classification of record pairs based on different weights assigned to different attributes, as is for example used in the probabilistic record linkage approach [22]. However, one major drawback of using ABFs is that attribute values that are common in a database being encoded (such as the names of large cities or common last names) will lead to common BFs that have the same bit pattern. Frequency-based attacks can be mounted that can identify the values encoded in such commonly occurring BFs [16, 17, 18, 35].

Alternatively, values from several quasi-identifying attributes can be encoded into one BF. Two main such methods have been developed. The first is known as cryptographic long-term key (CLK) [36], where a single BF is generated per record and all attribute values are hashed into this BF. The second method is known as record-level BF (RBF) [20], where in the first step individual ABFs are generated, one per attribute used for the linkage. Different numbers of bits are sampled from these ABFs based on weights assigned to attributes, where these weights can be calculated using the probabilistic record linkage approach [22] or based on domain knowledge. The sampled bits are concatenated into one BF, which is randomly permuted to improve privacy, before being sent to a LU for comparison.

In our approach we adapt the RBF method. We also first generate ABFs and combine them into one BF per record. To deal with missing values in quasi-identifying attributes, as we describe in Section 5.4, we however employ different attribute weightings for groups of records with different missingness patterns, where we redistribute the weights assigned to the attributes with missing values to non-missing attributes [7], and then select certain numbers of bits based on the updated weights. We finally apply different permutations to records with different missingness patterns to prevent frequency [16, 17, 35] and pattern mining based attacks [28]. We illustrate our approach in Fig. 3 for two records with different missing patterns.

4. Protocol overview

Our proposed PPRL approach for missing data involves three parties, the two database owners (DOs) and a linkage unit (LU). Each DO has a database, $\mathbf{D}_A$ and $\mathbf{D}_B$, respectively, which they aim to link without having to share the sensitive quasi-identifying attribute values in their own database 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 outlined in Fig. 4, the DOs first agree on the parameters to be used in the protocol. Then they individually generate a set of attribute-level BFs (ABFs) [14]

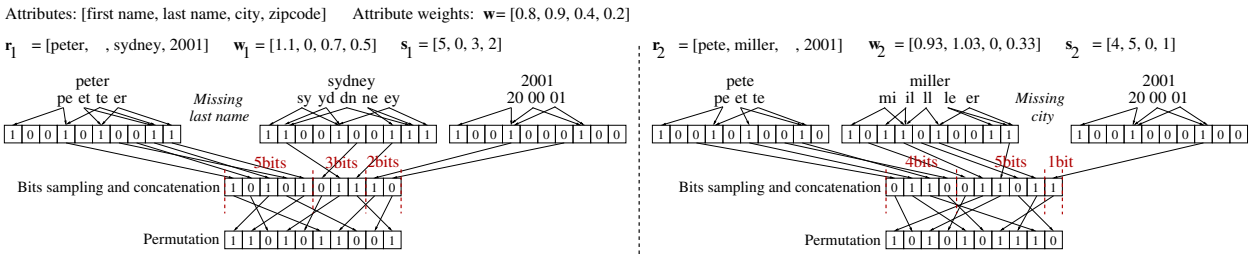

Figure 3: Record-level BF generation for records r_1 and r_2 from Fig. 1 that have different missingness patterns (r_1 has a missing last name while r_2 has city missing). First, attribute-level BFs of length $l = 10$ are generated, and based on the attribute weights in $\mathbf{w}$ and the missingness pattern, weights are redistributed from the attribute with a missing value to other attributes that do contain values, shown as $\mathbf{w}_1$ and $\mathbf{w}_2$. This results in different numbers of selected bits per attribute, $\mathbf{s}_1$ and $\mathbf{s}_2$. Finally, different permutations are applied. We describe this process in Sections 4 and 5.

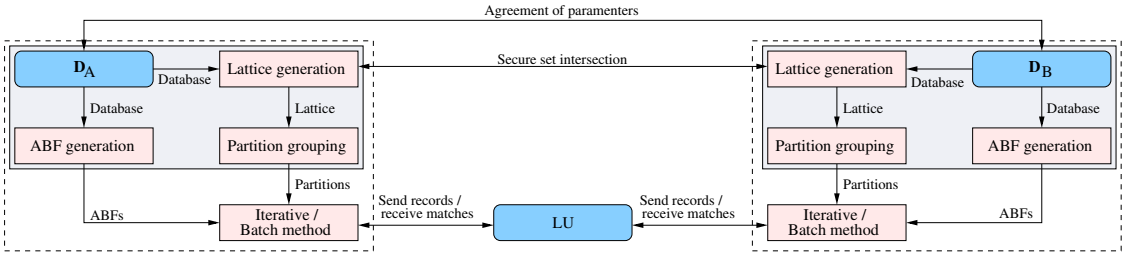

Figure 4: Overview of our proposed protocol for privacy-preserving record linkage for databases with missing values, as described in Section 4. The two databases, $\mathbf{D}_A$ and $\mathbf{D}_B$, and the linkage unit (LU) are shown in blue, while pink boxes show the main steps of our approach. The shaded areas are the common steps performed by each database owner as described in Section 5.

for each record in their database, where one ABF is generated for each of the non-missing quasi-identifying attributes in a record that is used for the linkage.

Next, the DOs individually generate local lattices, $\mathbf{L}_A$ and $\mathbf{L}_B$, where each represents the unique missingness patterns that occur in the quasi-identifying attributes in their own database. Each such pattern will represent one or multiple records in a database depending upon how missing values occur in records. We describe this process in detail in Sections 5.2 and 5.3. The DOs then participate in a secure set intersection (SSI) protocol [19] to identify the common missingness patterns between their corresponding databases, resulting in the lattice of common patterns, $\mathbf{L}_C$. A 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 reidentify each other's sensitive attribute values, as we discuss in Section 7.1. We define a common missingness pattern, m_C, as a pattern that occurs in $\mathbf{L}_C$.

Each DO now generates partitions that consist of records that have one or more missingness patterns. As we describe in Section 5.4, for each of these partitions, based on the missingness patterns of that partition, different bits will be selected (as illustrated in Fig. 3) 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. At the core of each partition is a 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 being either done upwards, downwards, or in both directions, as we discuss in Section 5.4. Based on these grouping methods the BF of each record will be added to one or multiple partitions.

The generated partitions are then compared by the LU where we propose two linkage methods, an iterative and a batch method, to identify matching record pairs as we describe in Sections 6.1 and 6.2, respectively.

In the iterative linkage method, the DOs agree upon an order of how the partitions are to be matched. Each DO sends a partition consisting of a set of BFs to the LU, which can then calculate the Dice coefficient similarity (following Eq. 1) between pairs of BFs, potentially using a blocking technique [9, 37] to speed-up the comparison process. The LU returns the classified matched record pairs (based on their BF similarities and a similarity threshold, s_t) back to the DOs, which then remove matched records 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 sent to the LU for comparison.

Algorithm 1: ***Common steps performed by the Database Owners***

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: Max. num. of missing attributes in $\mathbf{a}_L$
- w_t: Weight threshold
- l: ABF length
- d_t: Hamming distance threshold
- q: Q-gram length

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

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

One drawback of this iterative 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 [27]) where full encoded databases need to be transmitted to the LU in a single communication step. In contrast to the iterative linkage method, in the batch linkage method, which we describe in Section 6.2, all partitions are combined into one encoded database by each DO, respectively, and then sent to the LU.

5. Encoding and missing pattern processing steps

Each DO first performs the common steps outlined in Algorithm 1 and described in the following subsections.

5.1. Attribute-level Bloom filter generation

Based upon the parameters settings that have been agreed upon by the DOs, for each record r in its database **D**, in line 6 of Algorithm 1, using the function *genABF*(), a DO generates an attribute-level BF (ABF) for each attribute value *r.a* in the list of quasi-identifying attributes $\mathbf{a}_L$ to be used in the linkage process. An ABF of length l bits is generated by encoding *r.a* (converted into its set of q-grams) using the list of hash functions, **H**, that have been agreed upon by the DOs. If the attribute value *r.a* is missing an empty BF with only 0-bits is returned by *genABF*(). Each ABF is added to the list $\mathbf{b}_r$ of ABFs for record r, and this list is then inserted into the inverted index, **A** (in line 8), with the record's identifier, *r.id*. This inverted index will be used in later steps of the protocol when partitions are generated.

5.2. Missing pattern table generation

For each record $r \in \mathbf{D}$, in line 9 of Algorithm 1 we generate its missingness pattern (a bit vector), m_r, based on the list of attributes that are used for the linkage, $\mathbf{a}_L$ (for which ABFs were generated). We allow for a subset of required attributes $\mathbf{a}_R \subset \mathbf{a}_L$ to be defined. These are the attributes that cannot be missing, for example because they contain crucial information that is needed to link records, such as 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 quasi-identifying attribute in $\mathbf{a}_L$, the function *genMissPattern()* generates a 0-bit if the value is missing in record r, or a 1-bit if it exists in r. The function returns a bit vector m_r of length $|\mathbf{a}_L|$. If any of the required attributes in $\mathbf{a}_R$ is missing, then *genMissPattern()* will return a missingness pattern consisting of only 0-bits, which means a record will not be considered further in the linkage process. Similarly, 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, is too low), a record will not be considered either (lines 10 and 11 in Algorithm 1). The parameter n and the list of required attributes $\mathbf{a}_R$ can be used to control the linkage quality for those records that contain many missing values.

Each DO builds a table **M** of these missingness patterns (implemented as an inverted index), where the keys are the unique patterns, and for each pattern we have a set of all record identifiers, *r.id*, of the records with that pattern (line 11), as illustrated in Fig. 1.

5.3. Lattice generation and finding common patterns

Each DO now uses its missing pattern table, **M**, to create a local lattice structure, $\mathbf{L}_L$, that represents the patterns of missing values in its database (line 12 in Algorithm 1). Patterns are sorted in descending order based on their numbers of missing values. While the length of the missingness patterns is determined by the linkage attributes that have been encoded and are used to compare records, $\mathbf{a}_L$, the shape of the lattices being generated is determined by the parameters $\mathbf{a}_R$, and n. Assuming $|\mathbf{a}_L| - n > 1$ (more than one attribute needs to contain a value), then the lower part of lattices will be incomplete, as can be seen for example in Fig. 5. There can however be situations where a single attribute value can be useful to meaningfully link records of the same

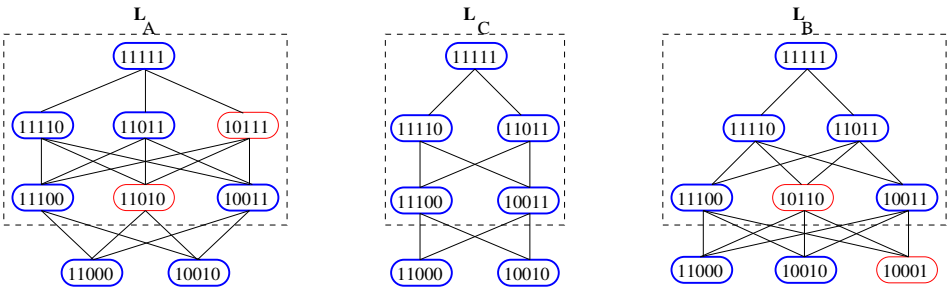

Figure 5: Local, $\mathbf{L}_A$ and $\mathbf{L}_B$, and the common $\mathbf{L}_C$ (center), lattices where at least one common missingness pattern occurs in each layer of the common lattice. We set the maximum number of attributes that can be missing to $n = 3$ and the Hamming distance threshold to $d_t = 2$.

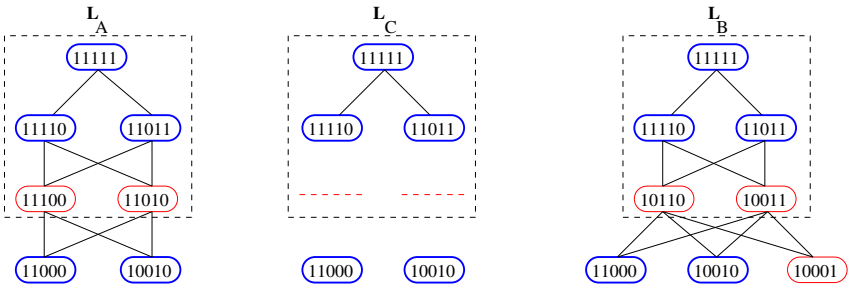

Figure 6: Local and common (center) lattices where the common lattice has missing layers.

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.

The DOs then participate in a secure set intersection (SSI) protocol [19], where the elements of the sets to be intersected are the missingness bit patterns in the local lattice, $\mathbf{L}_L$, of each DO. The output of the SSI protocol is the set of bit patterns that occur in the local lattices of both DOs, which form a new lattice of all common bit patterns, $\mathbf{L}_C$ (line 13 in Algorithm 1).

The generated common lattice, $\mathbf{L}_C$, can be categorised into one of four types, which are (1) all missing value 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 lattices, where there are common patterns in each layer of the generated lattices (at least one pattern with a certain number of 1-bits occurs in common); (3) some patterns are common across all local lattices, however there are some layers in $\mathbf{L}_C$ that have no common patterns; and (4) none of the patterns from the local lattices 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 of how records with different missingness patterns can be compared to obtain meaningful similarities between encoded records.

Figs. 5 and 6 show examples of types (2) and (3), 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 an example of the *upper* and *lower* grouping methods (as we describe next) with a Hamming distance threshold $d_t = 2$.

5.4. Grouping missingness patterns into partitions

A missingness pattern in the common lattice, $m_C \in \mathbf{L}_C$, is the basis of how partitions are formed, where records are inserted into one or more partitions based on their missingness patterns. Records in the same partition will then be compared across the databases, as we discuss in Sections 6.1 and 6.2.

In Algorithm 2 we expand line 14 from Algorithm 1, where the function *groupPattern()* is called. To allow a comparison of records that have a similar missingness patterns, for a common pattern m_C, its neighbouring patterns from the local lattice, $m_N \in \mathbf{L}_L$ are grouped into a set $\mathbf{m}_N$ based on the number of bits that differ between their missingness patterns (the Hamming distance between two 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 1.

Algorithm 2 : ***Grouping patterns into partitions***

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

Output:
- $\mathbf{P}$: Inverted index of partitions (grouped patterns)

1: $\mathbf{P}$ = {} // Initialise inverted index of partitions (grouped patterns)
2: **for** $m_C \in \mathbf{L}_C$ **do**: // Loop over common missingness patterns
3: $\mathbf{P}[m_C] = [m_C]$ // Start the partition with the common pattern itself
4: $\mathbf{m}_N = getNeighbours(m_C, \mathbf{L}_L, g, d_t)$ // Get neighbours of m_C
5: **for** $m_N \in \mathbf{m}_N$ **do**: // Loop over neighbours of the common pattern
6: $w_N = sumDiffWeight(\mathbf{w}, m_C, m_N)$ // Sum weights of bits that differ
7: **if** $w_N \leq w_t$ **then**: // Check if weight is at most the weight threshold
8: $\mathbf{P}[m_C].add(m_N)$ // Add neighbouring pattern to the current partition
9: return $\mathbf{P}$

Definition 1 (Partition). *Given $m_C \in \mathbf{L}_C$, and one or more $m_N \in \mathbf{L}_L$, and 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 local 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 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 only (g = *upper*), to common patterns in the same and lower lattice layers only (g = *lower*), or to common patterns in the same, upper, and lower lattice layers (g = *both*). We discuss these three grouping methods in more detail below.

For example in Fig. 5, in $\mathbf{L}_A$, for the common pattern 11111 and with $d_t = 2$, the following other patterns will be its neighbouring patterns with grouping *upper*: 11010, 10011, 11100, 11110, 11011, and 10111. On the other hand, in Fig. 6, in $\mathbf{L}_A$ for the common pattern 11100 and grouping *lower* (and again $d_t = 2$) the following other patterns will be its neighbouring patterns: 11111, 11110, 11011, 10111, 11010, and 10011.

We assume that the patterns in a lattice 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 Hamming distance (calculated using the function *HD()* which returns the number of bits that differ between two patterns), for each common pattern we can define its neighbourhood of other local missingness patterns as follows.

Definition 2 (Pattern neighbourhood). *For a common missingness pattern m_C and Hamming distance threshold $d_t \geq 1$, 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 bits that differ 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. When using this grouping method, 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. When this grouping method is used, 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 into 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.

In line 4 in Algorithm 2, this set of neighbouring missingness patterns, $\mathbf{m}_N$, is generated. Each neighbouring pattern $m_N \in \mathbf{m}_N$ is further assessed (in lines 6 to 8) based on the weights assigned to attributes. An attribute weight list, $\mathbf{w}$, is defined by the DOs to identify the importance of each attribute in $\mathbf{a}_L$, where

Algorithm 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 record pairs
2: **c** = {} // Initialise inverted index of matched record IDs
3: **P** = *sortPartByNumMiss*(**P**) // Sort partitions
4: **for** $(m_C, \mathbf{m}_p) \in \mathbf{P}$ **do**: // Loop over partitions
5: $\mathbf{B}_p$ = {} // Initialise the inverted index of BFs for this partition
6: **for** $m \in \mathbf{m}_p$ **do**: // Loop over missing patterns in partition
7: $\mathbf{r}_m$ = **M**[m] // Get record identifiers with this pattern
8: $\mathbf{r}'_m$ = *remCompRec*($\mathbf{r}_m$, **c**) // Remove records already matched
9: **for** $r.id \in \mathbf{r}'_m$ **do**: // All records with this pattern
10: $\mathbf{b}_r$ = **A**[$r.id$] // List of ABFs for this record
11: b_r = *genBF*($\mathbf{b}_r, m_C$, **w**) // Generate the BF for this record
12: b'_r = *permBF*(b_r, m_C) // Partition specific BF permutation
13: $r.eid$ = *encrRecID*($r.id, m_C$) // Encrypt record ID
14: $\mathbf{B}_p[r.eid] = b'_r$ // Add to index of BFs of this partition
15: *sendToLU*($\mathbf{B}_p, s_t$) // Send the BFs to the LU for matching
16: $\mathbf{R}_p$ = *receiveFromLU*() // Receive matched record pairs
17: **for** $(r_1.eid, r_2.eid) \in \mathbf{R}_p$ **do**: // Process matched record pairs
18: $s = \mathbf{R}_p[(r_1.eid, r_2.eid)]$ // 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**

a higher weight means the attribute is more relevant for the linkage process. For example, the DOs might assign a higher weight to a first name attribute compared to a gender attribute because first name is generally more important to identify records that refer to the same individual than gender. The weights in **w** can either be set based on domain expertise or using the match weight calculations employed in probabilistic record linkage [2, 22].

The weight threshold, w_t, ensures that only those records that have common linkage attributes (based on their weights) are included into the same partition. Therefore, two patterns, m_C and m_N, that have common non-missing attributes with low weights, for example only gender and age, will not be grouped into a partition. The calculation of the weight w_N in the function *sumDiffWeight()* (in line 6 of Algorithm 2) 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: $m_d = m_C \oplus m_N$. Any 1-bit in the bit vector m_d corresponds to an attribute in $\mathbf{a}_L$ that is either missing in m_C or m_N, but not both. We then calculate w_N by summing all attribute weights in **w** where the corresponding bit in m_d is set to 1:

$$w_N = \sum_{i=1}^{|\mathbf{a}_L|} \begin{cases} \mathbf{w}[i] & \text{if } m_d[i] = 1, \\ 0 & \text{otherwise.} \end{cases} \quad (2)$$

If the summed weight w_N is below the weight threshold w_t for a neighbouring pattern m_N, then m_N will be added to the partition (in line 8 in Algorithm 2). 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 more similar in order to be grouped into the same partition.

For example, let us assume $w_t = 0.5$ and the five quasi-identifying attributes first name, last name, street address, city, and zipcode 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 Fig. 6, assuming we have a grouping method g = *upper* and a Hamming distance threshold of $d_t = 1$, then the two (not common) neighbouring patterns 11100 and 11010 can possibly be grouped with the common pattern 11110. We can calculate the weights of the corresponding different bits as:

- $11110 \oplus 11100 = 00010 \rightarrow w_N = 0.8 \times 0 + 0.9 \times 0 + 0.6 \times 0 + 0.4 \times 1 + 0.2 \times 0 = 0.4$
- $11110 \oplus 11010 = 00100 \rightarrow w_N = 0.8 \times 0 + 0.9 \times 0 + 0.6 \times 1 + 0.4 \times 0 + 0.2 \times 0 = 0.6$

As a result, the weight $w_N = 0.4$ for $m_N = 11100$ (with missing city and zipcode) is the only one below the threshold $w_t = 0.5$, and therefore only this pattern is added to the partition formed by the common missingness pattern 11110 (with missing zipcode).

Algorithm 2 is performed by each DO individually. Its output is a set of partitions, **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). In the following two sections we describe how these partitions, together with the encoded BFs of all records, as generated in Algorithm 1, can be used to link the encoded databases in either an iterative fashion or using a batch approach.

6. Linkage steps

Based on the partitions generated, 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.

6.1. Iterative linkage

The aim of the iterative linkage method is to ensure that each record pair is compared only once in the matching process, even if a certain record occurs in several partitions. To achieve this goal, each DO performs the iterative steps outlined in Algorithm 3.

This iterative linkage method loops over the partitions, **P**, generated in Algorithm 2, where these partitions are sorted (in line 3 of Algorithm 3) with increasing number of missing values in their keys m_C (their common missingness pattern). This is because partitions with less missing values allow for more meaningful comparison of records given they contain more non-missing quasi-identifying attribute values.

We then loop over these sorted partitions, where in line 4 in Algorithm 3 we retrieve the key of a partition, its common missing pattern m_C, and the list of all patterns that were grouped into this partition, $\mathbf{m}_p$. We initialise a list $\mathbf{B}_p$ which will hold all record BFs that will be generated for this partition (line 5), and then loop over all missing patterns, m, in the partition. For each such pattern, we retrieve the identifiers of all records with that pattern (as the list $\mathbf{r}_m$) from the missing pattern table **M**. To prevent a previously matched record to be compared again, in line 8 we remove any record identifier from $\mathbf{r}_m$ that is in the set of matched records, **c**. In line 9 we then loop over the set $\mathbf{r}'_m$ of not yet matched records with a given missingness pattern m, retrieve their corresponding list of ABFs, $\mathbf{b}_r$, and generate one BF b_r per record using the function *genBF()* in line 11, which as input also takes the missingness pattern m_C of this partition, and the list of attribute weights, **w**.

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 [7]. This weight distribution is calculated as:

$$\mathbf{w}'[a] = \mathbf{w}[a] + \frac{\sum \mathbf{w}[a_m]}{HW(m)}, \tag{3}$$

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 (FN), last name (LN), and city (CT) are non-missing, while street address (SA) and zipcode (ZC) are missing. The attributes were originally assigned the weights 0.8 (FN), 0.9 (LN), 0.6 (ST), 0.4 (CT), and 0.2 (ZC), respectively. The weights 0.6 and 0.2 of street address and zipcode are distributed to the other three attributes as:

$$\begin{aligned} \mathbf{w}'[\text{FN}] &= 0.8 + (0.6 + 0.2)/3 = 1.067, \\ \mathbf{w}'[\text{LN}] &= 0.9 + (0.6 + 0.2)/3 = 1.167, \\ \mathbf{w}'[\text{CT}] &= 0.4 + (0.6 + 0.2)/3 = 0.667. \end{aligned}$$

After distributing the weights of missing value attributes, we sample bits from each corresponding ABF to generate the final record BF [20]. For each non-missing attribute a we calculate its number of bits, $\mathbf{s}[a]$, to be sampled as (rounded to the nearest integer):

$$\mathbf{s}[a] = \left\lfloor \frac{\mathbf{w}'[a]}{\sum_a \mathbf{w}'[a]} \times l \right\rceil, \tag{4}$$

where $\mathbf{w}'[a]$ is the weight for attribute a adjusted using Eq. 3, 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 368 for first name, 402 for last name, 230 for city, 0 for street address and zipcode (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 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 of a pseudo random number generator [8]) upon which the permutation is based. As a result, it will be very difficult for the LU to successfully apply a frequency or pattern mining based cryptanalysis attack [16, 17, 28, 35] across the BFs from different partitions.

Furthermore, to prevent that the LU can identify the BFs that correspond to the same record across partitions (using the record identifiers, *r.id*), in line 13 we generate a partition specific encrypted record identifier, *r.eid*. Only these encrypted record identifiers are sent to the LU together with their partition specific BFs, as well as the similarity threshold s_t used to decide if compared BFs are matches or not (line 15 in Algorithm 3).

Once the LU has compared and linked the BFs in a partition, as we describe in Section 6.3, it returns a set of matched record pairs (with their similarities) from that partition, $\mathbf{R}_p$, in line 16. The DOs then loop over these matches, get the similarity s of a matched record pair, and convert the encrypted record identifiers back to their original values (line 19). They then add 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. This means a matched record will not be included in any following partitions.

6.2. Batch linkage

The aim of the batch linkage method, as detailed in Algorithm 4, is to limit the communication between the DOs and the LU to one single exchange of messages. This in contrast to iterative linkage which requires one communication step per partition.

In line 3 of Algorithm 4 we loop over the partitions, **P**, generated in Algorithm 2 and retrieve the common missing pattern, m_C, and the list of all patterns that were

Algorithm 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 this pattern
6:      for r.id ∈ r_m do:                   // All records with this pattern
7:        b_r = A[r.id]                      // List of ABFs for this record
8:        b_r = genBF(b_r, m_C, w)           // Generate the BF for this record
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 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)                // Extract best matching record pairs
16: return R
```

Algorithm 5: ***Blocking and linking RBFs***

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

Output:
- **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)                // Dice similarity, Eq. 1, 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
```

grouped into this partition, $\mathbf{m}_p$. The pattern m_C is used as the key of this partition. We then loop over all patterns, m, in the partition (line 4). For each such pattern, we retrieve the identifiers of all records with that pattern (as the list $\mathbf{r}_m$) from the missing pattern table **M** in line 5. To get all records with a given missing pattern m, we loop over the list $\mathbf{r}_m$ (line 6), and for each record we retrieve its 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**. The function *genBF()* applies the weight distribution discussed for the iterative method in Section 6.1. The generated BF is then permuted in line 9, using the common pattern m_C as the seed of the permutation function, and a partition specific encrypted record identifier, *r.eid*, is generated as we described in Section 6.1. The mapping of 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**, then the DO sends **B** with the similarity threshold, s_t, to the LU to conduct matching (line 13). The LU compares and matches the BFs received from the DOs as we describe in Section 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 DOs receive from the LU, in line 15 of Algorithm 4, they use 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 3 for the iterative linkage method. This function uses the mapping table **E** to obtain the original record identifiers and partition keys 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, **R**, which is returned in line 16 of Algorithm 4.

6.3. Record pair comparison by the linkage unit

In Algorithm 5 we outline the steps the LU performs for linking a pair of such lists of BFs. First, the LU applies a blocking technique on the BFs received from both DOs (lines 2 and 3), where we assume this blocking technique is a black box that groups records into blocks [9, 37]. In line 4 the LU then identifies the common blocks that occur in the lists of BFs from both databases, and in line 5 it loops over these blocks, **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 **g** and iterates over all possible pairs of records in the block. The LU compares the BFs for a record pair using the Dice coefficient similarity function, $sim_D()$, given in Eq. 1, and if the resulting similarity s is at least the threshold s_t then a pair is classified as a match and added to the set of matches **R** with its similarity s. Finally, the LU sends the set of classified matches back to the DOs in line 11.

7. Analysis

We now analyse our approach with regard to privacy, linkage quality, and scalability. 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. [20] and Christen et al. [16, 8].

7.1. Privacy

We now analyse what the parties involved in our approach can learn from the data they receive from each other. In line with many other PPRL protocols [9, 37, 38], we assume all parties follow the honest-but-curious (HBC) adversary model [25], where however the DOs are not colluding with the LU [4].

The DOs first agree on the values of the parameters to be used in the linkage process. While $\mathbf{a}_L$ (list of linkage attributes) and $\mathbf{w}$ (list of attribute weights) allow each DO to learn about the common attributes in the databases being linked and their importance, no sensitive information about individual records is being revealed in this step. None of the parameters needed to generate the BFs, nor the method of how the permutation of BFs is 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 individually generate their own table of missingness patterns and their local lattice, which is used to identify the common patterns by employing a secure set intersection protocol [19]. From this protocol, each DO learns which missingness patterns occur in the database of the other DO, as well as the patterns that cannot occur in the other database (the patterns in the local lattice of a DO that do not occur in the common lattice). A DO does however not learn how many records the other database contains for each pattern, nor any sensitive information about individual records in the other database. Knowing the patterns that are local to the other database might leak some information that a DO can try to exploit, as we discuss below.

The encoding of sensitive quasi-identifying attribute values, first into ABFs [14] and then into RBFs [20], is conducted individually by each DO. It has been shown that RBFs are more secure than ABFs with regard to several cryptanalysis attacks [16, 17, 18, 28]. In our approach we employ an adapted RBF method where, like the original RBF approach [20], we sample bits and permute the final BFs. However, while in the original RBF method the full databases are encoded in the same way (same bit positions sampled and same permutation of bit positions applied to all BFs), in our approach we apply different sampling (based on weight redistribution) and different bit position permutations to each partition.

For an adversary who aims to reidentify the sensitive values encoded in the partitions of BFs sent from the DOs to the LU, our approach of partitioning a database reduces the amount of frequency information available in a full encoded database because of the partitioning process. Given most known cryptanalysis attacks on BF encoding for PPRL exploit frequent bit patterns in a set of BFs [16, 17, 18, 28], our approach will therefore be more secure compared to the original RBF approach because bit patterns in BFs cannot be analysed across partitions. Furthermore, in the batch approach, where all partitions are concatenated into one single set of BFs before being sent to the LU, the LU cannot identify which subset of BFs corresponds to a partition.

During the iterative linkage approach, and as the final step of the batch linkage approach, 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) back to the DOs. Because the DOs know how the record 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 quasi-identifying attributes are the same (depending upon the missingness pattern of records in a partition). However, as we discuss below, 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 how this similarity was obtained. If *upper* grouping has been applied (as we discussed in Section 5.4), then a similarity $s < 1$ can occur because one of the two records has a missing value in an attribute where the record from the other database did not have a missing value. For *lower* grouping, a similarity $s < 1$ means that not all attribute values in the common pattern (the partition key) were the same, because the values in some not missing attributes from a record in a neighbouring missingness pattern were not compared. For grouping *both* either of these situations can occur.

For example, assume the common missingness pattern m_C = 11111 and the neighbouring pattern m_N = 10111 have been grouped into a partition (using *upper*), and the following three records, one (r_1) from database $\mathbf{D}_A$, and two (r_2 and r_3) from database $\mathbf{D}_B$:

r_1	peter	smith	sydney	2000	nsw
r_2	peter		sydney	2000	nsw
r_3	pedro	smythe	sydney	2010	nsw

The similarities for both record pairs (r_1,r_2) and (r_1,r_3) will be below 1. For both pairs, the DO of database $\mathbf{D}_A$ cannot learn which of the above situations occurred for each of the two pairs because it does not know the missingness pattern of records from the other database. Therefore, once grouping of patterns is applied to

form partitions, the uncertainty of what patterns records have is making it more difficult for a curious DO to try to identify the values in the record of the other DO in any matching record pair that has a similarity $s < 1$.

One exception to this improved privacy is with grouping *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 $s = 1$. In the above example, if last name is not compared, then the record pair (r_1,r_2) 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 $s = 1$ on a subset of attribute values, however it is not able to learn this subset nor the actual compared values.

For grouping *upper*, the inclusion of BFs based on records that have missing values, as in the above example, will mean certain BFs will have less 1-bits compared to others. However, given the generally wide range in the lengths of name and address values (with the exception of zipcodes), a curious LU will not be able to distinguish those BFs that correspond to records with a missing value in an attribute from those that do have shorter values across their attributes. Furthermore, even if the LU could identify which records do have a missing value, the use of BFs and sampling of bit positions does mean no information about the not missing values in that record is being revealed [20].

To summarise the privacy characteristics of the three proposed grouping methods, grouping *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. Grouping *upper* 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 grouping *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.

7.2. Linkage quality

As with any record linkage method, a major aspect that determines linkage quality is the choice of suitable quasi-identifying attributes that can be used for a linkage [1]. The patterns of how missing values occur in the attributes used for linkage will be the biggest factor influencing linkage quality [3, 4, 5, 6]. In our approach, based on the set $\mathbf{a}_L$ of attributes used for linkage (given as input to Algorithm 1), the maximum number of missing values, n, in this set, and the set of attributes that cannot be missing, $\mathbf{a}_R$, a user can customise a linkage based on their knowledge of both data quality and missingness patterns in the databases to be linked. A thorough data exploration step to assess data quality prior to conducting a linkage is highly recommended [1].

For PPRL based on BF encoding (assuming textual attributes are encoded into BFs), the major parameters that determine linkage quality are l, q, and k [14, 31]. If RBF encoding as proposed by Durham et al. [20] 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 [22], will also influence linkage quality.

Specific to our approach of handling missing values in the attributes used for linkage, the grouping method, g, and the corresponding Hamming distance and weight thresholds, d_t and w_t, will influence how partitions of BFs are generated, which in turn determines what record pairs with certain missingness patterns will be compared. Larger values for d_t and w_t mean more missingness patterns are grouped into a partition and therefore records are potentially inserted into more partitions. This leads to more record pairs being compared and therefore an increased chance for true matching record pairs to be compared (higher recall), however at the cost of potentially also more false matching pairs to be classified as matches (lower precision).

The choice of grouping method, will influence linkage quality because it determines how records with different missingness patterns are grouped into partitions and then compared by the LU.

For grouping *upper*, neighbouring missingness patterns, m_N, that are grouped with a common pattern, m_C, in an upper lattice layer can lead to a decrease of the number of true matching record pairs (lower recall) but also to less false matching pairs (higher precision). This is because the larger number of missing attribute values in records with pattern m_N means their corresponding encoded BFs contain more 0-bits compared to records with pattern m_C. For example, if pattern $m_N = 10111$ is grouped into a partition with pattern $m_C = 11111$, then all bits in the BFs that encode records with the pattern 10111 will have only 0-bits for the second attribute. This will lead to lower Dice coefficient similarities and therefore potentially missed true matches and also less false matches.

For grouping *lower*, neighbouring missing patterns, m_N, that are grouped to a common pattern, m_C, in a

lower lattice layer (where patterns contain more missing values) can lead to an increase of the number of true matching record pairs (higher recall) but also more false matching pairs (lower precision). Because only the attribute values that are not missing in the pattern m_C are included in the BFs generated for this partition, there are non-missing attributes in the records with pattern m_N that are not included in these BFs. Less detailed information from a smaller number of attributes is encoded into BFs, potentially leading to several records having the same or similar values in these attributes. For example, assume two records (representing two individuals) that have the same first and last names and the same address, and where only their date of birth attribute has a different value (such highly similar records are not uncommon in databases that cover large student residencies). If date of birth is the attribute with missing values in m_C, then these two records will become an exact match with grouping *lower*, because date of birth is not encoded in the corresponding BFs of these records.

Finally, grouping *both* generates partitions that include neighbouring patterns m_N that are above, below, and in the same level as a pattern m_C in the lattice structure. The generated partitions will be larger than with the two previous grouping methods, and each record will potentially be grouped into a larger number of partitions. As a result, the linked data set 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 described in Section 6.1 is employed, the DOs can order the partitions being generated and sent to the LU for linkage, making sure 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 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 only based on their bit patterns because it does not know in which partition a BF occurs. This can potentially lead to wrongly matched record pairs. The set of all record pairs classified as matches by the LU, $\mathbf{R}'$, is returned to the DOs (in line 14 in Algorithm 4). Because the DOs know the partitions in which each BF occurs (based on the mapping table $\mathbf{E}$ used in Algorithm 4), they can (as with the iterative approach), group record pairs back into partitions and link those pairs with the smallest number of missing values and highest similarities first.

7.3. Scalability

In term of scalability, we analyse the complexity 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 missingness pattern table. The complexity of this step is $O(|\mathbf{D}| \cdot |\mathbf{a}_L|)$, where $|\mathbf{D}|$ is the number of records in the database being encoded and $\mathbf{a}_L$ is the list of linkage attributes. Once the missing pattern table, $\mathbf{M}$, is generated, the DOs build their local lattice. The maximum number of missingness patterns (lattice nodes) is $p = \sum_{i=0}^{|\mathbf{a}_L|-n} \binom{|\mathbf{a}_L|}{i}$, based on the parameter n which provides the maximum number of allowed missing attribute values.

Using a secure intersection protocol [19] the local lattices 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 [19]. Once the common lattice is obtained, each DO generates the partitions, $\mathbf{P}$, where the number of partitions is limited by $O(p)$ because each partition has a missingness pattern from the common lattice as its key.

The iterative linkage method, shown in Algorithm 3, loops over the partitions, and in each partition 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 Hamming distance parameter, d_t. A Hamming ball is formed around a common pattern m_C where all neighbouring patterns m_N are included that have $HD(m_C, m_N) \leq d_t$. 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 $\mathbf{D}$ is inserted into $|\mathbf{a}_L|^{d_t}$ partitions, then the iterative linkage method will have a worst case complexity of $O(|\mathbf{a}_L|^{d_t} \cdot |\mathbf{D}|)$ 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, shown in Algorithm 4, 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. A single communication step of $O(|\mathbf{a}_L|^{d_t} \cdot |\mathbf{D}|)$ BFs is required from each DO to the LU.

Finally, the number of comparisons of pairs of BFs by the LU, as outlined in Algorithm 5, depends upon the blocking technique employed [8]. 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 $O((|\mathbf{a}_L|^{d_t} \cdot |\mathbf{D}|)^2/p)$, while for the batch method it would be $O((|\mathbf{a}_L|^{d_t} \cdot |\mathbf{D}|)^2)$.

8. Experimental evaluation

We now first describe the setup and then the data sets we used to evaluate our proposed approach to deal with missing values in the context of PPRL.

8.1. Experimental setup

In our evaluation we compared our approach with two baseline approaches. The first baseline is the k-nearest neighbour (named k-NN) based PPRL approach for dealing with missing values as proposed by Chi et al. [4]. In this approach, attribute-level Bloom filters (ABFs) are generated and used to compare records. We calculated the ABF weight scores, selected k-NN records, and performed the linkage as detailed by Chi et al. [4]. 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 then calculated the similarities between a record with a missing value and its k nearest neighbours using the Dice coefficient similarity (following Eq. 1) between the corresponding ABFs of the not missing attribute values. 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 developed by Durham et al. [20]. This approach does not consider missing values but has been shown to obtain high linkage quality for data sets without missing values. We first generated ABFs, and then sampled bits in these ABFs based on either setting all attribute weights equally to 1 (named EW), or by using the Fellegi and Sunter [22] 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 for each attribute using q-grams of length $q = 2$, and setting the number of hash functions to optimal such that around half of all bits in the ABFs are set to 1 [31]. As with the RBF baseline, we set the attribute weights, $\mathbf{w}$, in Algorithm 1, to be either all equal (EW) or to weights calculated using the approach by Fellegi and Sunter (FS) [22].

We compared the baseline approaches with both the iterative and batch methods discussed in Section 6, where we applied the three grouping methods *upper*, *lower*, and *both* (as per Definition 2). We also ran experiments without any grouping (named No group) to evaluate the impact of grouping on the obtained linkage quality. For our linkage methods we set the parameters $d_t = [1, 2, 3]$, $w_t = [1, 2, 3]$ 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.

In term of linkage quality, we used precision (ratio of correctly classified true matches over all classified matches) and recall (ratio of classified true matches over all true matches) [39]. We show precision and recall at different similarity thresholds s_t ranging from 0.5 to 1.0 in 0.1 steps, and measured results as precision-recall plots. We also show runtimes to evaluate the scalability of our approach as compared to the baselines.

To evaluate privacy, we used the cryptanalysis attack proposed by Christen et al. [16]. This attack aligns frequent BFs and plaintext values in a public database $\mathbf{G}$ with the aim to reidentify the most frequent values encoded in these BFs. We assume the LU acts as the adversary and tries to reidentify the attribute values encoded in the BFs sent to it by the DOs [16, 17, 18]. We conducted this attack assuming the worst-case scenario of the adversary gaining access to a database $\mathbf{D}$ of one DO, where $\mathbf{D} \equiv \mathbf{G}$, and trying to reidentify the values in $\mathbf{D}$ 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.

We implemented all approaches using Python 2.7 and ran experiments on a server with 2.4 GHz CPUs running Ubuntu 16.04. We will make our programs and data sets available to facilitate repeatability.

8.2. Generating data sets with missing values

To provide a realistic evaluation of our approach, we based all our experiments on a large real-world database, the North Carolina Voter Registration (NCVR) database as available from: `http://dl.ncsbe.gov/`. We used *FirstName*, *MiddleName*, *LastName*, *StreetAddress*, *City*, and *ZipCode*, as the set of linkage attributes, $\mathbf{a}_L$, because these are commonly used as quasi-identifiers in record linkage [1, 9, 14, 20].

To allow the evaluation of our approach on different database sizes and with data of different quality, we generated pairs of data sets by extracting records in a snapshot of the NCVR database from October 2019 with

Table 1: Number of records with missing values in the given attributes for the two data sets 100,000 records, where K = 1,000 records. Both data sets with 0% or 20% levels of corruption have the same numbers of records with missing values in the given attributes.

Data set	*FirstName*	*MiddleName*	*LastName*	*StreetAddress*	*City*	*ZipCode*
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

Table 2: Missingness patterns and their numbers of records in the data set pairs with 100,000 records, where K = 1,000 records, and the missingness pattern corresponds to quasi-identifying attributes *FirstName*, *MiddleName*, *LastName*, *StreetAddress*, *City*, and *ZipCode*.

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

7.6 million voter records. We modified the GeCo data corruptor [40] to generate various data sets with different data quality characteristics. During the corruption process we kept the identifiers (*VoterID*) of the selected and modified records, which allowed us to identify true matches and calculate linkage quality.

We generated pairs of data sets from the NCVR database that contained 50,000, 100,000, 500,000, and 1,000,000 records, respectively. For each size, we generated data set pairs that had 100% matching records (records with the same *VoterID*), where we then generated a corrupted version of these data sets by applying various corruption functions [40] on between 1 to 3 randomly selected attribute values on 20% of all records. As a result, while 80% of true matching record pairs were exact duplicates the remaining 20% of pairs were only approximate matching.

We then introduced missing values into 20% and 50% of records, respectively, for the quasi-identifying attributes *MiddleName*, *LastName*, *StreetAddress*, and *City*. We did not introduce any missing values into *FirstName* and *ZipCode* 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 quasi-identifying attributes.

We introduced different missingness patterns into the generated data sets 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 3.1. For MCAR, for each missingness pattern we randomly sampled a subset of records from a data set and introduced missing values into the corresponding attributes according to that missingness pattern. This ensures missing values are introduced into records without any patterns or correlations with any other attribute values in the same record.

For MAR, we introduced missingness patterns according to the *ZipCode* attribute by first randomly sampling subsets of records according to the different *ZipCode* values they have. Next, we selected a missingness pattern randomly for each such subset of records and then introduced missing values into the corresponding attributes according to that pattern. As a result, in the MAR data sets the missingness patterns are correlated with the values in the *ZipCode* attribute.

We also generated pairs of data sets that resulted in a common lattice that either had all layers in common or not, corresponding to types (2) and (3) of common lattices as we illustrated in Figs. 5 and 6.

In total we generated 64 data set pairs, 16 pairs for each of the four data set sizes. For each size we have generated eight pairs of MCAR and eight pairs of MAR data sets, four pairs each where missingness patterns result in a missing layer and four pairs that had missingness patterns in all lattice layers. Each of these four pairs consists of two with 0% and 20% corruptions, one each with 20% and 50% missing values, respectively.

Table 1 shows the number of records containing missing values in certain attributes in the data sets containing 100,000 records, and Table 2 shows the numbers of records with a specific missingness pattern in these data

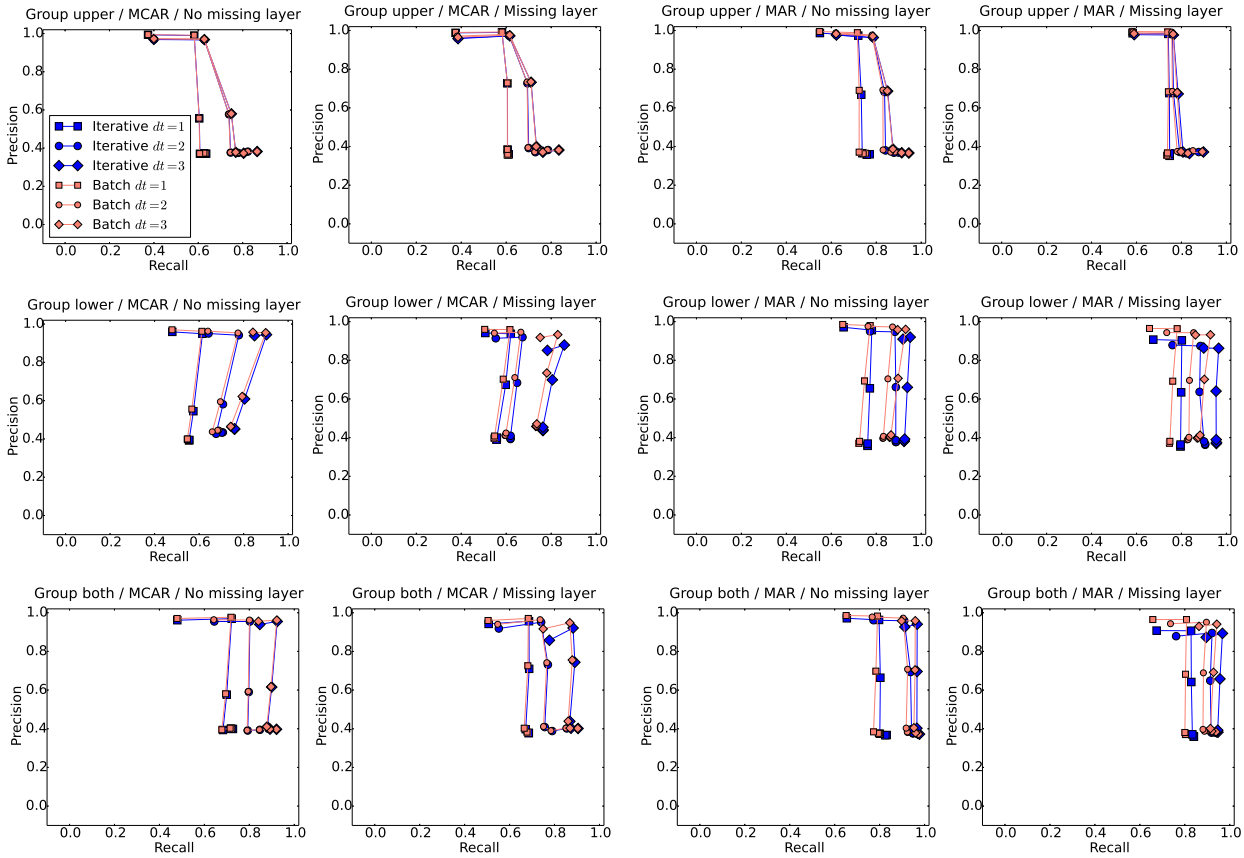

Figure 7: Precision and recall of the iterative and batch methods for different Hamming distance thresholds, d_t, for the different grouping methods (row-wise), and with MCAR in the left two and MAR in the right two columns (both with and without missing layers of patterns).

sets. The number of records with a certain missingness pattern are the same for MCAR and MAR data sets.

9. Results and discussion

We now present and discuss our results, starting with the linkage quality results we obtained with our approach compared to the baseline approaches, followed by a discussion of runtime results. We then assess the privacy of our approach compared to the baselines using a recently proposed attack method [16].

9.1. Linkage quality results

In Fig. 7 we show precision and recall results for our iterative and batch linkage approaches for different Hamming distance thresholds, 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 *upper* grouping method, patterns with more missing attribute values are grouped into partition(s) with less missing attribute values, and this can result in more BF pairs being classified as non-matches because they have lower numbers of 1-bits, potentially increasing the number of missed matches. With *lower* grouping, on the other hand, patterns with less missing attribute values are grouped into partitions that contain more missing values in their records. The corresponding BFs are constructed using less attributes which leads to an increase in the number of false matches when similarity thresholds are low, especially with data sets that contain a missing layer. This results in grouping *lower* to provide more false matches (but also more true matches) compared to grouping *upper*. As can be seen in the grouping *lower* plots in Fig. 7, recall values are generally higher than with grouping *upper*. The best results are obtained with grouping *both*, where each pattern is grouped with both of its neighbouring upper and lower partitions according to the missingness patterns.

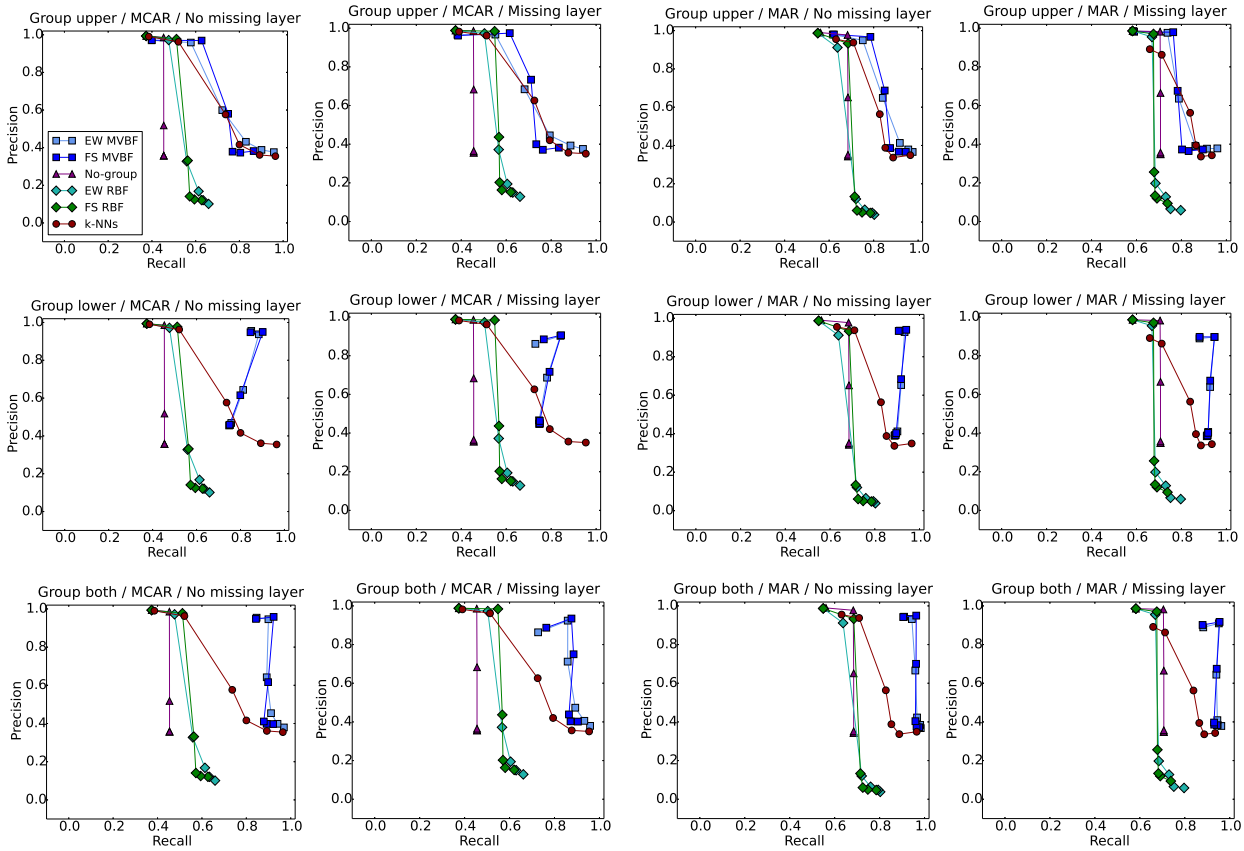

Figure 8: Precision and recall of our approach (named MVBF) as averaged over the iterative and batch methods, and the different data sets we described in Section 8 (different missingness and corruption percentages), for the different grouping methods (row-wise). We also show the baseline approaches RBF and k-NN, and our approach without any grouping applied (No group). Results for the data sets with MCAR are shown in the left two and MAR in the right two columns. EW refers to equal and FS to Fellegi-Sunter attribute weighting as discussed in Section 8.

Comparing the iterative and batch linkage methods, grouping *upper* provides similar results for both. However, iterative linkage slightly outperforms batch linkage when the two other grouping methods are used. This is because with batch linkage matched pairs of BFs cannot be removed after each iteration. The best match selection method employed by the DOs, described in Section 6.2, 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 be seen in both Figs. 7 and 8, rather unexpectedly recall becomes lower for the *lower* grouping method for lower similarity thresholds (while one would expect recall to increase with lower thresholds). This is because true matching record pairs that do contain missing values will occur in partitions that are handled in later iterations. However, because less attributes are compared with the *lower* grouping method, more false matches are classified in the earlier iterations, and once such records are matched they are removed from later iterations (and therefore not compared to their true matching record in the other database).

In Fig. 8 we compare our approach to the RBF [20] and k-NN [4] baselines, as well as our approach without any pattern grouping 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 (equal (EW) or based on the Fellegi and Sunter [22] (FS) weight calculation) is small, with FS outperforming 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 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.

Table 3: Average runtimes (in seconds) used by a database owner (DO) and the linkage unit (LU) for linking data sets of different sizes using the different linkage approaches. * indicates estimated runtimes (using linear regression) for the batch method for the data sets with 1 million records.

Linkage approach		50,000 DO / LU	100,000 DO / LU	500,000 DO / LU	1,000,000 DO / LU
RBF		364.5 / 0.4	1101.4 / 1.9	4959.0 / 41.1	9681.2 / 135.9
k-NN		2509.9 / 510.4	7233.4 / 1655.7	99626.0 / 37433.5	190486.3 / 151605.8
Iterative, g = *upper*	$d_t = 1$	529.1 / 0.6	616.4 / 0.8	5320.8 / 14.5	8465.8 / 38.5
	$d_t = 2$	673.7 / 0.8	978.0 / 1.2	5641.7 / 13.0	11652.4 / 42.3
	$d_t = 3$	625.7 / 0.7	1080.3 / 1.3	6275.7 / 12.4	11233.3 / 37.6
Iterative, g = *lower*	$d_t = 1$	682.0 / 0.6	808.9 / 0.9	7031.5 / 14.6	12163.7 / 35.4
	$d_t = 2$	892.0 / 0.7	1328.9 / 1.2	7876.5 / 12.1	14513.8 / 28.1
	$d_t = 3$	922.3 / 0.7	1556.3 / 1.4	9165.7 / 11.0	16577.8 / 33.2
Iterative, g = *both*	$d_t = 1$	733.5 / 0.6	857.7 / 0.9	7260.0 / 15.3	11773.2 / 31.3
	$d_t = 2$	851.7 / 0.7	1465.1 / 1.3	8360.8 / 12.8	15531.3 / 33.4
	$d_t = 3$	1002.1 / 0.8	1679.1 / 1.3	9120.1 / 12.5	17429.8 / 35.5
Batch, g = *upper*	$d_t = 1$	2160.9 / 0.4	4110.4 / 0.8	68946.5 / 11.4	145438.6* / 24.0*
	$d_t = 2$	3220.6 / 0.7	8225.3 / 1.3	117859.8 / 21.6	248467.7* / 45.5*
	$d_t = 3$	3523.8 / 0.7	10029.4 / 1.6	134283.4 / 18.3	282924.1* / 38.4*
Batch, g = *lower*	$d_t = 1$	4226.1 / 1.8	10374.7 / 4.1	153330.5 / 54.7	323355.7* / 114.9*
	$d_t = 2$	5604.4 / 2.4	15028.0 / 6.1	216800.7 / 101.2	457330.3* / 214.0*
	$d_t = 3$	7872.4 / 4.0	21502.2 / 9.5	268216.0 / 190.5	563877.9* / 404.1*
Batch, g = *both*	$d_t = 1$	4363.9 / 1.9	11255.4 / 3.8	157145.9 / 56.5	331116.9* / 119.0*
	$d_t = 2$	5942.3 / 2.7	18343.7 / 9.1	229025.5 / 105.6	482070.6* / 222.1*
	$d_t = 3$	8996.9 / 4.4	24915.9 / 10.3	276997.2 / 168.5	580565.6* / 356.0*

The k-NN baseline [4], the only previously proposed PPRL method that can handle missing values, generally shows both lower precision and recall results compared to our approach. The k-NN method 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. This indicates that the k-NN approach is highly sensitive with regard to this imputation process. This is especially the case for lower similarity thresholds where the k-NN approach more likely will select less similar neighbouring records that will result in incorrect matched record pairs. As can be also seen, k-NN performs better on the MAR compared to the MCAR data sets, which is because MAR results in groups of records that have more similar neighbouring records with the same missingness pattern. It therefore 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.

As can also be seen from Fig. 8, the RBF baseline [20] resulted in more missed matches for these databases 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 can lead to 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 and therefore more false matches.

Furthermore, as can be seen in both Figs. 7 and 8, the linkage results obtained with the MAR data sets are generally better than with the MCAR data sets. This is because records in the MAR data sets have been generated such that they have the same missingness patterns if they have the same *ZipCode* value. This indicates that our grouping based approach can make use of MAR patterns leading to improved linkage quality results.

9.2. Runtime results

Table 3 shows the average runtimes required by a database owner (DO) and the linkage unit (LU) for the different linkage approaches. 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

Table 4: Average number of records sent to the linkage unit in the batch linkage method on the MCAR data sets of different sizes.

Grouping method	Number of records	$d_t = 1$	$d_t = 2$	$d_t = 3$
$g = upper$	50,000	62,250 / 58,250	103,625 / 103,876	112,625 / 113,501
	100,000	124,500 / 116,500	207,250 / 207,750	225,250 / 227,000
	500,000	622,501 / 582,500	1,036,251 / 1,038,750	1,126,251 / 1,135,000
	1,000,000	1,247,501 / 1,170,001	2,096,251 / 2,101,251	2,277,501 / 2,295,001
$g = lower$	50,000	136,000 / 140,000	208,500 / 205,875	265,875 / 262,375
	100,000	272,000 / 280,000	417,000 / 411,750	531,750 / 524,750
	500,000	1,360,000 / 1,400,000	2,085,000 / 2,058,750	2,658,750 / 2,623,750
	1,000,000	2,822,500 / 2,900,000	4,280,001 / 4,225,001	5,427,500 / 5,355,000
$g = both$	50,000	154,875 / 154,875	250,125 / 249,501	316,500 / 315,625
	100,000	309,750 / 309,750	500,250 / 499,000	633,000 / 631,250
	500,000	1,548,750 / 1,548,750	2,501,251 / 2,495,000	3,165,001 / 3,156,250
	1,000,000	3,200,000 / 3,202,500	5,126,251 / 5,113,751	6,455,001 / 6,437,501

therefore less records are included in the following partitions. On the other hand, with the batch method the whole database is sent to the LU once only, where each record likely occurs in multiple partitions.

This is illustrated in Table 4, where we show the average number of records that each DO sends to the LU for the MCAR data sets generated using the batch linkage method. As can be seen, the number of records increases as the Hamming distance threshold, d_t, is increased because this means a record with a certain missingness pattern will be inserted into a larger number of partitions. As can also be seen, for the data sets with 1 million records and using grouping *both*, less than 7 million RBFs are being generated, resulting in a less than seven-fold overhead of the batch method.

As can be seen from Table 3, runtimes are higher for grouping *lower* compared to grouping *upper* for the iterative linkage method. This is because with grouping *upper* the first partition will contain more BFs (from partitions that contain more missing attribute values) compared to grouping *lower*, and more BFs will be matched in the first iteration. As a result, these BFs will not be included in later partitions. With the grouping *lower* method, however, a smaller partition is processed in the first iteration, and BFs with different missingness patterns will be included in later iterations, which therefore become larger because BFs are only matched later. This results in overall longer runtimes for grouping *lower*.

With grouping *both*, we obtain runtime results comparable to the grouping *lower* method. These runtimes depend upon how many records in the first partition, which is generally the largest, are classified as matches and are therefore removed. Given grouping *both* has shown to achieve the best linkage results, as we discussed in Section 9.1, we recommend to use this grouping method over the others (for both the iterative and batch linkage methods).

For the iterative approach 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 Hamming distance threshold, d_t, the actual similarities between BFs also affect runtimes. As can be seen from Table 3, the LU requires higher runtimes with *upper* than *lower* because the LU compares more BF pairs in the first partition, and where the not similar pairs (that are not classified as matches) are again included in the following partitions. Even for $d_t = 1$ does the LU require longer runtimes because smaller partitions are generated and therefore less BF pairs can be matched in the earlier iterations.

Comparing our approach with the baseline approaches, the RBF approach resulted in the lowest runtime because each record is only encoded into one RBF and then sent to the LU once for comparison. However, the k-NN approach resulted 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.

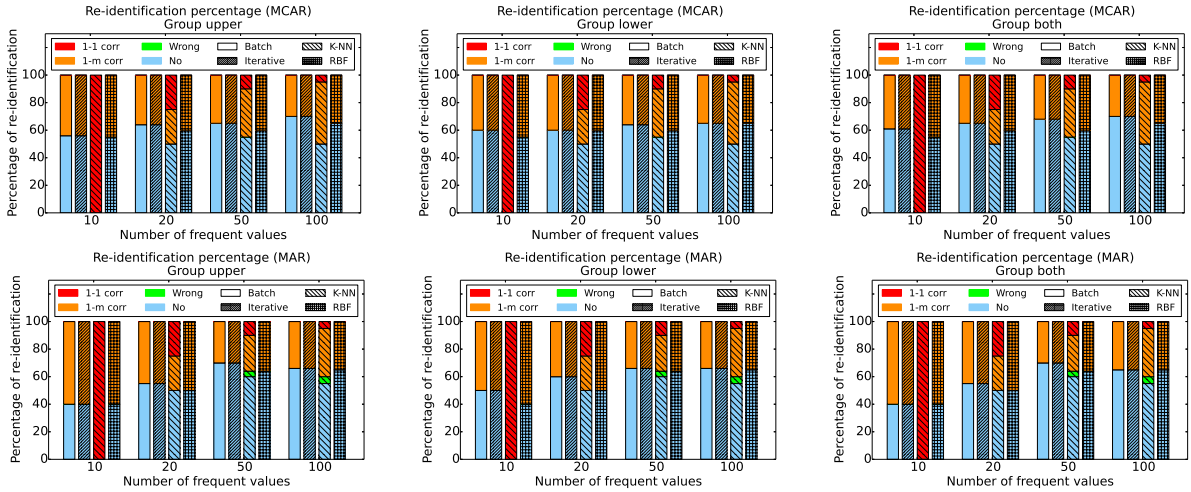

Figure 9: Reidentification results for the MCAR (top row) and MAR (bottom row) data sets, where grouping upper (left column), grouping lower (middle column), and grouping both (right column) were applied. The bars show the percentages of BFs for which attribute values were reidentified either correctly with only 1 (1-1 corr) or several (1-m corr) values, or wrong or no attribute value(s) were reidentified.

9.3. Privacy results

Figure 9 shows the reidentification results using the frequency-alignment based cryptanalysis attack proposed by Christen et al. [16] applied on all evaluated approaches with both the MCAR and MAR data sets. We assessed the reidentification accuracy of the attack by calculating the percentages of (1) correctly reidentified attribute 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 attribute values from the plaintext data set, respectively [16].

As can be seen from Fig. 9, 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 data sets compared to MCAR data sets. 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 Fig. 9 shows, the k-NN baseline provides the weakest privacy protection (the attack achieved the best reidentification results) for both MCAR and MAR data sets with one-to-one correct reidentifications 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 reidentifications. 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 more privacy due to the weight distribution and random bit sampling process it uses. The reidentification results show that our approach provides stronger privacy compared to the baselines which makes it more applicable in realistic PPRL scenarios.

10. Conclusions and future work

We have presented a novel approach to consider missing values in privacy-preserving record linkage (PPRL) using record-level Bloom filter (BF) encoding. Our approach is based on a lattice of missingness patterns, 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. We conducted an extensive analysis and experimental evaluation of our approach with databases of different sizes and with different patterns and numbers of missing values. Our results show that our approach can

achieve high linkage quality, where it substantially outperforms two baseline approaches while providing privacy against cryptanalysis attacks. As future work, we aim to extend our experiments to data sets with other missing types of data, such as missing not at random. We also aim to improve the scalability of our approach by applying filtering to remove redundant record comparisons in the batch linkage method.

Acknowledgements

This research has been partially funded by the Australian Research Council under project DP160101934. We like to thank Anushka Vidanage for his assistance.

References

[1] P. Christen, Data Matching – Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection, Springer, Heidelberg, 2012.

[2] T. N. Herzog, F. J. Scheuren, W. E. Winkler, Data Quality and Record Linkage Techniques, Springer, New York, 2007.

[3] I. C. Anindya, M. Kantarcioglu, B. Malin, Determining the impact of missing values on blocking in record linkage, in: PAKDD, Melbourne, 2019, pp. 262–274.

[4] Y. Chi, J. Hong, A. Jurek, W. Liu, D. O'Reilly, Privacy preserving record linkage in the presence of missing values, Elsevier IS 71 (2017) 199–210.

[5] J. Ferguson, A. Hannigan, A. Stack, A new computationally efficient algorithm for record linkage with field dependency and missing data imputation, Elsevier IJMI 109 (2018) 70–75.

[6] H. Goldstein, K. Harron, Record linkage: a missing data problem, Methodological Developments in Data Linkage (2015).

[7] T. C. Ong, M. V. Mannino, L. M. Schilling, M. G. Kahn, Improving record linkage performance in the presence of missing linkage data, Elsevier JBI 52 (2014) 43–54.

[8] P. Christen, T. Ranbaduge, R. Schnell, Linking Sensitive Data – Methods and Techniques for Practical Privacy-Preserving Information Sharing, Springer, Heidelberg, 2020.

[9] D. Vatsalan, P. Christen, V. Verykios, A taxonomy of privacy-preserving record linkage techniques, Information Systems 38 (6) (2013) 946–969.

[10] A. Gkoulalas-Divanis, D. Vatsalan, D. Karapiperis, M. Kantarcioglu, Modern privacy-preserving record linkage techniques: An overview, IEEE TIFS (2021).

[11] R. Pita, C. Pinto, S. Sena, R. Fiaccone, L. Amorim, S. Reis, M. Barreto, S. Denaxas, M. Barreto, On the accuracy and scalability of probabilistic data linkage over the Brazilian 114 million cohort, IEEE JBHI 22 (2) (2018) 346–353.

[12] S. Randall, A. Ferrante, J. Boyd, et al., Privacy-preserving record linkage on large real world datasets, Elsevier JBI 50 (2014) 205–212.

[13] K. Schmidlin, K. M. Clough-Gorr, A. Spoerri, Privacy preserving probabilistic record linkage (P3RL): a novel method for linking existing health-related data and maintaining participant confidentiality, BMC MRM 15 (1) (2015) 46.

[14] R. Schnell, T. Bachteler, R. J., Privacy-preserving record linkage using Bloom filters, BMC MIDM 9 (1) (2009).

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

[16] P. Christen, T. Ranbaduge, D. Vatsalan, et al., Precise and fast cryptanalysis for Bloom filter based privacy-preserving record linkage, IEEE TKDE 31 (11) (2018) 2164–2177.

[17] M. Kuzu, M. Kantarcioglu, E. Durham, B. Malin, A constraint satisfaction cryptanalysis of Bloom filters in private record linkage, in: PETS, Waterloo, Canada, 2011, pp. 226–245.

[18] F. Niedermeyer, S. Steinmetzer, M. Kroll, R. Schnell, Cryptanalysis of basic Bloom filters used for privacy preserving record linkage, JPC (2014).

[19] C. Dong, L. Chen, Z. Wen, When private set intersection meets big data: an efficient and scalable protocol, in: ACM SIGSAC, Berlin, 2013, pp. 789–800.

[20] E. A. Durham, M. Kantarcioglu, Y. Xue, C. Toth, M. Kuzu, B. Malin, Composite bloom filters for secure record linkage, IEEE TKDE 26 (12) (2014) 2956–2968.

[21] H. Newcombe, J. Kennedy, S. Axford, A. James, Automatic linkage of vital records, Science 130 (3381) (1959) 954–959.

[22] I. P. Fellegi, A. B. Sunter, A theory for record linkage, JASA 64 (328) (1969) 1183–1210.

[23] A. P. Dempster, N. M. Laird, D. B. Rubin, Maximum likelihood from incomplete data via the EM algorithm, Wiley JRSS-B 39 (1) (1977) 1–22.

[24] L. Dusserre, C. Quantin, H. Bouzelat, A one way public key cryptosystem for the linkage of nominal files in epidemiological studies, Medinfo 8 (1995) 644–647.

[25] Y. Lindell, B. Pinkas, Secure multiparty computation for privacy-preserving data mining, JPC 1 (1) (2009).

[26] M. Kantarcioglu, A. Inan, W. Jiang, B. Malin, Formal anonymity models for efficient privacy-preserving joins, Elsevier DKE 68 (11) (2009) 1206–1223.

[27] D. Vatsalan, P. Christen, Scalable privacy-preserving record linkage for multiple databases, in: ACM CIKM, Shanghai, 2014, pp. 1795–1798.

[28] A. Vidanage, T. Ranbaduge, P. Christen, R. Schnell, Efficient pattern mining based cryptanalysis for privacy-preserving record linkage, in: IEEE ICDE, Macau, 2019, pp. 1698–1701.

[29] R. J. A. Little, D. B. Rubin, Statistical Analysis with Missing Data, 3rd Edition, Wiley, Hoboken, 2020.

[30] J. Han, M. Kamber, Data mining: concepts and techniques, 2nd Edition, Morgan Kaufmann, 2006.

[31] D. Vatsalan, P. Christen, Privacy-preserving matching of similar patients, Elsevier JBI (2016).

[32] D. Karapiperis, A. Gkoulalas-Divanis, V. S. Verykios, FEDERAL: A framework for distance-aware privacy-preserving record linkage, IEEE TKDE 30 (2) (2017) 292–304.

[33] R. Schnell, C. Borgs, Encoding hierarchical classification codes for privacy-preserving record linkage using Bloom filters, in: ECML/PKDD DINA, Würzburg, 2019, pp. 142–156.

[34] T. Ranbaduge, R. Schnell, Securing Bloom filters for privacy-preserving record linkage, in: ACM CIKM, Galway, 2020, pp. 2185–2188.

[35] M. Kroll, S. Steinmetzer, Cryptanalysis of Bloom filter encryptions of databases with several personal identifiers, in: BIOSTEC, Lisbon, 2015, pp. 341–356.

[36] R. Schnell, T. Bachteler, J. Reiher, A novel error-tolerant anonymous linking code, Working paper, GRLC (2011).

[37] D. Karapiperis, V. S. Verykios, An LSH-based blocking approach with a homomorphic matching technique for privacy-preserving record linkage, IEEE TKDE 27 (4) (2015) 909–921.

[38] M. Kuzu, M. Kantarcioglu, A. Inan, E. Bertino, E. Durham, B. Malin, Efficient privacy-aware record integration, in: EDBT, Genoa, 2013, pp. 167–178.

[39] D. Hand, P. Christen, A note on using the F-measure for evaluating record linkage algorithms, Stat Comput 28 (3) (2018).

[40] P. Christen, D. Vatsalan, Flexible and extensible generation and corruption of personal data, in: ACM CIKM, San Francisco, 2013, pp. 1165–1168.