Privacy Preserving Record Linkage with PPJoin

Ziad Sehili[1], Lars Kolb[1], Christian Borgs[2], Rainer Schnell[2], Erhard Rahm[1]

[1]University of Leipzig
{sehili,kolb,rahm}@informatik.uni-leipzig.de

[2]University of Duisburg-Essen
{rainer.schnell,christian.borgs}@uni-due.de

Abstract:
Privacy-preserving record linkage (PPRL) becomes increasingly important to match and integrate records with sensitive data. PPRL not only has to preserve the anonymity of the persons or entities involved but should also be highly efficient and scalable to large datasets. We therefore investigate how to adapt PPJoin, one of the fastest approaches for regular record linkage, to PPRL resulting in a new approach called P4Join. The use of bit vectors for PPRL also allows us to devise a parallel execution of P4Join on GPUs. We evaluate the new approaches and compare their efficiency with a PPRL approach based on multibit trees.

1 Introduction

The record linkage process tries to find pairs of entities across different databases that refer to the same real-wold object. Beside the field of data integration, it is increasingly used in research applications, for example in medicine, the social sciences and official statistics. In these fields, protecting the identifiers of the entities is usually required by law. Therefore, if such linkages are permitted, special techniques protecting the identifiers have to be used. The set of techniques for record linkage without revealing identifiers is called *Privacy Preserving Record Linkage* or *PPRL*. Due to the many applications, PPRL is an active field of research in Computer Science, Statistics and some application fields as Epidemiology, Health Service Research and Survey Methodology [Sch15].

Like traditional approaches for record linkage, PPRL has an inherent scalability problem if each (encrypted) record needs to be compared with each other record resulting in a quadratic complexity. The usual means to improve efficiency and thus scalability to larger datasets is to reduce the search space, e.g. by appropriate filter and blocking techniques, or/and to perform record linkage in parallel on many processors [Chr12b, KTR12]. PPJoin (Position Prefix Join) [XWLY08] is an efficient approach for regular record linkage exploiting several filters to reduce the search space; its efficiency has been confirmed in independent evaluation studies for diverse datasets [KTR10, JLFL14]. We will therefore investigate how to adapt this scheme to PPRL where records are encrypted by bit vectors. We will also propose and evaluate the parallel execution of the adapted PPJoin scheme on graphical processing units (GPUs).

Our contributions are thus as follows

- We present how PPJoin can be adapted to evaluate the similarity of bit vectors for PPRL (Section 4).
- We show how the adapted PPJoin approach, which we call P4Join (Privacy-Preserving PPJoin), can be executed in parallel on GPUs (Section 5).
- We evaluate the efficiency of the new PPJoin approaches (Section 6). The evaluation also includes a comparison with a previously proposed approach based on multibit trees [BRS13].

Before outlining these contributions we begin with preliminaries on the assumed approaches for record linkage and PPRL (Section 2) and a discussion of related work including the multibit tree approach (Section 3). At the end, we summarize and close with an outlook.

2 Preliminaries

For record linkage we apply so-called similarity joins that determine all pairs of records with a similarity above a minimal threshold. We first introduce the notion of such similarity joins together with a simple length filter that can be utilized to reduce the number of necessary comparisons. Furthermore, we introduce the assumed model for encrypting records by bit arrays to support a privacy-preserving record linkage.

2.1 Set similarity joins and length filter

To detect duplicate records between two heterogeneous data sources R and S, similarity joins identify all pairs $(r,s) \in R \times S$ with a similarity above a given threshold t, i.e. $Sim(x,y) \geq t$ for comparable attribute values x and y. The efficient processing of such similarity joins for different kinds of similarity measures has been the focus of much research in the past, e.g., [AGK06, HCKS08, MF12, RLW$^+$13, SHC14, WLF10, XWL08, XWLY08]. In this work we focus on the popular *Jaccard* similarity because it is well suited for both string-tokenized records and bit arrays. Given two records with attribute values x and y, represented as (multi-) sets of tokens (e.g. n-grams), the Jaccard similarity is defined as:

$$Sim_{Jaccard}(x,y) = \frac{|x \cap y|}{|x \cup y|} = \frac{|x \cap y|}{|x| + |y| - |x \cap y|}. \tag{1}$$

Example: Consider two records $x = tomas$ and $y = tommas$ tokenized to bigrams as shown in Fig. 1. The resulting Jaccard similarity is 6/7, since all (6) bigrams of x overlap with the 7 bigrams of y.

The similarity function $Sim_{Jaccard}$ allows the application of a simple *length filter* to avoid the evaluation of the Cartesian product to identify all pairs of similar records. This is because the minimal similarity (token overlap) can only be achieved if the lengths of the input records do not deviate too much. Formally, for two records x and y with $|x| \leq |y|$, it holds that

$$Sim_{Jaccard}(x, y) \geq t \Rightarrow |x| \geq \lceil t \cdot |y| \rceil \tag{2}$$

For example, two records cannot satisfy a similarity threshold $t = 0.8$ if their lengths differ by more than 20%. If the first record has, say, ten tokens a comparison is no longer needed for all records with less than eight or more than 12 tokens. The length filter can thus achieve a substantial reduction of the search space and is used in many implementations including PPJoin and Multibit Trees.

2.2 Privacy Preserving Record Linkage

For privacy preservation, we consider the detection of duplicate records in fully encrypted datasets. We therefore encrypt the records' attribute values individually *before* passing them to a semi-trusted third party that run the record linkage process.

To encrypt a single record, we map all n-grams of the (relevant) attribute values to a bit vector (array) of fixed size as proposed in [SBR11]. Specifically, each n-gram (of each n-gram set) is hashed to *multiple* bits by applying k independent hash functions, each defining an index of a bit which is set to one. This can be achieved by a double hashing scheme combining two independent base hash functions f and g to determine the k hash values $h_1(x), \ldots, h_k(x)$ for each n-gram x [KM06]:

$$h_i(x) = (f(x) + i \cdot g(x)) \mod l.$$

As base hash functions, [SBR09] proposed the usage of two keyed hash message authentication codes (HMACs), namely, HMAC-SHA1 and HMAC-MD5 for f and g, respectively. The mapping of records and their n-grams to bit arrays is illustrated in Figure 1 for the two names *tomas* and *tommas*.

The similarity between two records given by the bit arrays (or fingerprints) x and y can now be determined analogously to determining the Jaccard similarity by measuring the degree of overlap between the set bit positions. The resulting similarity is also known as the *Tanimoto* similarity and can be expressed as:

$$Sim_{Tanimoto}(x, y) = \frac{|x \wedge y|}{|x \vee y|} = \frac{|x \wedge y|}{|x| + |y| - |x \wedge y|} \tag{3}$$

with $|x|$ denoting the number of set bits (or cardinality) in the bit array x.

For the example in Figure 1 we have 11 set bits in the intersection of the two bit arrays and 12 in the union resulting in a similarity of $11/12$.

It was shown in [SBR09] that the described encryption scheme is similarity-preserving. It also allows applying the length filter introduced before, i.e. two records can only meet a minimal similarity if their number of set bits does not differ too much.

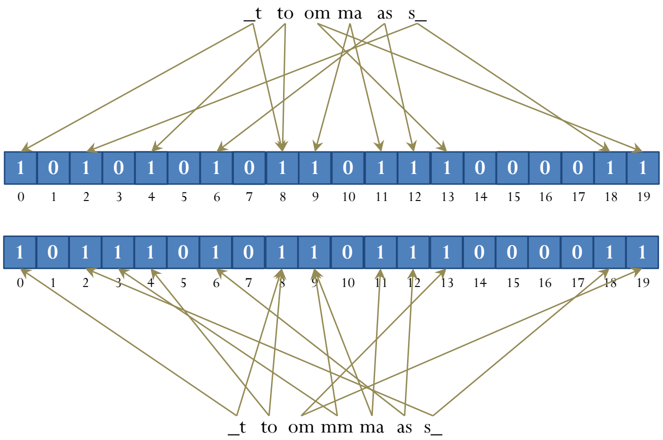


Figure 1: Example of the encryption of two names *tomas* and *tommas*, each tokenized to bigrams, using $k = 2$ hash functions and bit arrays of size 20 bits.

3 Related Work

Record Linkage Record linkage, entity resolution or data matching is the problem of finding similar records referring to the same real-world entity. It has been addressed by numerous research studies and approaches as summarized in several surveys [EIV07, Chr12b, KR10]. Key challenges include achieving complete and accurate results at a good efficiency and scalability. Many approaches aim at improving efficiency by reducing the search space, e.g. by so-called indexing and blocking techniques [DN11, Chr12a]. Furthermore, parallel record linkage approaches have beend devised, e.g. for MapReduce/Hadoop platforms [VCL10, KTR12] or by using graphic processors (GPUs) [FPS$^+$13, NKH$^+$13]. The use of similarity joins [Coh00] for finding all pairs of records with a certain minimal similarity allows several performance improvements by exploiting charactistics of the considered similarity measure and the prespecified similarity threshold. This holds especially for the broad class of so-called *signature-based similarity joins* [AGK06] where the comparison of records is based on their signatures, such as the set of tokens (e.g., n-grams) for selected attributes. In this case, one cannot only utilize the introduced length filter but also exclude all pairs of records that do not share at least one token in their signature. Further proposed optimizations for such similarity joins include the use of a so-called prefix filter and dynamically created inverted indexes [BMS07]. The PPJoin approach [XWLY08] includes these and further optimizations for improved efficiency. Since this algorithm is the basic of our work, it will be presented in detail in Section 4.

Privacy Preserving Record Linkage In [VCV13], a comprehensive taxonomy and survey of privacy-preserving record linkage techniques is provided covering different encryption schemes, linkage and scalability approaches. The approach of [ALM05] uses hashing to map TF/IDF tokens of records to bit vectors for privacy-preserving similarity joins. It applies a token-based blocking to find candidate pairs that are then compared using the Jaccard similarity function but uses no further filtering to eliminate dissimilar pairs of records.

Similarity Filtering with Multibit Trees The use of multibit trees promises a better efficiency for PPRL and will be considered in our evaluation for comparison purposes. The approach was originally suggested to rapidly query large databases of chemical *fingerprints* (bit arrays) [KNP10]. A query bit array A is being searched for in a database B, with the aim to retrieve all elements in B whose similarity with A is above the threshold t. Multibit Trees are based on the Tanimoto coefficient presented in Eq. 3 as a similarity measure.

As suggested in [BRS13], Multibit trees can easily be used for PPRL with the encryption scheme described in Section 2. If we have two files with records to compare with each other, we can use the larger file to build up the multibit trees and use each record (fingerprint) of the second file for searching similar records. The time taken to build up the index structure is therefore much less important than the query time, which increases linearly with the size of the query file.

The original paper [KNP10] describes several variations and we focus here on the multibit tree scheme that has shown the best evaluation results in [BRS13] and [Sch14] and also outperformed blocking approaches such as Sorted Neighborhood and Canopy Clustering. In this scheme, we partition the fingerprints into buckets according to their lengths such that all fingerprints with the same number of set bits belong to the same partition (or bucket). To apply the length filter, we can then restrict the search for similar fingerprints to the partitions meeting the length criterion of Eq. 2.

Query efficiency is further improved by organizing all fingerprints of a partition within a multibit tree. A multibit tree is a binary tree to iteratively assign fingerprints to its nodes based on so-called match bits. A match bit refers to a specific position of the bit array and can be 1 or 0: it indicates that all fingerprints in the associated subtree share the specified match bit. When building up the multibit tree, one match bit or multiple such bits are selected in each step so that the number of unassigned fingerprints can be roughly split by half (”half clustering” strategy). The split is continued as long as the number of fingerprints per node does not fall under a limit ([KNP10] recommends a limit of 6). The match bits can then be used for a query fingerprint to determine the maximal possible similarity for subtrees when traversing the tree and can thereby eliminate many fingerprints to evaluate.

4 Adaptation of PPjoin for Encrypted Data

PPJoin (Position Prefix Join) [XWLY08] is a signature-based similarity join algorithm that applies several optimizations for improved efficiency, in particular the length filter, a prefix filter and a position filter. We first describe the prefix filter already proposed in [CGK06]. We then outline PPJoin and its adaptation for PPRL.

4.1 Prefix Filter

The prefix filter [CGK06] for signature- or token-based similarity joins exploits that similar records must share at least one token. Depending on the similarity threshold and length of the records, the overlap must even occur for a subset of the tokens. For example, two records with ten tokens each can only meet a minimal similarity of 0.8 if they overlap in at least nine tokens. Hence it is sufficient to check whether subsets of two tokens each overlap; without such an overlap in the subsets the record pair cannot match and safely be eliminated from further consideration. To maximize this filter idea one builds the subsets with the least frequent tokens; i.e. one takes the prefix of a record's tokens ordered by their overall frequency.

Given two records x and y represented as sorted token sets (using the same ordering), the overlap between their tokens is defined as:

$$Overlap(x, y) = |x \cap y| \tag{4}$$

From the similarity join definition using the Jaccard similarity (Eq. 1) we can derive the required minimal overlap in order to meet the similarity threshold t:

$$Sim_{jaccard}(x, y) \geq t \Leftrightarrow Overlap(x, y) \geq \lceil \frac{t}{1+t} \cdot (|x| + |y|) \rceil = \alpha \tag{5}$$

Pairs of records with an overlap less than the *minimal overlap* α cannot meet the similarity join condition so that the relatively expensive similarity computation can be saved for them.

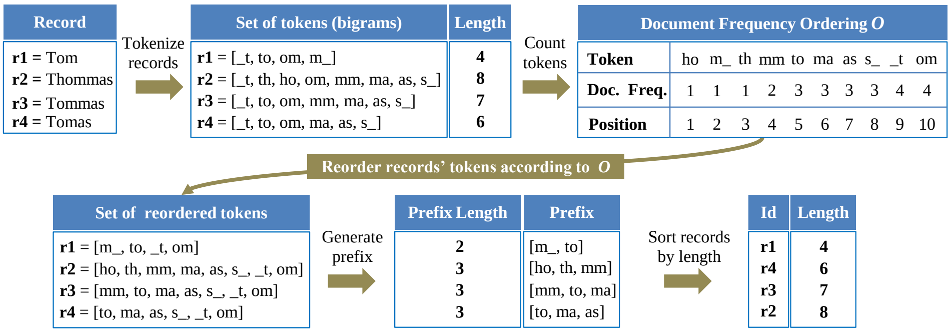
The prefix filter uses this observation to determine prefixes of sufficient size so that they have to overlap to meet the minimal similarity. This is the case for prefix lengths $|x|-\alpha+1$ and $|y| - \alpha + 1$ for x and y, respectively. For the mentioned example above $\alpha = 9$ and the prefixes have to be of length 2.

4.2 Position Prefix Join (PPJoin)

As mentioned before, PPJoin is a set similarity join algorithm using the Jaccard similarity to find *all* similar pairs (x, y), if $Sim_{Jaccard}(x, y) \geq t$. To do this in an efficient way, PPJoin applies length filtering, prefix filtering and position filtering [1]. As in [XWLY08], we describe PPJoin for finding similar records in a single input file R. The more general case with two (duplicate-free) input files can be mapped to this case by merging the two files into one and observing that only records from different input files are compared with each other [VCL10].

In order to apply the length and prefix filters, a preprocessing step is executed first followed by the actual filtering and record comparisons.

[1] [XWLY08] also proposed the use of a suffix filter but this filter is not considered here due to its likely high overhead for encrypted data.



Record
r1 = Tom
r2 = Thommas
r3 = Tommas
r4 = Tomas

Set of tokens (bigrams)	Length
r1 = [_t, to, om, m_]	**4**
r2 = [_t, th, ho, om, mm, ma, as, s_]	**8**
r3 = [_t, to, om, mm, ma, as, s_]	**7**
r4 = [_t, to, om, ma, as, s_]	**6**

Document Frequency Ordering *O*										
Token	ho	m_	th	mm	to	ma	as	s_	_t	om
Doc. Freq.	1	1	1	2	3	3	3	3	4	4
Position	1	2	3	4	5	6	7	8	9	10

Set of reordered tokens
r1 = [m_, to, _t, om]
r2 = [ho, th, mm, ma, as, s_, _t, om]
r3 = [mm, to, ma, as, s_, _t, om]
r4 = [to, ma, as, s_, _t, om]

Prefix Length	Prefix
2	[m_, to]
3	[ho, th, mm]
3	[mm, to, ma]
3	[to, ma, as]

Id	Length
r1	**4**
r4	**6**
r3	**7**
r2	**8**

Figure 2: Preprocessing step: The records are first tokenized (to bigrams) and the tokens' frequencies are determined. The resulting document frequency ordering O is then used to order the tokens of the records according to their frequency. After the computation of their prefixes for a threshold $t = 0.8$, the records are sorted by ascending length.

(a) Preprocessing: The main goals of preprocessing are to determine the lengths and prefixes of all records and to sort the records according to their lengths. Determining the prefixes requires to determine the tokens' frequencies and reorder the tokens per record. These tasks are achieved in two passes. First PPJoin reads the records sequentially, determines their lengths and counts the occurrence of each token in *all* the records. The tokens are then sorted by ascending frequency which corresponds to the *document frequency ordering* in IR. In the second pass, the tokens of each record are sorted according to the overall frequency, i.e. from the least frequent to the most frequent token. Furthermore for each record x, the prefix $pref(x)$ is computed. According to Eq. 5, the minimal overlap and thus the prefix length also depends on the length of the record y to which x is to be compared. Since these other records are yet unknown and of varying length, PPJoin applies the following safe prefix length retaining all true matches:

$$|pref(x)| = \lceil (1 - t) \cdot |x| \rceil + 1 \tag{6}$$

At the end of preprocessing the records are sorted by length to support the use of the length filter in the next step.

Figure (2) illustrates the preprocessing for an input of four records $r_1 = Tom$, $r_2 = Thommas$, $r_3 = Tommas$ and $r_4 = Tomas$ and a Jaccard similarity threshold $t = 0.8$.

(b) Comparison of records: In the second step, PPJoin jointly applies the length and prefix filters while processing the records in ascending order of their lengths. To quickly determine the records with overlapping prefixes, PPJoin builds up an inverted index I on-th-fly recording per token all (relevant) records having this token in their prefix.

We explain this process by using the four records returned by the preprocessing shown in Figure 2. The first (shortest) record r_1 has length 4 and prefix $pref(r_1) = [m_{\_}, to]$. The index I is initialized with the two tokens pointing to record $r1$ (Figure 3a). The next record r_4 has length 6 and prefix $pref(r_4) = [to, ma, as]$. The length filter check reveals that $|r_1| < \lceil t \cdot |r_4| \rceil$ $(4 < \lceil 0.8 \cdot 6 \rceil)$ so that $r1$ does not have to be compared against $r4$ and any further records which have at least length 6. Hence, when adding the new

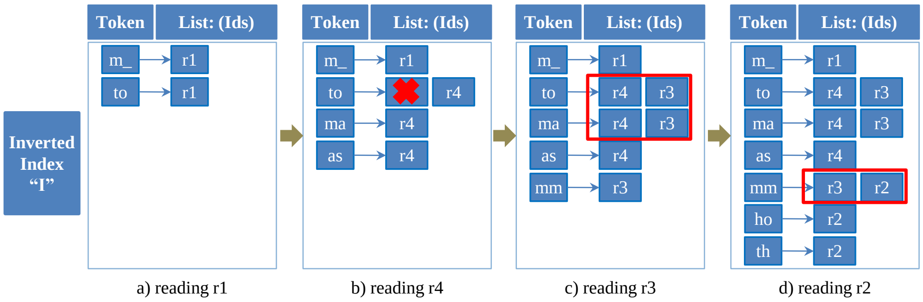


Figure 3: Index usage for PPJoin

prefix tokens to the index, all index entries found for $r1$ can be safely deleted as shown in Figure 3b. When processing the further records r_3 and r_2 the inverted index reveals the prefix overlaps between r_4 and r_3 and between r_3 and r_2 (shown in red squares in Figure 3c and 3d respectively) so that the respective records need to be compared with each other.

PPJoin makes additional use of the prefix tokens by a so-called *position filter*. To approximate the maximal possible overlap between two records, it considers the number of common prefix tokens as well as the position of the last common prefix token. With this information one can derive the maximal possible overlap by assuming that all non-prefix tokens overlap to the maximal extent possible. For example, records r_3 and r_2 in Figure 3d, share only one prefix token, mm, at positions $pos(mm)_{r3} = 1$ and $pos(mm)_{r2} = 3$. This last common token separates each record into a left part lp representing the tokens already seen and a right part rp of unseen tokens. Furthermore the length of rp equals the length of the record minus the position of the last common token, e.g. $rp(r_3) = |r_3| - pos(mm) = 7 - 1 = 6$ and $rp(r_2) = 8 - 3 = 5$. The right parts can thus overlap in at most 5 tokens; together with the prefix overlap in the left parts $(= 1)$ the total overlap between r_2 and r_3 is at most $5 + 1 = 6$.

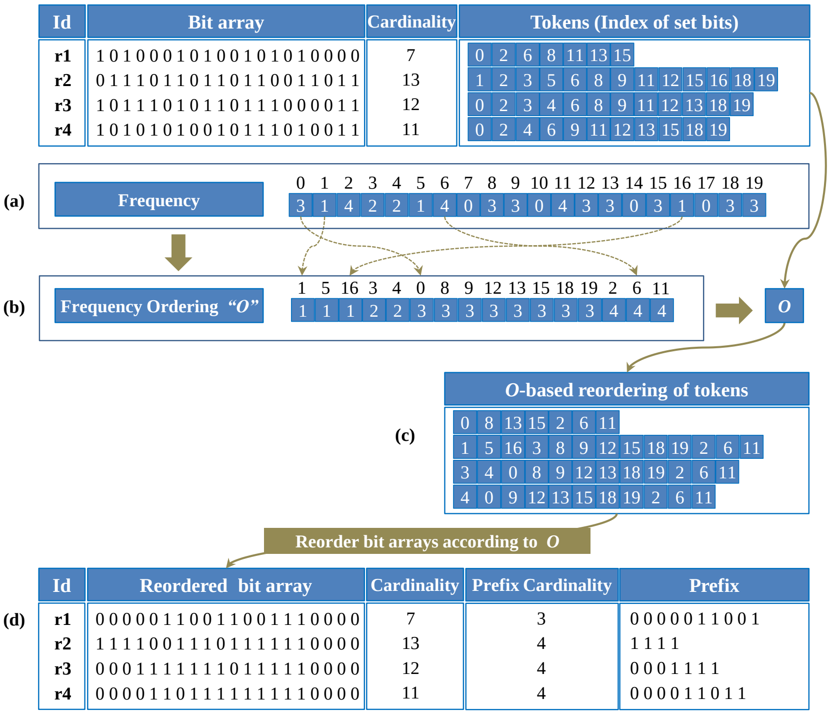
In general, the maximal overlap between two records x and y can be determined by

$$MaxOverlap(x, y) = |lp(x) \cap lp(y)| + min(|rp(x)|, |rp(y)|)$$

The record pair can be filtered out from the similarity comparison if this maximal overlap is smaller than the minimally needed overlap α given in Eq. 5. This is the case for our example because $MaxOverlap(r_3, r_2) = 6 < \alpha = 7$.

4.3 PPJoin for Encrypted Data (P4Join)

For the PPRL version of PPJoin, called P4Join (Privacy-Preserving Prefix Position Join), all input records are encrypted as same-sized bit arrays as explained in Section 2.2. Many parts of PPJoin can then be adapted rather easily by considering the set bit positions (indexes) as the new ”tokens”. The length of a record thus corresponds to the number of set



(a)
Frequency
(b)
Frequency Ordering "*O*"
O
(c)
O-based reordering of tokens
Reorder bit arrays according to *O*
(d)

Id	Bit array	Cardinality	Tokens (Index of set bits)
r1	1 0 1 0 0 0 1 0 1 0 0 1 0 1 0 1 0 0 0 0	7	0 2 6 8 11 13 15
r2	0 1 1 1 0 1 1 0 1 1 0 1 1 0 0 1 1 0 1 1	13	1 2 3 5 6 8 9 11 12 15 16 18 19
r3	1 0 1 1 1 0 1 0 1 1 0 1 1 1 0 0 0 0 1 1	12	0 2 3 4 6 8 9 11 12 13 18 19
r4	1 0 1 0 1 0 1 0 0 1 0 1 1 1 0 1 0 0 1 1	11	0 2 4 6 9 11 12 13 15 18 19

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Frequency	3	1	4	2	2	1	4	0	3	3	0	4	3	3	0	3	1	0	3	3

	1	5	16	3	4	0	8	9	12	13	15	18	19	2	6	11
Frequency Ordering "*O*"	1	1	1	2	2	3	3	3	3	3	3	3	3	4	4	4

O-based reordering of tokens
0 8 13 15 2 6 11
1 5 16 3 8 9 12 15 18 19 2 6 11
3 4 0 8 9 12 13 18 19 2 6 11
4 0 9 12 13 15 18 19 2 6 11

Id	Reordered bit array	Cardinality	Prefix Cardinality	Prefix
r1	0 0 0 0 0 1 1 0 0 1 1 0 0 1 1 1 0 0 0 0	7	3	0 0 0 0 0 1 1 0 0 1
r2	1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 1 0 0 0 0	13	4	1 1 1 1
r3	0 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 0 0	12	4	0 0 0 1 1 1 1
r4	0 0 0 0 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 0	11	4	0 0 0 0 1 1 0 1 1

Figure 4: P4Join preprocessing. The top part shows the computation of the frequency ordering O and the bottom part the reordering of the bit arrays and the computation of prefixes for $t = 0.8$.

bits in its bit array which we call its *cardinality* to avoid confusion. We first explain the adapted preprocessing phase and then the use of the filters in the main processing phase. As we will see, P4Join can be realized without building an inverted index. We will also devise an improved position filter.

(a) Preprocessing: As before, preprocessing works in two passes. In the first pass we determine for each index position its frequency, i.e. we count for how many records it is set. The index positions are then ordered in ascending frequency resulting in an ordering O. Figure 4a and 4b show an example of the position frequencies and the generation of O for four records that are encrypted by bit arrays of size 20. Index positions with frequency 0 (positions 7, 10, etc.) are unset in all bit arrays and can simply be ignored in the further processing. In the second pass, the bits per bit array are reordered according to O as shown in Figure 4c. The prefix length is again calculated according to Eq. 6. Again, the length refers to the number of set positions (cardinality) and not to the size of the prefix. For example in Figure 4d the prefix of record r_1 contains *all* starting bits of the reordered r_1 bit array until we reach the prefix's cardinality 3. Hence, prefixes of the same length may have different sizes.

(b) Comparison of bit arrays: In this second step we process records differently than for PPJoin. This is because we observed in preliminary experiments that the overhead to

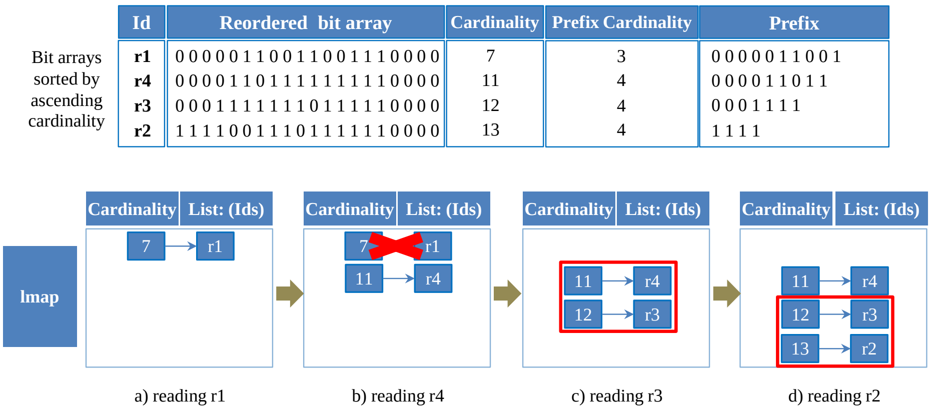
Algorithm 1: P4Join (without preprocessing)

```
Input  : List of records (bit array) $R$ sorted by ascending cardinality;
         Similarity threshold $t$;
Output: $\{(x, y) \in R \times R \mid Sim(x, y) \geq t\}$

1  result ← [];
2  lmap ← new Map(length: List<record>);
3  foreach $x \in R$ do
4      foreach $e \in lmap$ do
5          if $e.length < |x| \cdot t$ then                                   // Length filter
6              lmap.remove($e$);
7          else
8              foreach record $y \in e.List$ do
9                  if $pref(x) \wedge pref(y) \neq 0$ then                    // Prefix filter
10                     if not Positional_Filter ($x, y$) then
11                         if ($Sim_{Tanimoto}(x, y) \geq t$) then
12                             result.add($x, y$);
13     lmap.add($x$);
14 return result;

15 Boolean Positional_Filter($x, y$)
16     $prefOverlap \leftarrow |pref(x) \wedge pref(y)|$;
17     $p_1$ ← position of the last set bit in $pref(x)$;
18     $p_2$ ← position of the last set bit in $pref(y)$;
19     $diff_1 \leftarrow 0$;
20     $diff_2 \leftarrow 0$;
21     if $p_1 > p_2$ then
22         $diff_1$ ← number of set bits in $pref(x)$ having position $> p_2$;
23     else
24         $diff_2$ ← number of set bits in $pref(y)$ having position $> p_1$;
25     $restOverlap \leftarrow min[(|x| - |pref(x)| + diff_1), (|y| - pref(y) + diff_2))]$;
26     $MaxOverlap \leftarrow prefOverlap + restOvelap$;
27     return $MaxOverlap < \lceil (|x| + |y|) \cdot \frac{t}{(1+t)} \rceil$;          // MaxOverlap < MinOverlap?
```

maintain and use an inverted index is too high compared to the achievable savings (which are lowered by the fact that the actual similarity computation is much faster for bit arrays than for general sets of tokens). We similarly observed that using the original position filter of PPJoin results in a performance degradation so that we devise a new position filter.

Algorithm 1 shows the pseudo-code of the main phase of P4Join using as input the output of the preprocessing step as well as the similarity threshold t. It uses a data structure $lmap$ (Line 2) that lists for each record length (cardinality) the records that are still relevant for comparisons based on the length filter. While the shown algorithm uses all three supported filters (length, prefix, position) we can easily deselect some of these filters to evaluate their relative performance impact (this will be used in the evaluation). We now discuss the use and implementation for each of the filters.

Length Filter: According to the shown algorithm, all records are read sequentially in the given order with ascending cardinality (see top of Figure 5). When a record x with cardinality c is read, it is added to $lmap$ either together with a new entry for cardinality c or by appending it to an existing record list for c (Line 13). The first record is immedi-



	Id	Reordered bit array	Cardinality	Prefix Cardinality	Prefix
Bit arrays sorted by ascending cardinality	**r1**	0 0 0 0 0 1 1 0 0 1 1 0 0 1 1 1 0 0 0 0	7	3	0 0 0 0 0 1 1 0 0 1
	r4	0 0 0 0 1 1 0 1 1 1 1 1 1 1 1 1 0 0 0 0	11	4	0 0 0 0 1 1 0 1 1
	r3	0 0 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 0 0 0	12	4	0 0 0 1 1 1 1
	r2	1 1 1 1 0 0 1 1 1 0 1 1 1 1 1 1 0 0 0 0	13	4	1 1 1 1

a) reading r1

b) reading r4

c) reading r3

d) reading r2

Figure 5: P4Join processing for four sample records

ately added while for a non-empty $lmap$ the addition follows after applying the length filter (and possibly further filters) and performing the necessary similarity checks against already read records. For example, reading the first record r_1 results in the state of $lmap$ shown in (Figure 5a) with r_1 listed for cardinality 7.

The $lmap$ data structure makes it easy and efficient to identify candidates that no longer need to be considered according to the length filter. This is achieved by the second $foreach$ loop where we check for each length in $lmap$ whether the length filter applies in comparison with the current record x. Each record with such a length can safely be excluded from all further comparisons since all further records have the same or larger cardinality than x (Lines 4 to 6). When processing the second record r_4 we therefore determine that cardinality 7 does not satisfy the length filter for threshold $t = 0.8$ ($7 < \lceil 0.8 \cdot 11 \rceil$). Hence this entry together with its record $r1$ can be safely deleted from $lmap$ (Figure 5b). After eliminating the entry, record r_4 is added to a new $lmap$ element for cardinality 11. The further records, r_3 and r_2, do not allow a similar reduction so that all records need to be compared with each other if we would only apply the length filter.

Prefix Filter: The prefix filter avoids the comparison for two records if their prefixes do not share at least one set bit position. In PPJoin the prefix overlap is checked with an inverted index which is relatively expensive to maintain and use for bit arrays. This is because the bit arrays make it very cheap to directly determine the overlap by simply computing the `AND` operation between the two prefixes. Only if the resulting bit array has at least a single bit set we need to further consider the pair of records (Line 9); otherwise we can avoid the similarity computation. Figures 5c and 5d show in red squares the two pairs that pass the prefix filter. For example the records r_4 and r_3 pass the length filter ($11 > \lceil 0.8 \cdot 12 \rceil$) and the prefix filter (`000011011 AND 0001111 = 000011000`), so that they must be compared with each other (unless the position filter avoids this which will be the case, see below). By contrast, records r_4 and r_2 have no common bit position in their prefixes so that their similarity comparison is saved.

Position Filter: Applying the PPJoin position filter to bit arrays turned out to be relatively expensive (partly due to the need to determine and use the positions of last common prefix tokens) but was also limited by an overly imprecise estimate for the maximal overlap. We can determine a more accurate maximum overlap between two records by considering the *last prefix tokens* instead of the *last common prefix token*. We explain the idea for records r_4 and r_3 having cardinalities 11 and 12 and the prefixes of set positions $pref(r_4) = [4, 5, 7, 8]$ and $pref(r_3) = [3, 4, 5, 6]$ as shown in the top of Figure 5. The overlap between the two prefixes equals 2 and the last common prefix token, 5, has the positions $pos(5)r_4 = 2$ and $pos(5)r_3 = 3$. Using the PPJoin position filter, we thus obtain $MaxOverlap(r_4, r_3) = 2 + min(11 - 2, 12 - 3) = 11$. This overlap is not smaller than the minimum required overlap $\alpha = \lceil(11 + 12) \cdot (0.8/1.8)\rceil = 11$ (Eq. 5) so that the similarity computation between r_4 and r_3 cannot be saved.

However, we observe that token 6 in $pref(r_3)$ cannot contribute to $MaxOverlap$ because it refers to a bit position smaller than some tokens in $pref(r_4)$, namely 7 and 8. In general, given two prefixes $pref(x)$ and $pref(y)$ having as last tokens a and b respectively, if $a < b$ then the only prefix tokens that can contribute to $MaxOverlap$ (beside the common tokens) are those tokens from $pref(y)$ greater than a. For r_3, the maximal number of possibly overlapping tokens is thus the number of common prefix tokens (2) plus the number of non-prefix tokens (8), i.e. 10 in total. For r_4, the maximal possible overlap is the number of common prefix tokens (2), plus the number of non-prefix tokens (7) plus the number of tokens in $pref(r_4)$ greater than the last token 6 in $pref(r_3)$ $(= 2)$, i.e. 11 in total. The maximal overlap is the minimum of 10 and 11 which is 10 and below the minimal overlap $\alpha = 11$. Hence, we can filter out r_4 and r_3 from the similarity comparison.

This new position filtering can thus prune more dissimilar pairs of records compared to the original PPJoin position filter. Its implementation as a function is shown in the bottom part of Algorithm 1. It determines the last tokens in the two prefixes. The smaller of the two determines how many tokens of the other prefix may contribute to $MaxOverlap$. The function returns the boolean value true if the maximal overlap is smaller than the minimally needed overlap α. Hence in the main algorithm the similarity comparison is only performed if the position filter does not apply (Line 10).

5 Matching Encrypted Data with GPUs

The utilization of Graphical Processing Units (GPUs) to speed-up similarity computations is a comparatively new approach [FPS$^+$13]. Modern GPUs provide thousands of cores that allow for a massively-parallel application of the same instruction set to disjoint data partitions. The availability of frameworks like CUDA[2] and OpenCL[3] simplify the utilization of GPUs to parallelize general purpose algorithms. In this work, we rely on OpenCL which, in contrast to CUDA, is supported by different hardware vendors. An OpenCL-Program, which is also referred to as a *Kernel*, is written in a dialect of the *C* programming language. The number of kernels that can be applied to disjoint data partitions simultaneously de-

[2] https://developer.nvidia.com/category/zone/cuda-zone

[3] https://www.khronos.org/opencl/

pends on the number of cores provided by the GPU and the memory requirements of the Kernel instances.

Compared to developing "CPU programs", GPUs and their programming languages like OpenCL have several limitations. Before and after the computation, input and output data must be transferred between the main memory of the host system and the memory of the GPU. Furthermore, on GPUs dynamic memory allocation is not possible at runtime, i.e., the resources required by an algorithm must be known and allocated a priori. Usually the memory capacity of GPUs is smaller than the available main memory. This requires to divide the overall workload into smaller tasks which are executed by the GPU in multiple rounds. Thereby, the scheduling of the individual tasks should minimize the data volume to be transferred between host system and GPU. Furthermore only basic data types (e.g., int, long, float) and fixed-length data structures (e.g., arrays) can be used. Despite such limitations, the utilization of GPUs is a promising approach to speed up record linkage on encrypted datasets. All records are represented as bit arrays of equal length, which can be expressed by arrays of type long (64 bits). Furthermore, the similarity computation can be broken down into simple bit operations which can be easily processed by GPUs.

We describe the GPU version of P4Join for the general case of two input datasets R and S; the special case with only one dataset R is easily supported by comparing R with itself. The preprocessing is performed on the host system while the filtering and similarity checks are performed on the GPUs. Only the length and prefix filters are applied as a GPU implementation of the position filter becomes already too complex compared to the similarity computation itself. We also support a hybrid scheme utilizing both CPUs and GPUs for similarity computation.

5.1 Preprocessing

In a preprocessing step, the records of the input datasets R and S are sorted by their cardinality, i.e., the number of bits set to one. Additionally, the bits of all records are ordered by the document frequency in ascending order. Furthermore, each record is annotated with its cardinality and its prefix fingerprint (see Section 4.3). In general, the input datasets as well as the resulting correspondences exceed the available memory of the GPU. Thus, R and S are range-partitioned into fixed-sized partitions. Similar to [HKGR13], pairs (R_i, S_j) are then iteratively shipped to the GPU for similarity computation. Before a partition pair is shipped to the GPU, it is checked whether the two partitions contain at least one record pair which passes the length filter (see Eq. 2). To this end, the cardinality values of the first and the last record of the two (sorted) partitions are compared. If there is no such record, the pair is skipped. Otherwise it is transferred to the GPU.

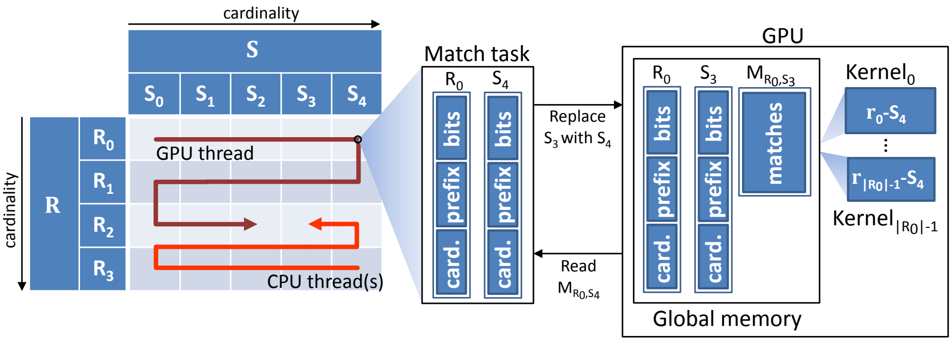


Figure 6: Execution scheme for hybrid GPU/CPU-based record linkage minimizing the data transfer between the host program and the GPU.

5.2 Massively-parallel Record Linkage

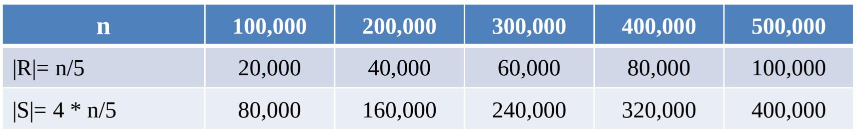
The GPU executes a kernel instance for each record $r \in R_i$. A kernel instance is responsible for comparing r with each $s \in S_j$. Thereby, the kernel iterates over all $s \in S_j$ (which are sorted by cardinality) and consecutively checks whether the length filtering constraint is fulfilled and whether the evaluation of subsequent $s' \in S_j$ can be skipped. For each remaining candidate record, it is checked whether the prefix fingerprints do overlap. If this is the case, their similarity is computed according to Eq. 3.

The indexes of all $s \in S_j$ with $Sim_{Tanimoto}(r, s) \geq t$ are written to a kernel instance-specific range of an output array of type int which can be accessed by all kernel instances. The partial results are later unified by the host program.

To minimize the data volume to be transferred between the main memory and the GPU we also support a *hybrid CPU/GPU execution scheme* similar to [HKGR13]. We utilize a task queue that supports the parallel matching of different partition pairs on both the GPU as well as on the CPU. A dedicated thread takes tasks from this queue and submits them to the GPU. In addition to this GPU thread, several CPU threads can access the job queue from the opposite end to independently perform matching on the CPU. We select jobs and ship partitions using the scheme displayed in Figure 6. This scheme ensures that after completion of a GPU job only one new partition need to be transferred to it. The other partition remains in the GPU's memory and is reused for the next job.

6 Evaluation

We evaluate the proposed P4Join approach and its filters for several datasets of different sizes and compare the resulting execution times with the use of the Multibit tree algorithm and a naive nested loop implementation. We also evaluate the GPU implementation of P4Join. Before presenting the evaluation results we introduce the experimental setup



n	100,000	200,000	300,000	400,000	500,000
|R|= n/5	20,000	40,000	60,000	80,000	100,000
|S|= 4 * n/5	80,000	160,000	240,000	320,000	400,000

Figure 7: Size of the datasets (number of records) used in the evaluation

including the used datasets.

Note that for a given dataset all the algorithms produce the same match result, i.e. they find *all* the pairs having a similarity greater or equal to a predefined threshold. Hence, the quality of the match result does not depend on the considered algorithms but on the input datasets and the encryption method. The quality of the used encryption method has already been evaluated in [SBR09, BRS13].

6.1 Experimental setup

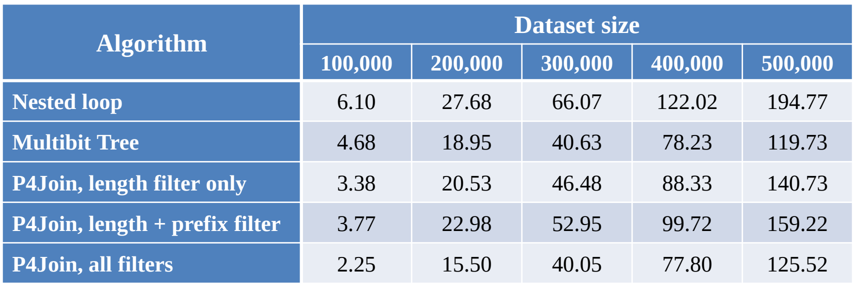
In our experimental setup we largely follow the settings used in previous evaluations on privacy-preserving record linkage such as [BRS13]. We utilize the data generator from [Chr05] to generate five differently sized datasets of person records. Each dataset consists of n records that are assigned to two subsets R and S of size $1/5 \cdot n$ and $4/5 \cdot n$ respectively such that R contains duplicate records of records in S (see Figure 7). The records are tokenized into bigrams which are mapped to bit vectors of length $l = 1000$ using $k = 20$ hash functions. For each configuration, we apply a Jaccard similarity threshold of $t = 0.8$. For the Multibit tree experiments, we use a Java implementation of the approach[4] with the clustering strategy *split in half* and the minimal node size of 6 (see Section 3).

All configurations of the first experiment are conducted single-threaded on a desktop machine with a 4-core $2.67 GHz$ CPU and 4GB of main memory. For the second experiment, we utilize two low-profile GPUs, namely a Nvidia GeForce GT 610 (48 CUDA cores@810MHz, 1GB memory, 35 €) and a Nvidia GeForce GT 540M (96 CUDA cores @672MHz, 1GB memory). For the GPU-based comparisons, the datasets R and S are split into partitions with a maximum size of $2,000$ records.

6.2 Comparing P4Join with Multibit Trees and NestedLoop

Figure 8 shows the achieved execution times for record linkage on the five datasets for a naive nested loop implementation (evaluating the Cartesian product), the previously proposed Multibit tree as well as the P4Join and its filters. As expected the execution times increase substantially (almost quadratically) with the dataset size. The Multibit Tree approach consistently outperforms NestedLoop albeit only to a relatively small degree of less

[4] http://www.birc.au.dk/~tgk/TanimotoQuery/



Algorithm	Dataset size				
	100,000	200,000	300,000	400,000	500,000
Nested loop	6.10	27.68	66.07	122.02	194.77
Multibit Tree	4.68	18.95	40.63	78.23	119.73
P4Join, length filter only	3.38	20.53	46.48	88.33	140.73
P4Join, length + prefix filter	3.77	22.98	52.95	99.72	159.22
P4Join, all filters	2.25	15.50	40.05	77.80	125.52

Figure 8: Runtime in minutes for P4Join with different filters compared with the nested loop and multibit tree approaches

than a factor of 2. This shows already that the applied filtering and reduction of the search space are not so significant if the similarity computations are cheap which is the case for computing the Taminoto similarity on bit arrays. This also limits the effectiveness of the P4Join implementation for which we separate the results based on the use of the different filters.

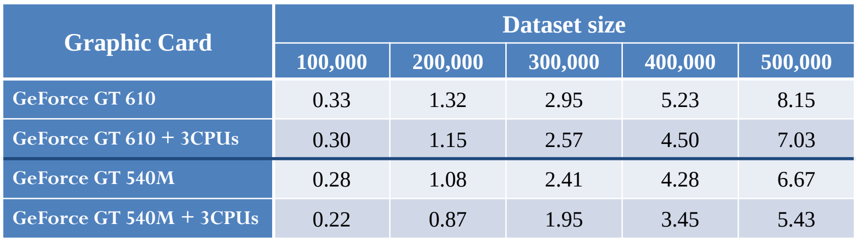
The best results are achieved by applying all filters including the new position filter. In this case we achieve similarly good results than for the Multibit Tree. Most of the savings in execution time compared to NestedLoop are already achieved by the length filter which can exclude many comparisons with a simple length check. By contrast the prefix and position filters incur a check per record pair which may be unsuccessful and at best saves a single comparison which is not much more expensive for bit arrays than the filter check. This was especially a limitation for the prefix filter that did not pay off in combination with the length filter. To explain this somewhat surprising result we checked more closely the record and prefix characteristics. We observed that for our settings the generated bit arrays have an average cardinality of 300 and an average prefix length of 60. Such large prefixes lead to a high probability of non-empty intersections so that relatively few comparisons could be saved while the overhead of the prefix check occurs for every record pair passing the length filter.

P4Join preprocessing was generally very fast compared the whole execution time. It varied from $6s$ for the smallest dataset ($100,000$ records) to $24s$ for the largest ($500,000$ records),i.e. less than 0.3% of complete execution time.

6.3 GPU-based results

The relatively slow improvements in execution time show the need for more optimizations such as the use of parallel processing. As outlined in Section 5, the simplicity of P4Join made it possible to develop a parallel implementation for GPUs that we evaluate now.

Figure 9 shows the achieved execution times for the five datasets on the two considered graphic cards. We also present results for the hybrid case when we use three CPU threads



Graphic Card	Dataset size				
	100,000	200,000	300,000	400,000	500,000
GeForce GT 610	0.33	1.32	2.95	5.23	8.15
GeForce GT 610 + 3CPUs	0.30	1.15	2.57	4.50	7.03
GeForce GT 540M	0.28	1.08	2.41	4.28	6.67
GeForce GT 540M + 3CPUs	0.22	0.87	1.95	3.45	5.43

Figure 9: Runtime in minutes of two different graphic cards, also in hybrid mode with 3 CPU threads.

for similarity computation in addition to the GPUs. We observe that both GPUs allow huge improvements in efficiency by reducing execution times by more than a factor 10 and a factor 15-20 for the largest dataset compared to the sequential CPU execution of P4Join. The hybrid approach allows a further improvement by 10-20% by utilizing the CPU threads for additional parallelism and saved data transfers to the GPU. For the largest dataset we could thus improve the execution time to only about 5 minutes compared to 125 minutes for the sequential P4Join execution and 195 minutes for NestedLoop.

The results show the high potential of executing P4Join in parallel and that both GPU and CPU parallelism can be effectively combined.

7 Conclusions and Future Work

We showed how the PPJoin approach for similarity joins can be adapted to privacy-preserving record linkage where sensitive records are encrypted by bit arrays. The new approach called P4Join supports a length, prefix and an optimized position filter. We also showed how P4Join can be executed in parallel on GPUs. Our evaluation revealed that the efficient similarity computation for bit arrays reduces the optimization potential for filter techniques such as the ones in PPJoin and other similarity join implementations. Still the proposed P4Join approach and especially the length filter and the new position filter proved to be effective in reducing the execution time. The biggest performance gains are achieved by the parallel computation on GPUs with a speedup of up to 20 even for low-profile graphic cards.

We see a strong need for further research on improving the efficiency of PPRL schemes to support their scalability to very large datasets. First, the almost quadratic increase of execution times w.r.t input size needs to be improved, e.g. by the use of tailored blocking mechanisms. Furthermore, parallel processing should be employed more comprehensively, in particular on clusters of processing nodes in addition to the node-specific parallel record linkage using GPUs and several CPU cores.

References

[AGK06] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient Exact Set-Similarity Joins. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 918–929, 2006.

[ALM05] Ali Al-Lawati, Dongwon Lee, and Patrick McDaniel. Blocking-Aware Private Record Linkage. In *Proceedings of 2nd International Workshop on Information Quality in Information Systems*, pages 59–68, 2005.

[BMS07] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling Up All Pairs Similarity Search. In *Proceedings of the 16th International Conference on World Wide Web*, pages 131–140, 2007.

[BRS13] Tobias Bachteler, Jörg Reiher, and Rainer Schnell. Similarity Filtering with Multibit Trees for Record Linkage. Technical Report WP-GRLC-2013-01, German Record Linkage Center, 2013.

[CGK06] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

[Chr05] Peter Christen. Probabilistic Data Generation for Deduplication and Data Linkage. In *Proceedings of 6th International Conference on Intelligent Data Engineering and Automated Learning*, pages 109–116, 2005.

[Chr12a] Peter Christen. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Transactions on Knowledge and Data Engineering*, 24(9):1537–1555, 2012.

[Chr12b] Peter Christen. *Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Data-centric systems and applications. Springer, 2012.

[Coh00] William W. Cohen. Data Integration Using Similarity Joins and a Word-Based Information Representation Language. *ACM Transactions on Information Systems*, 18(3):288–321, 2000.

[DN11] Uwe Draisbach and Felix Naumann. A Generalization of Blocking and Windowing Algorithms for Duplicate Detection. In *Proceedings of 5th International Conference on Data and Knowledge Engineering*, pages 18–24, 2011.

[EIV07] Ahmed K. Elmagarmid, Panagiotis G. Ipeirotis, and Vassilios S. Verykios. Duplicate Record Detection: A Survey. *IEEE Transactions on Knowledge and Data Engineering*, 19(1):1–16, 2007.

[FPS$^+$13] Benedikt Forchhammer, Thorsten Papenbrock, Thomas Stening, Sven Viehmeier, Uwe Draisbach, and Felix Naumann. Duplicate Detection on GPUs. In *Proceedings of the 15th Conference on Database Systems for Business, Technology, and Web*, pages 165–184, 2013.

[HCKS08] Marios Hadjieleftheriou, Amit Chandel, Nick Koudas, and Divesh Srivastava. Fast Indexes and Algorithms for Set Similarity Selection Queries. In *Proceedings of the 24th International Conference on Data Engineering*, pages 267–276, 2008.

[HKGR13] Michael Hartung, Lars Kolb, Anika Groß, and Erhard Rahm. Optimizing Similarity Computations for Ontology Matching - Experiences from GOMMA. In *Proceedings of the 9th International Conference on Data Integration in the Life Sciences*, pages 81–89, 2013.

[JLFL14] Yu Jiang, Guoliang Li, Jianhua Feng, and Wen-Syan Li. String Similarity Joins: An Experimental Evaluation. *Proceedings of the VLDB Endowment*, 7(8):625–636, 2014.

[KM06] Adam Kirsch and Michael Mitzenmacher. Less Hashing, Same Performance: Building a Better Bloom Filter. In *Algorithms - ESA 2006, 14th Annual European Symposium*, pages 456–467, 2006.

[KNP10] Thomas Greve Kristensen, Jesper Nielsen, and Christian N. S. Pedersen. A tree-based method for the rapid screening of chemical fingerprints. *Algorithms for Molecular Biology*, 5:9, 2010.

[KR10] Hanna Köpcke and Erhard Rahm. Frameworks for entity matching: A comparison. *Data & Knowledge Engineering*, 69(2):197–210, 2010.

[KTR10] Hanna Köpcke, Andreas Thor, and Erhard Rahm. Evaluation of entity resolution approaches on real-world match problems. *Proceedings of the VLDB Endowment*, 3(1), 2010.

[KTR12] Lars Kolb, Andreas Thor, and Erhard Rahm. Dedoop: Efficient Deduplication with Hadoop. *Proceedings of the VLDB Endowment*, 5(12):1878–1881, 2012.

[MF12] Ahmed Metwally and Christos Faloutsos. V-SMART-Join: A Scalable MapReduce Framework for All-Pair Similarity Joins of Multisets and Vectors. *Proceedings of the VLDB Endowment*, 5(8):704–715, 2012.

[NKH$^+$13] Axel-Cyrille Ngonga Ngomo, Lars Kolb, Norman Heino, Michael Hartung, Sören Auer, and Erhard Rahm. When to Reach for the Cloud: Using Parallel Hardware for Link Discovery. In *Proceedings of the 10th International Extended Semantic Web Conference*, pages 275–289, 2013.

[RLW$^+$13] Chuitian Rong, Wei Lu, Xiaoli Wang, Xiaoyong Du, Yueguo Chen, and Anthony K. H. Tung. Efficient and Scalable Processing of String Similarity Join. *IEEE Transactions on Knowledge and Data Engineering*, 25(10):2217–2230, 2013.

[SBR09] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. Privacy-preserving record linkage using Bloom filters. *BMC Med. Inf. & Decision Making*, 9:41, 2009.

[SBR11] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. A Novel Error-Tolerant Anonymous Linking Code. Technical Report WP-GRLC-2011-02, German Record Linkage Center, Duisburg, 2011.

[Sch14] Rainer Schnell. An efficient privacy-preserving record linkage technique for administrative data and censuses. *Statistical Journal of the IAOS*, 30(3):263–270, 2014.

[Sch15] Rainer Schnell. Privacy Preserving Record Linkage. In Katie Harron, Harvey Goldstein, and Chris Dibben, editors, *Methodological Developments in Data Linkage*. Wiley, 2015. To appear.

[SHC14] Akash Das Sarma, Yeye He, and Surajit Chaudhuri. ClusterJoin: A Similarity Joins Framework using Map-Reduce. *Proceedings of the VLDB Endowment*, 7(12):1059–1070, 2014.

[VCL10] Rares Vernica, Michael J. Carey, and Chen Li. Efficient parallel set-similarity joins using MapReduce. In *Proceedings of the International Conference on Management of Data*, pages 495–506, 2010.

[VCV13] Dinusha Vatsalan, Peter Christen, and Vassilios S. Verykios. A taxonomy of privacy-preserving record linkage techniques. *Information Systems*, 38(6):946–969, 2013.

[WLF10] Jiannan Wang, Guoliang Li, and Jianhua Feng. Trie-Join: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints. *Proceedings of the VLDB Endowment*, 3(1):1219–1230, 2010.

[XWL08] Chuan Xiao, Wei Wang, and Xuemin Lin. Ed-Join: An Efficient Algorithm for Similarity Joins With Edit Distance Constraints. *Proceedings of the VLDB Endowment*, 1(1):933–944, 2008.

[XWLY08] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient Similarity Joins for Near Duplicate Detection. In *Proceedings of the 17th International Conference on World Wide Web*, pages 131–140, 2008.