Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Privacy-Preserving Record Linkage with Apache Spark

Author: Onno Valkering

1st supervisor: Adam Belloum
2nd reader: Rob van Nieuwpoort

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 17, 2018

Abstract

Privacy-Preserving Record Linkage
with Apache Spark

by Onno VALKERING

Privacy considerations obligate careful and secure processing of personal data. This is especially true when personal data is linked against databases from other organizations. During such endeavours, privacy-preserving record linkage (PPRL) can be utilized to prevent needless exposure of sensitive information to other organizations. With the increase of personal data that is being gathered and analyzed, scalable PPRL capable of handling massive databases is much desired. In this work we evaluate the Hadoop-ecosystem, in particular Apache Spark, as an option to scale PPRL. Not only is it valuable to have a scalable PPRL implementation, but one based on the Hadoop-ecosystem and Spark would also be commonly deployable and could take advantage of further development of the ecosystem. The results show that a PPRL solution based on Apache Spark outperforms alternatives when it comes to handling multiple millions of records; can scale to dozens of nodes on a cluster; and is on-par with regular record linkage implementations in terms of achieved results.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Privacy-preserving record linkage

The insight that organizations can gain from analyzing data may lead to a competitive advantage and/or improved decision making. The prospect of this valuable insight can be an incentive for organizations to start or extend gathering and analyzing data on a Big Data scale. This scale is characterized by massive volumes of varied data that are gathered and/or analyzed at a high velocity [16].

It is possible that, intentionally or unintentionally, personal data is also captured when operating on such a Big Data scale. When this is the case, privacy considerations obligate careful and secure processing of personal data. This is especially true when personal data is linked against databases from other organizations, for example with record linkage[1] applications. The goal of record linkage is to identify one and the same entities across multiple databases [10, pp. 3-4]. When databases from different organizations are the subject of record linkage, measures can be taken to prevent unnecessary exposure of sensitive information to any of the other participating organizations. When records are found that represent, with a sufficiently high confidence, the identical entity, only the relevant database owners will be notified. Still without exposing any sensitive information, until organizations, in agreement, exchange full details. This extension of record linkage is known as privacy-preserving record linkage (PPRL).

There are, apart from the various techniques that can be used, two major ways to perform PPRL: with a *two-party protocol* or with a *three-party protocol* [10, pp. 193-194]. When using a two-party protocol, the involved organizations directly and solely communicate with each other. In case of a three-party protocol, a trusted third-party, called the linkage unit, is involved to perform the actual linkage. Choosing between the two protocols involves a trade-off between security and practicality. Two-party protocols are considered to be more secure, as it is not possible for organizations to collude with the linkage unit, but are computationally more intensive and complex

[1] In other domains record linkage is also referred to as *data matching* or *entity resolution*.

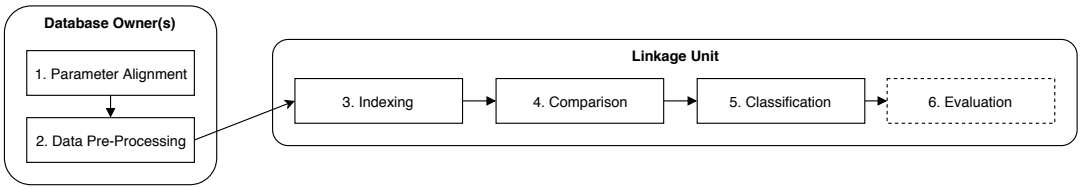

FIGURE 1.1: Steps of the PPRL process.

than three-party protocols [10, pp. 193-194]. Whether or not to account for the possibility of colluding organizations depends on the adversary model applicable. In this work the most commonly used *honest-but-curious* (HBC) model [41] is assumed. In essence, this model states that while organizations are interested in sensitive information from other organizations, they won't break the rules to gain it. Therefore colluding organizations aren't included in the HBC adversary model and we can limit this work to the more practicable three-party protocol PPRL.

1.1.1 The PPRL process

A typical record linkage process [10, pp. 23-35] consists of six steps: *data pre-processing*, *indexing*, *comparison*, *classification*, *evaluation* and *clerical review*. Two alterations have been made, on account of the privacy-preserving aspects of PPRL, to this process (figure 1.1). Namely, we discarded the clerical review, as this might undermine privacy, and consider parameter alignment as a mandatory first step in the process.

1. Parameter alignment — With PPRL, records have to be encoded in such a way that the original values of fields can't be recovered by other participants. On the other hand, it must remain possible to perform meaningful comparisons of the encoded records. These comparisons are made to determine if two records are matching, i.e. are the identical entity. To achieve this, participants must use the same parameters for certain tasks. In this step the parameters are determined and shared.

2. Data pre-processing — In this step each organization encodes[2] its records. Records used for PPRL typically consist of multiple fields that are quasi-identifiers (QIDs). QIDs are attributes that, when combined, may trace back to a specific entity [38]. Before encoding, it is useful to first clean and format these QIDs in a uniform way. After encoding, the records are send to the linkage unit. Since access to plain records is required, this step needs to be performed within the organization's boundaries.

3. Indexing — Next, the linkage unit will index the encoded records. Typically, an indexing technique is used that can efficiently group potentially matching records. By only comparing these records the total amount of comparisons that have to be performed can be reduced. This is a crucial step in terms of scalability, as the amount of comparisons of a naive pairwise approach could be n^p in the worst case (where p is the number of databases, each containing n records).

[2] To encode with the aim to obfuscate is also known as *data masking*.

4. Comparison — Through comparison of two records the degree of (dis)similarity can be determined. Various similarity or distance functions can be used for this purpose, the most appropriate one depends on the encoding scheme used. Commonly used functions include *Hamming distance*, *Dice coefficient* and *Jaccard index*.

5. Classification — Once it is known how (dis)similar two records are, a classification can be made: match or non-match. For regular record linkage various decision models exist, including machine learning models [18]. For PPRL, other options than a threshold function have not been thoroughly explored [43].

6. Evaluation — Metrics such as accuracy, precision and recall can be used for evaluation. Measuring privacy itself is more difficult [42], and is out of scope for this work. Furthermore, in case of private real-world data it might not be possible to perform a complete evaluation, as some metrics require full disclosure to the data. In particular, the amount of false-positives and/or -negatives might not be known.

1.1.2 Applications of PPRL

Although the goal of PPRL is the same as that of regular record linkage, the use cases are often slightly different. As mentioned, the use cases for PPRL typically ask for careful and secure processing of data. Take for example the application of record linkage on a single database, this essentially is *data deduplication*. If the database would be so extensive that the organization's own IT capabilities are too limited, the organization could decide to rent a more powerful infrastructure from a public cloud provider. However, if the database contains personal data, the organization can make use of PPRL to effectively make sure that other parties, in this case the public cloud provider, can't learn anything from the data. In such a *private data deduplication* scenario, the public cloud provider would function as the linkage unit.

PPRL applications that involve multiple parties are typically found in the domains of crime and fraud detection, government services and healthcare [43]. An interesting example, involving multiple parties, is outlined in [20]. This example considers a measure that can be taken during a virus outbreak. Namely, comparing airline passenger lists with hospital records to be able to alert passengers in case it is retrospectively discovered that another passenger aboard of the same flight had been infected with the virus. Such an application requires linkage between personal data originating from multiple organizations. For this part PPRL can be used.

Another example, described in [13], is the identification of terrorist suspects that enter or leave a country. This is already done by the European Union that shares information about inbound travelers with the United States to check if any of them is on a terrorist watchlist [25]. This kind of surveillance and screening can make use

PPRL to prevent privacy intrusions of those that are not on any watchlist. Similar applications are possible for checking against lists of wanted criminals or checking for open debts when engaging in (risky) financial products, without violating privacy.

Some PPRL techniques can also be adapted to perform *privacy-preserving similarity search* (PPSS). In this case the matching criteria are relaxed and a record does no longer can only be matched to a single other record, but can be matched to any number of other record that matches the criteria, i.e. any record that is similar enough. This privacy-preserving approach to similarity search may, for example, be used to find similar patients based on their medical records. Other possible applications of PPSS include analysis of clinical trails and healthcare software that automatically personalizes to specific patients [40].

1.2 PPRL on a Big Data scale

PPRL on a Big Data scale poses several challenges, as listed in [43]:

- attaining high-quality linkage results in noisy and varied Big Data;
- preserving a certain privacy level, even with multiple participants;
- scalability when the size and amount of databases increases.

In recent papers [43, 36] the *Hadoop-ecosystem*, and *Spark* in particular, is suggested as an option for scaling PPRL. The umbrella term Hadoop-ecosystem encompasses Hadoop itself (i.e. *HDFS*, *MapReduce* and *YARN*) [37, 44], as well as the various tools and frameworks that integrate seamlessly with Hadoop (e.g. *HBase*, *Spark* and *Pig*). An important advantage of the Hadoop-ecosystem is its widespread adoption and accessibility. Most major public cloud providers, for example Amazon Web Services (AWS), offer fully managed Hadoop clusters[3]. This allows organizations that don't possess the required hardware and infrastructure required for a Hadoop cluster to still make use of the Hadoop-ecosystem. It is known that Spark can achieve great performance and scale to hundreds of nodes [47]. However no clear picture exists how well it can handle PPRL [43]. The last evaluation, of those available in scientific literature, used the MapReduce framework on a relatively small dataset of 300,000 record and was limited to 2-4 compute nodes [22]. Since then (2013), the Hadoop-ecosystem has developed further and new tools and frameworks have been released.

In this work the fitness of the Hadoop-ecosystem for PPRL is re-evaluated, by presenting and benchmarking scalable PPRL implementations based on Spark. Not only is it valuable to have a scalable PPRL implementation, but one based on the Hadoop-ecosystem and Spark would also be commonly deployable and could take advantage of further development of the ecosystem and community expertise.

[3] https://aws.amazon.com/emr/

1.2.1 Thesis outline

This thesis is structured as follows. In the next chapter (2), related work is discussed. Then, the theoretic foundation of the PPRL encoding scheme used in this work is laid out in chapter 3. In the subsequent chapter (4) a description is provided on how to find matches based on the encoded records. Then, in chapters 5, two alternative Spark implementations are discussed. Further applications of the PPRL implementations are described in 6. In this last two chapters and we discuss future work (7) and state our conclusions (8).

Reproducibility

All source code written and datasets used as part of this work are available on GitHub[4] under the MIT open-source license, or are publicly available elsewhere.

Acknowledgment

The cluster-deployed experiments in this work have been made possible by SURFsara, that granted access to their Hathi Hadoop cluster for this purpose.

[4] https://github.com/onnovalkering/bigpprl

Chapter 2

Related work

2.1 Secure multi-party computation

With secure multi-party-computation (SMC) two or more parties engage, without an intermediary, in a joint effort to solve some arbitrary computation [28]. The input that is required from each party is not revealed to the other parties. However, afterwards every party receives the output of the computation intended to solve. This makes it especially useful for calculations based on personal data originating from different organizations. As a drawback, SMC suffers from significant computational overhead [43] that limits its ability to be used for efficient and large-scale PPRL.

2.2 LSHDB

LSHDB is an open-source[1] database and execution engine, developed using Java, capable of performing similarity search, record linkage and PPRL [21]. It supports batch processing, as well as online querying. To optimize query speed it relies on locality-sensitive hashing for indexing of records and features distributed and parallel capabilities. We identified two shortcomings that impede efficient and large-scale deployment of LSHDB for PPRL. First, the parallel execution only kicks in once a query is submitted and works only in favor of solving that particular query. There is no coordination between threads in case of multiple concurrent queries. Secondly, underlying LSHDB is the NoSQL database LevelDB. This is a on-disk database that may impose a disk reading latency bottleneck when querying large amounts of records.

In this thesis LSHDB is used as a performance baseline to indicate the current state of open-source PPRL solutions. For us to use LSHDB for this purpose, some modification had to be made to the source code. The modified source code, along with a description of the modifications can be found on GitHub[2].

[1] https://github.com/dimkar121/LSHDB
[2] https://github.com/onnovalkering/lshdb-star

Chapter 3

Encoding

3.1 Bloom filter encoding

As mentioned in section 1.1.1 (The PPRL process), records have to be encoded before they are sent to the linkage unit. The Bloom filter data structure [8] has been adopted for this purpose [35]. Because of its effectiveness, good privacy protection and its relatively low computational costs, it has become a widely used encoding scheme for PPRL [10, 14, 20, 22, 36, 40, 43]. Compared to directly using cryptographic hash functions for encoding, Bloom filters encoding has the advantageous feature that a small difference in input doesn't produce a completely different output. Therefore, Bloom filter encoding can also be used for approximate matching of values instead of only exact matching. This makes Bloom filter encoding tolerant for modest data corruption such as typing errors or phonetic variation. An important characteristic, as data corruption often occurs in real-world personal data [39].

This work considers two Bloom filters types: *field-level Bloom filters* (FBF) and *record-level Bloom filters* (RBF). Before describing further details, some notations need to be introduced:

- R is a ordered list of fields (i.e. a record), f_i denotes the i^{th} field;
- W is a ordered list of weights, w_i denotes the weight intended for f_i;
- T is a set of tokens;
- v is a bit vector of length m, with all values initially set to 0;
- H is a set of k hash functions with range $[0, m)$.

3.1.1 Field-level Bloom filters

Records are encoded by first separately encoding all of its fields into FBFs. For an arbitrary textual field f, the encoding procedure is as follows [35]. First a bit vector v_{FBF} of length m_{FBF} is created. Next, the field f is tokenized into the set T using

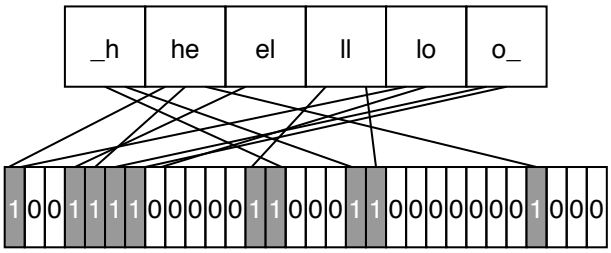

FIGURE 3.1: Creation of a static FBF ($l = 30$, $k = 2$).

n-grams of size two as tokens[1]. That is, they are decomposed into sets of every two consecutive characters of the field's value (e.g. $\_hello\_$ becomes $\{\_h, he, el, ll, lo, o\_\}$). Each token is then hashed using k independent hash functions of H. The output values of these hash functions are considered indices of v_{FBF}. Each of the corresponding bits, i.e. those in v_{FBF} with an index equal to at least one of the output values, is set to 1 (figure 3.1). After this, v_{FBF} is the FBF for the field f. To reduce the success rate of dictionary attacks against FBFs, a keyed cryptographic hash function (e.g. HMAC) must be used to hash tokens [43].

Dynamic sizing

Ideally, around 50% of the bits in a FBF are set to 1 (i.e. a uniform distribution). When this percentage deviates too much, a malicious party might infer the length of the field's value and/or the value of the parameter k [14]. To accomplish this, the predetermined value of m_{FBF} in the above procedure has to be replaced by a value that has been dynamically determined. This way v_{FBF} can be made shorter or longer to ensure not too many bits will remain 0. An equation for dynamically determining the value of m_{FBF} for FBFs of the same field[2], which is called *dynamic FBF sizing*, is provided in [14]:

$$l = \frac{1}{1 - \sqrt[kg]{p}}$$

This equation introduces two new variables: g and p. The first, g, denotes the average amount of tokens in T for the specific field. The variable p stands for the probability that a bit in the resulting FBF remains 0. Because we aim for a uniform distribution of the bit values, p is set to a fixed value of 0.5. Figure 3.2 illustrates the creation of a dynamic FBF, compared to the static FBF (figure 3.1) the bits are much more uniformly distributed.

[1] Fields need to be padded, on the left and on the right, with whitespace first (indicated with _ here).
[2] Here, using database terminology, field means a column of a table and not a cell of a single row.

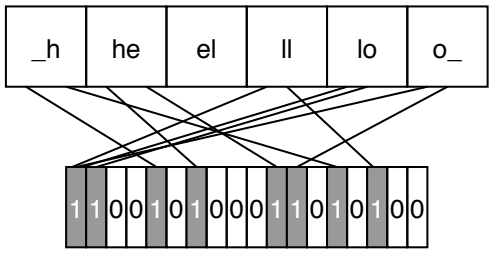

FIGURE 3.2: Creation of a dynamic FBF ($g = 6, l = 18, k = 2$).

Numerical data

Encoding of numerical data can also be done using Bloom filters. Instead of using n-grams as tokens, neighboring numbers are used as tokens to construct T [40]. Consider an integer value x, in this case the tokens are the numbers in the range $[x - b, x + b]$, with an interval of b_{intv}. Here, b and b_{intv} are parameter values. When encoding the value 10, with parameters $b = 2$ and $b_{intv} = 1$, the tokens will be $\{8, 9, 10, 11, 12\}$. A lower b value ($1 \leq b \leq 4$), increases accuracy, as T in that case will primarily contain numbers close to x. However, a greater degree of privacy protection is achieved by using a higher b value. Using $b = 5$ offers similar accuracy and privacy protection as the textual encoding counterpart [40].

Tokenization of floating-point values is slightly different, as the method for integer values might result in differently aligned neighbors for such values. For instance, 5.5 and 5.6 are very close, but $\{4.5, 5.0, 5.5, 6.0, 6.5\}$ and $\{4.6, 5.1, 5.6, 6.1, 6.6\}$ have no tokens in common. As a solution, the neighboring numbers are based on x', which is determined for a floating-point value x with the following function [40]:

$$x' = \begin{cases} x & x \bmod b_{intv} = 0 \\ x - (x \bmod b_{intv}) & x \bmod b_{intv} < b_{intv} \,/\, 2 \\ x + (b_{intv} - (x \bmod b_{intv})) & x \bmod b_{intv} \geq b_{intv} \,/\, 2 \end{cases}$$

This function returns the closest number to x on an arithmetic sequence with interval b_{intv}. The value x' is not included in T, but will only be used to determine the neighboring numbers. Consider the floating-point value 5.6, with parameters $b = 2$ and $b_{intv} = 0.5$. Then x' will be 5.5 and the tokens will be $\{4.5, 5.0, 5.6, 6.0, 6.5\}$.

3.1.2 Record-level Bloom filters

FBFs of a record can be combined into a single RBF. This has two main advantages. First, it reduces the amount of comparisons that have to be performed with a factor equal to the amount of fields. As the similarity of two records can be determined by

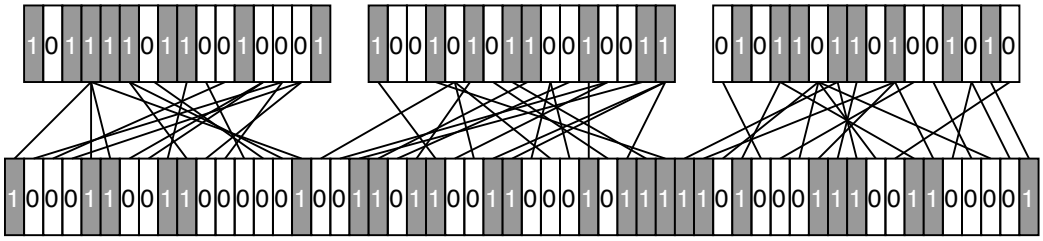

FIGURE 3.3: Sampling bits ($l = 16$, $W = \{.30, .35, .35\}$, $m_{RBF} = 54$).

a single comparison of the RBFs, instead of pairwise comparison of each FBF. Secondly, it makes it more difficult to infer original field values and thereby improves the privacy protection [11].

The procedure for constructing a RBF is as follows [14]. First, a bit vector v_{RBF} of length m_{RBF} is created. The length m_{RBF} must be set such that each FBF can take up a proportion of space that, at least, corresponds to its own length m_{FBF} and the FBF's weight w_i (i.e. m_{RBF} is the maximum of the values m_{FBF} / w_i calulated for each FBF, where $0 \leq w_i \leq 1$). Consider a FBF with $m_{FBF} = 200$ and $w = 0.25$, the RBF in that case must be at least of length 800 because 25% of its space is needed for the FBF of length 200. Next, a random sample is drawn, with replacement, from each FBF (figure 3.2). The sample size for a FBF is its weight w_i times m_{RBF}. Replacement is needed, as it might be the case that more than m_{FBF} bits are to be drawn from a FBF. In case the i^{th} drawn bit is 1, then the bit in v_{RBF} with index i is also set to 1. As a last step, the bits in v_{RBF} are randomly shuffled. This random shuffling must be preserved between RBF constructions, across the involved organizations. Otherwise the RBFs loose their ability to be compared. This can be accomplished by agreeing on a secret *random seed* that is only known by the involved organizations.

Field weighting

Weights for the fields might be known in advance, or estimated in agreement among the involved organizations. If this is not the case, the weights can be determined based on the discriminatory power of the fields. The Felligi-Sunter (FS) field weighting algorithm is commonly used for this purpose [13, 15]. This algorithm computes the agreement weight w_a and disagreement weights w_d for each field:

$$w_a = \log \frac{m}{u}$$

$$w_d = \log \frac{1-m}{1-u}$$

These weights can be combined into a single weight w_i as follows [14]:

$$w_i = \frac{w_a^{(i)} - w_d^{(i)}}{\sum_{j=1}^{|R|} w_a^{(j)} - w_d^{(j)}}$$

In the above equations m denotes the probability that the specific field is equal among record pairs M that are true matches. Similarly, u is the probability that a field is equal among true non-matching record pairs U. To calculate m and u it must be known, at least partly, what record pairs are true matches. If this is not known, as often the case with private real-world data, an Expectation Maximization algorithm can be used to estimate the probabilities m and u [46]. The public datasets used in this work allow to directly calculate m and u, based on the formal definitions [13]:

$$m_i = Pr[f_i^{(a)} \equiv f_i^{(b)} \mid (a, b) \in M]$$
$$u_i = Pr[f_i^{(a)} \equiv f_i^{(b)} \mid (a, b) \in U]$$

In these definitions, m_i and u_i represent the values of m and u for the i^{th} field. Likewise, $f_i^{(a)}$ and $f_i^{(b)}$ represent the i^{th} field of the records a and b respectively. Equality of field values is denoted with the $\equiv$ sign, and is true if the values of the two fields are exactly the same.

Intuitively, fields are assigned greater weights if agreement of two field values more likely indicates a true match than agreement by random chance, i.e. if fields are more discriminatory. Therefore, fields such as surname and address are typically assigned a higher weights than city and birthyear fields.

Chapter 4

Matching

4.1 The Hamming space

After the encoding step (chapter 3), what remains for the linkage unit to operate on is exclusively a collection of RBFs (possibly annotated with an identifier and the originating organization). As described in section 3.1, these RBFs have the property that the relative distance among them can be calculated. This property permits the collection of RBFs to be considered a metric space. An arbitrary metric space is defined as $\mathcal{M} = (X, d)$, where X is a set of items and d is a distance function [19]. To qualify as a metric space, four conditions must hold for $\mathcal{M}$ [5], for any $a, b, c \in X$:

1. $d(a, b) \geq 0$ for all a, b;
2. $d(a, b) = 0$ if and only if $a = b$;
3. $d(a, b) = d(b, a)$ for all a, b;
4. $d(a, b) \leq d(a, c) + d(c, b)$ for all a, b, c.

Since RBFs are essentially bit vectors, the applicable metric space can be defined as $\mathcal{H}^n = (\{0,1\}^n, d)$, this corresponds to a *Hamming space* of dimension n[1] [19]. The *Hamming distance*, denoted as d_h, is used as the distance function. Alternative distance functions do not satisfy all of above four conditions (e.g. *Dice coefficient*[2]) or are not meant to be used on bit vectors (e.g. *Cosine similarity*, *Euclidean distance*).

4.1.1 Hamming distance

The Hamming distance between two items is defined as the number of bit positions that are different. The function d_h can be implemented using *iteration*, *bitwise operations* or *vectors algebra*. Iteration loops over two items x and y (arrays) simultaneously and increments a counter each time when $x_i \neq y_i$. In the end the value of the counter is equal to the Hamming distance between x and y. If storing the items x and y using

[1] The dimension n will be equal to m_{RBF}.
[2] Doesn't satisfy the fourth metric space condition (triange inequality).

numerical data types, i.e. the binary representation of a number is equal to the bit vector or a part of the bit vector, than bitwise operations can be used:

$$d_h = hammingWeight(x \oplus y)$$

Above, $\oplus$ represents the XOR operator and *hammingWeight* counts the amount of bits that are set to 1. These operations combined calculate the Hamming distance. A third implementation relies on vectors algebra:

$$\mathbf{x} \cdot (-1(\mathbf{y} - 1)) + \mathbf{y} \cdot (-1(\mathbf{x} - \mathbf{1}))$$

Here, $\mathbf{x}$ and $\mathbf{y}$ are vectors and the minus 1 and multiplication by -1 should be performed element-wise. Performing the dot product results in the Hamming distance.

A micro-benchmark has been performed to get an indication of the runtime performances of each of the possible Hamming distance calculation implementations (see appendix A.2). Based on the results of this benchmark, the bitwise implementation is chosen as the to-use implementation as it offers significantly better runtime performance than both the iteration and vector algebra implementations.

4.2 Nearest neighbour

Matching records within the Hamming space can be generalized to a k-nearest neighbour(s) (k-NN) [19] problem, where $k = 1$ for PPRL or $k \geq 2$ for the similarity search variant (PPSS, section 1.1.2). Characteristics that still set apart PPRL are:

- the Hamming space of RBFs is typically high-dimensional ($n > 3000$);
- few k-NN structures are explored for high-dimensional Hamming space [31];
- the nearest neighbour(s) must be within some distance-threshold radius.

Assuming a two-database scenario, an interpretation of the PPRL process steps (section 1.1.1) that fulfill the matching of records based on k-NN, could be as follows. First, the records of one database are indexed (step 3 of the PPRL process) similar to how indexing is performed with k-NN [27]. Then, every record of the other database is used as query input for the constructed k-NN index structure to find its closest neighbour(s) in the other database (step 4 of the PPRL process). A *match* or *non-match* classification is given to the identified neighbour(s), if any, when a certain distance-threshold is not exceeded (step 5 of the PPRL process). The actual value of the distance-threshold is a parameter and depends on the implementation. In case of multiple *match* classifications, only the top k neighbour(s) are kept. In the next chapter (5) two alternative implementation approaches that follow this general procedure are presented, evaluated and discussed.

Chapter 5

Scaling PPRL

5.1 Locality-sensitive hashing

To be able to efficiently search a metric space, such as $\mathcal{H}^n$ (section 4.1), for matching items, some preparation in the form of indexing is required. With PPRL, an indexing method to efficiently group similar RBFs (those in close proxmity based on d_h) is of interest. By only pairwise comparing RBFs in the same group the total amount of comparisons that have to be performed can be greatly reduced. This is called blocking [4]. Blocking creates groups, or *blocks*, of items based on a *blocking key* [43]. To apply blocking to RBFs, blocking keys must be created based on RBF bit vectors in such a way that similar RBFs will have the same blocking key, and thus end up in the same block. For this, *Locality-sensitive hashing* (LSH) is commonly used [43].

In contrast to cryptographic hash functions, that are designed to prevent collisions, LSH functions are hashing functions that aim to collide in case of input that is similar, i.e. in close proximity within the metric space based on d. This property can be expressed formally for an arbitrary LSH function $h : X \rightarrow U$ [19], take any $a, b \in X$:

$$d(a,b) \leq r_1 \Rightarrow Pr[h(a) = h(b)] \geq p_1$$

$$d(a,b) > r_2 \Rightarrow Pr[h(a) = h(b)] \leq p_2$$

For the LSH function to be effective, it must adhere to the inequalities $p_1 > p_2$ and $r_1 < r_2$. When it does, it is called (r_1, r_2, p_1, p_2)-sensitive to the function d. This property makes the output of a LSH function suitable as blocking keys. Items a and b are placed in the same block with at least probability p_1, if they are within a radius r_1. Because $p_1 > p_2$, items in the same block are more likely to be similar than dissimilar. Still, when placed in the same block, it doesn't mean that items will match by definition in the PPRL-sense. However, it narrows the search and thereby increases scalability compared to a pairwise linear search. Since we are using the Hamming distance as the distance function for our RBF metric space, we also need to use a LSH function that is (r_1, r_2, p_1, p_2)-sensitive to this distance function. *Hamming LSH* (HLSH) is such a function [19] and is commonly used [13, 23, 43].

5.1.1 HLSH-based blocking

In essence, HLSH is a function that samples bits from a given item [13]. If items a and b are equal, we can reason that the output of the HLSH function must also be equal, as all sampled bits will have the same values. More distant items will likely yield different outputs, as some or all sampled bits are more likely to be different.

A blocking scheme for the Hamming space based on HLSH is as follows [23]. We construct L number of hash tables, that contain mappings from each $x_i \in X$ to their corresponding HLSH function output $h(x_i)$. For the construction of each hash table, a distinct HLSH function is used that samples k bits. Each item that has the same value $h(x_i)$ within the same hash table is considered to be in the same block. Having more hash tables increases the probability that items are placed in the same block one or more times. On the other hand, too many hash tables will increase the amount of redundant comparisons. An optimal value of L can be determined by [23]:

$$L = \left\lceil \frac{\ln(\delta)}{\ln(1-\rho^k)} \right\rceil$$

In this equation, δ is a confidence parameter that indicates the probability that two similar items do not end up in the same block. The value of δ should be set to a sufficiently low value, 0.1 or 0.05 [23]. The parameter ρ is defined as [24]:

$$\rho = 1 - \frac{r_1}{m_{RBF}}$$

Sampling k bits from a RBF bit vector can be implemented in various ways. Here we consider two possible implementations. In both implementations, the positions that are sampled for each of the L hash tables must be determined at random. Similar to bit sampling during the creation of RBFs (section 3.1.2), the sampled positions must be equal among all involved organizations. To achieve this, a secret random seed can be used that is only known by the involved organizations.

Suppose we have two HLSH functions, h_1 and h_2, that sample positions $\{1, 3, 5\}$ and $\{2, 4, 6\}$ respectively. As input items consider:

$$x_1 = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}$$

$$x_2 = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

A straightforward algorithm is to iterate over each input item. At every position that is to be sampled, prepend the value of the bit to a sequence. This sequence is then used as output. By prepending instead of appending, we can interpret the output

value as a binary number. Converting the binary number to a decimal number is a storage-efficient way of representing the output compared to arrays and strings[1].

$$h_1(x_1) = 101 = 5$$
$$h_2(x_1) = 011 = 3$$
$$h_1(x_2) = 111 = 7$$
$$h_2(x_2) = 000 = 0$$

Bit sampling can also be implemented using matrix multiplication. For this we need two matrices, **X** and **K**. Each row in **X** corresponds to an input item:

$$\mathbf{X} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \end{bmatrix}$$

The matrix **K** in turn indicates column-wise which bits are to be sampled. To preserve to the order of sampled positions, incremental values are used. For the i^{th} position the value to use is 2^{i-1}. For the functions h_1 and h_2:

$$\mathbf{K} = \begin{bmatrix} 2^0 & 0 \\ 0 & 2^0 \\ 2^1 & 0 \\ 0 & 2^1 \\ 2^2 & 0 \\ 0 & 2^2 \end{bmatrix}$$

The output values can be calculated by performing a matrix-matrix multiplication:

$$\mathbf{X}(\mathbf{K}) = \begin{bmatrix} h_1(x_1) & h_2(x_1) \\ h_1(x_2) & h_2(x_2) \end{bmatrix} = \begin{bmatrix} 5 & 3 \\ 7 & 0 \end{bmatrix}$$

To get an indication of the runtime performance of the above two bit sampling implementations, a micro-benchmark has been performed (see appendix section A.3). The matrix multiplication implementation has been chosen as the to-use implementation. Based on the benchmark results, this implementations offers better runtime performance and has a lower memory-footprint.

In the remainder of this text, the term *HLSH blocking key* will be used to refer to a list, regardless of representation, of sampled bits outputted by a HLSH function.

[1] Java/Scala uses 32 bits for a single integer, compared to 16 bits per character for strings (UTF-16).

5.1.2 Spark implementations

Two Spark implementations based on HLSH (section 5.1.1) have been developed. One using the *resilient distributed dataset* (RDD) API and the other using *Spark SQL*[2]. The RDD API is the low-level API of Spark that provides more control over the performed operations. Spark SQL is a higher level API that lets Spark perform some optimization by itself using the Catalyst optimizer [2].

Both implementations consist of two phases. In the first phase (figure 5.1) the involved organizations load (and clean, if required) their records and encode them into RBFs. Also, for each RBF a set of HLSH blocking keys is generated using the matrix multiplication method as described in section 5.1.1. This whole phase corresponds to the second step of the PPRL process (Data Pre-Processing, section 1.1.1).

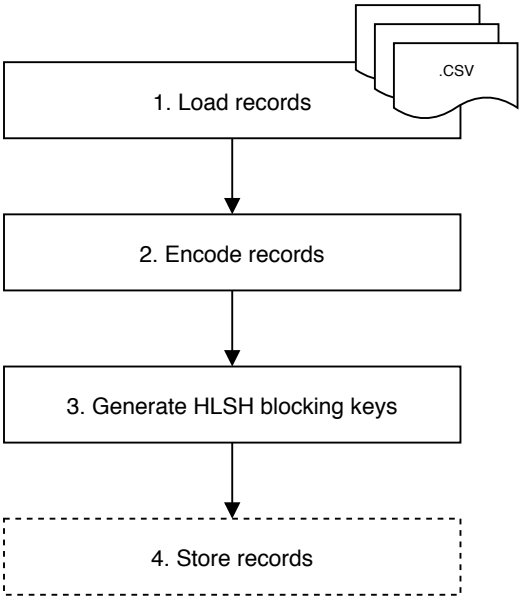

FIGURE 5.1: The first phase of both the HLSH implementations.

At the end of the first phase, the output is stored on disk so that it can be transferred to the linkage unit[3]. The linkage unit will then perform the second phase (figure 5.2). The input for the second phase is a collection with the following (pseudo-)signature:

$$[\langle \text{Id}, \text{RBF}, [\text{Key1}, \text{Key2}, \ldots] \rangle, \ldots]$$

Here, $[\ldots]$ denotes a list and $\langle \ldots \rangle$ a sequence of elements (tuple) that corresponds to a single record. The number of keys depends of the parameter L, but is at least one.

The steps of the second phase are for the most part (logically) the same between the RDD API and Spark SQL implementations, only the generation of candidates (step seven) is considerably different and influences the implementation of the subsequent steps. Before discussing the seventh step, let's first describe the preceding step that

[2] Also refered to as the *Dataframe API* or *Dataset API*.
[3] This step is omitted during benchmarks of the implementations.

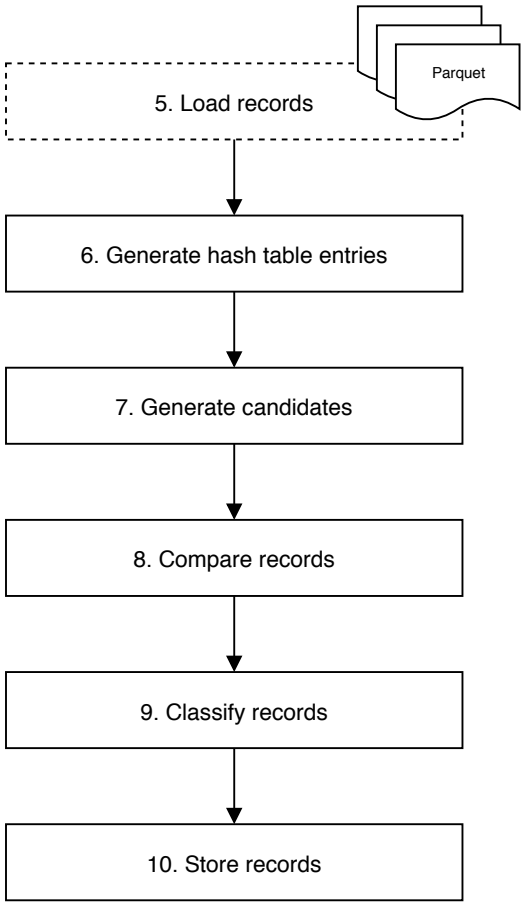

FIGURE 5.2: The second phase of both the implementations.

generates hash table entries. A hash table entry is generated for each of the HLSH blocking keys of records, thus effectively multiplying the total number of items by L. This is required to allow Spark to process the entries independently and thereby the hash tables in parallel. For each record, hash table entries are generated as follows:

$$\langle \text{Id}, \text{RBF}, [\text{Key1}, \text{Key2}, \ldots] \rangle \rightarrow \begin{cases} \langle \text{Key1}, \text{Id}, \text{RBF} \rangle \\ \langle \text{Key2}, \text{Id}, \text{RBF} \rangle \\ \ldots \end{cases}$$

Based on the hash table entries, the RDD API implementation generates candidate record pairs by performing a *cogroup*[4] on the HLSH blocking keys. This transformation results in a collection of groups (i.e. hash table), one for every distinct HLSH bloking key. All groups contains two records lists, one for each of the two databases. The pairwise combinations of the lists of records within each group (hash table) are the generated candidate record pairs. The (pseudo-)signature of this step is:

$$[\langle \text{Key1}, \text{Id1}, \text{RBF1} \rangle, \ldots] \rightarrow [\langle \text{Key1}, [\langle \text{Id1}, \text{RBF1} \rangle, \ldots], [\langle \text{Id2}, \text{RBF2} \rangle, \ldots] \rangle, \ldots]$$

[4] https://spark.apache.org/docs/latest/rdd-programming-guide.html#transformations

The Spark SQL implementation generates candidate record pairs differently based on the hash table entries. Namely, it performs a *JOIN*[5] based on the HLSH blocking keys. This results in the same candidate records pairs as the RDD API implementation, but they are not grouped by HLSH blocking keys and thus can, and must, be processed independently. The (pseudo-)signature of this operation is as follows:

$$[\langle \text{Key1}, \text{Id1}, \text{RBF1} \rangle, \ldots] \rightarrow [\langle \text{Key1}, \text{Id1}, \text{RBF1}, \text{Id2}, \text{RBF2} \rangle, \ldots]$$

The remaining three steps (8-10) are very similar between the two implementations. Comparing records is in both cases done using bitwise operations (section 4.1). Assigning classifications is done based on a simple threshold function, the threshold value itself is a parameter. Storing records is done using the *Parquet*[6] format, but could be replaced by any other format compatible with Spark and Hadoop (HDFS).

5.1.3 Single-node benchmark

The two implementations have been benchmarked against LSHDB (section 2.2), using the the *North Carolina voting register* (NCVR) dataset. The NCVR dataset is a public dataset with real-world personal data, a more detailed described, along with the appropriate parameters, is provided in section 6.1. Since LSHDB cannot be deployed on a cluster, we used a single-node setup for the comparison benchmark. A Docker-based Spark cluster[7] (version 2.1.1, one master node, one worker node) has been used to run the two Spark implementations. LSHDB has been run by directly invoking the `.jar`[8] file (JDK8). The host machine contained 8 CPU cores (Intel i7-6700), 32GB RAM and a SSD for storage. The average of four runs are used as the final result for each configuration, plotted in figure 5.3.

What stands out is that LSHDB is faster than both the Spark implementations for up to a million records. This can be explained by the overhead of Spark, that, in addition to the actual work, also performs *Job Scheduling*[9] and other cluster-related tasks, even on a single-node setup. LSHDB is a lot more lightweight in that area, it's a pure Java implementation that immediately and solely will work on the PPRL task. The RDD API implementation performs worse than both the Spark SQL implementation and LSHDB for all database sizes. Spark seems to optimize the Spark SQL implementation better as the database size increases, from around 2 million records a moderate runtime reduction, compared to LSHDB, is observable. Thereby making the Spark SQL implementation the fastest option for handling multiple millions of records, and LSHDB the fastest option for databases with sub-million records.

[5] An SQL inner equi-join, to be exact.
[6] `https://parquet.apache.org`
[7] `https://github.com/gettyimages/docker-spark`
[8] Java Archive; the file format of a Java executable.
[9] `https://spark.apache.org/docs/latest/job-scheduling.html`

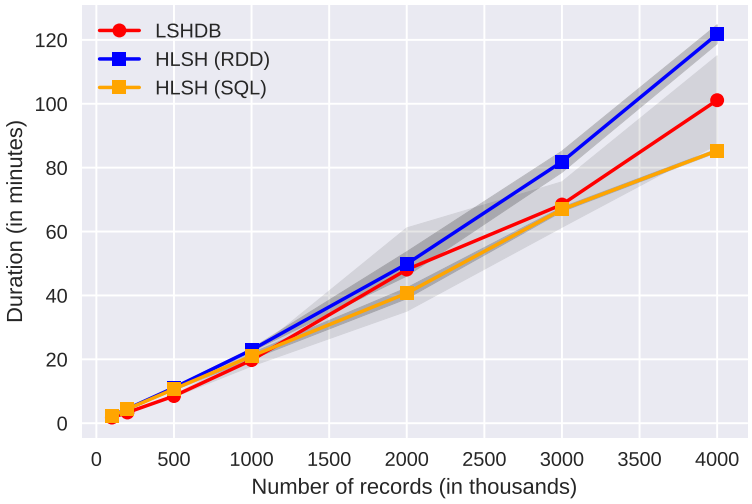

FIGURE 5.3: Singe-node runtimes of the HLSH implementations.

Implementation	Records	Precision	Recall	Accuracy	F1 Score
RL Toolkit	50,000	0.99	0.89	0.98	0.94
LSHDB	50,000	0.99	0.95	0.99	0.98
HLSH	50,000	0.99	0.88	0.99	0.93
RL Toolkit	100,000	0.99	0.89	0.98	0.94
LSHDB	100,000	0.99	0.95	0.99	0.97
HLSH	100,000	0.99	0.88	0.98	0.94

TABLE 5.1: Evaluation metrics for LSHDB and HLSH.

When we look at the accuracy of LSHDB and the HLSH implementations (table 5.1). The precision, accuracy and F1-score are all on-par. Except for recall. This means that although the HLSH (SQL) implementation, compared to LSHDB, is becoming faster as more records are added, it comes with a compromise of finding fewer of the total true matching records, i.e. there are more false-negatives. In the case of privacy, we can argue that extra false-positives is worse than having extra false-negatives. For example, with the watch-list example (section 1.1.2) it would be worse to falsely accuse someone to be on a watch-list, and have its privacy violated, than the other way around. Thus, having more false-negatives might be a reasonable compromise when speed is essential. This is strictly in terms of privacy, and not in security.

The same benchmark has been ran using the *Record Linkage Toolkit*[10], an open-source library for regular record linkage. This shows that LSHDB and the HLSH implementations are on-par with regular linkage results in terms of these evaluation metrics.

[10] https://github.com/J535D165/recordlinkage

5.1.4 Discussion

A drawback of HLSH results from the fact that blocking keys are generated individually for each record without considering any of the other records in the database(s), i.e HLSH is *data-oblivious* [1]. Thus, it is not known after generating a blocking key if that key will ensure that all the true matching records will be encountered. Furthermore, with HLSH only records with the exact same blocking key(s) are considered candidate record pairs. For example, if the generated HLSH blocking keys, for a certain hash table, of two true matching records differ only by one bit, the two records won't end up as a candidate record pair based on that hash table. As a consequence multiple HLSH blocking keys have to be generated in advance for every single record to increase the probability of colliding with true matching records.

The creation of multiple HLSH blocking keys blows-ups, in terms of total amount of items in that flow through the application, the dataset in the second phase of the both HLSH implementations. This burdens the RAM requirement and increases the amount of (redundant) comparisons that have to be performed. Other indexing structures exist, such as *LSH Forest* [1, 3], that are *data-dependent* and generally are less burdened by the aforementioned drawback. One of the key features of the LSH Forest structure is that, in essence, it can peek into nearby hash tables based on the distance between the blocking keys[11]. This lowers the amount of required hash tables and has the potential to increase scalability.

In section 5.2 an alternative to HLSH is presented that tries to prevent blowing-up of the amount of items and reduce redundant comparisons by incorporating the idea of peeking into nearby hash tables, just as with the LSH Forest structure.

5.2 Pivots

A *data-dependent* way of creating blocks is to use relative distances among the records, i.e. by applying clustering based on d_h. It is data-dependent because other records in the dataset are considered when creating the blocking keys. Only records in the same cluster are compared pairwise. Thus, compared to HLSH, hash tables are replaced with clusters. The general procedure as described in section 4.2 is still adhered.

Constructing a complete view of how all the records are positioned requires $\frac{1}{2}n(n-1)$ comparisons, which boils down to (almost) a complete pairwise comparison of the database. As discussed in section 1.1.1, this does not scale. Therefore, pivots are used to mark the centroids of the clusters. By using m pivots the amount of comparisons to construct the clusters is reduced to mn. Pivots can be selected using a random sample or a special pivot-selection strategy [29].

[11] The Hamming distance can be used to calculate the distance between two hash tables, since the HLSH blocking keys are essentially, just as RBFs, bit vectors.

A procedure for creating clusters based on pivots, described in [36], is as follows. First, a set of pivots is determined based on one of the two databases. This is done by taking a sample of size $3m$ [9], where m is the number of pivots, and than iteratively select pivots that are farthest away. The farthest away pivot is the pivot with the largest minimum distance to any of the previously selected pivots. Then, from the dataset that was used to determine the pivots, all non-pivots are assigned their most nearby pivot. Important is the the radius of a pivot, which is defined as the distance from a pivot p to the farthest away record r_f that is assigned to that pivot:

$$\text{rad}(p) = d_h(p, r_f)$$

Figure 5.4 illustrates an example of possible pivots and their assigned records. For the illustrative purpose the pivots and records are shown in a (fictional) 2D space.

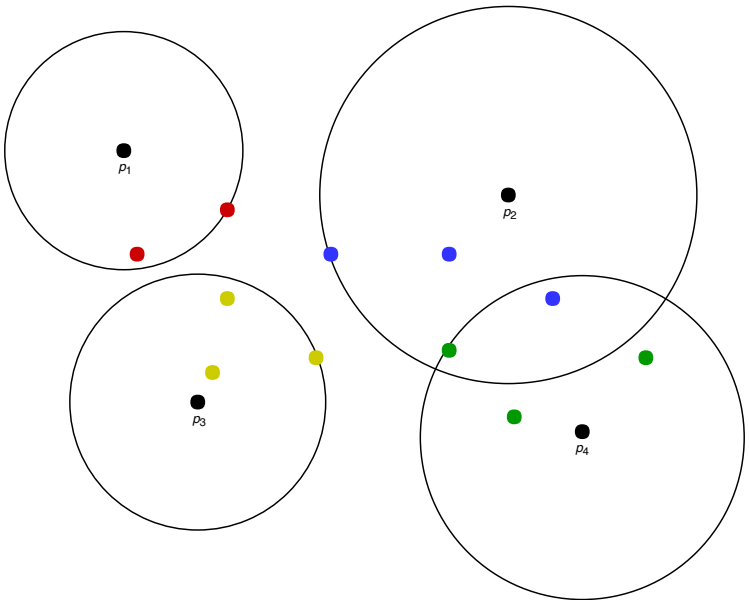

FIGURE 5.4: Clusters based on pivots (circles represent pivot radii).

Once the pivot clusters are constructed, the items from the other database are used as queries. This is done in two steps. First it is determined which pivot raddii are overlapping with the query record's radius. The radius of a query record q is the threshold-distance (section 4.2), i.e. any record that lies within this radius is considered a match. The query record is compared to each of the records that is assigned to one of the overlapping pivots. Before performing the complete comparison, the triangle inequality is checked (fourth condition of a metric space, section 4.1):

$$d_h(p, q) \leq d_h(p, r) + d_h(r, q)$$

$$d_h(p, q) \leq d_h(p, r) + \text{rad}(q)$$

$$d_h(p, q) - d_h(p, r) \leq \text{rad}(q)$$

The query record's radius can be used as a (temporary) substitute for the $d_h(r, q)$ distance, as shown above. If this inequality doesn't hold, it is impossible for the two record to have a distance lower or equal than the threshold-distance [36]. Figure 5.5 illustrates the querying. The query record q_1 overlaps with p_2, p_3 and p_4, therefore all the blue, orange and green records will be compared to.

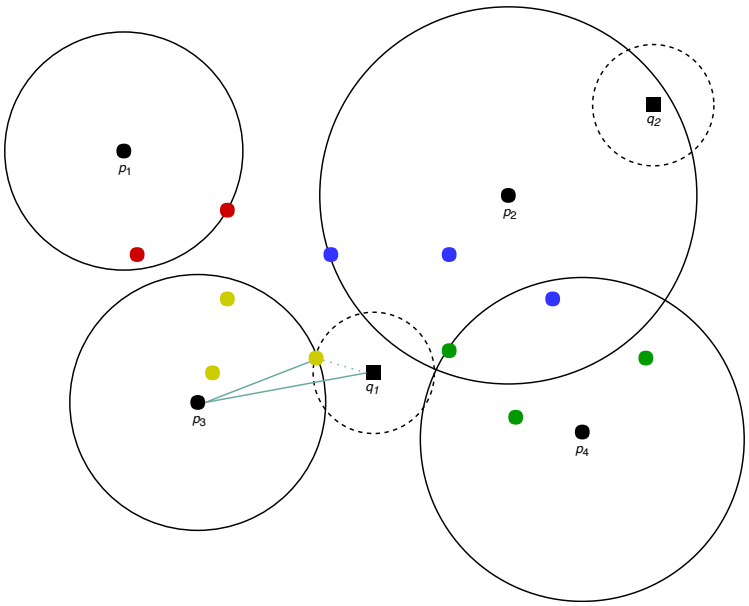

FIGURE 5.5: Querying the pivots clusters.

The overlap with a pivot can be interpreted as being part of a block, i.e. overlap becomes the blocking key. Since there is no maximum to the radius of pivots, they can potentially grow very large. Also, there is no limit to the amount of blocks a query record can be part of. We can reason that, potentially, a query record will be part of a lot of blocks because pivot radii have grown very large. Among these blocks can be blocks with pivots with only very distant records, because the pivot radius is determined based on one direction, but is applied in all directions. For instance, in figure 5.5, q_2 overlaps with p_2, but the assigned records are on the other side.

To support this reasoning, we looked at the amount of records that are typically assigned to a single pivot (table 5.2) and the amount of pivots that a single query records typically overlaps with (table 5.3). These are based on averages. We can observe that the amount of records that are assigned to a pivot decreases with the total amount of pivots, i.e. they seem to distribute nicely between pivots. However, endlessly increasing the amount of pivots imposes two problems. First, the amount of overlap increases with the number of pivots. Second, the pivot selection process itself requires $\frac{1}{2}m(m-1)$ comparisons, which does not scale well. Especially since the pivot selection cannot be fully parallelized. In a proposed distributed setup for the discussed pivot-based blocking approach, a pre-selection is performed in parallel but in the end still requires $\frac{1}{2}m(m-1)$ comparisons on a single node [17].

Pivots	Records	Mean	Std.	Min.	Max.
400	50,000	125	73	15	473
800	50,000	62	37	7	272
1,200	50,000	42	28	4	238
400	100,000	250	145	46	845
800	100,000	124	74	19	544
1,200	100,000	83	51	9	475

TABLE 5.2: Amount of records assigned to a pivot.

Pivots	Records	Mean	Std.	Min.	Max.
400	50,000	42	12	5	97
800	50,000	49	16	4	128
1,200	50,000	51	18	5	138
400	100,000	46	13	5	106
800	100,000	56	18	7	152
1,200	100,000	59	20	6	171

TABLE 5.3: Amount of overlapping pivots to a query record.

5.2.1 Strict pivot-based blocking

To improve the scalability of the discussed pivot-based blocking we propose a more strict pivot-based blocking approach. The aim is to decrease runtime by lowering the required pivots. The approach is as follows. First, m_{prime} *prime-pivots* are selected, say $m_{prime} \leq 2,048$ depending on the dataset size. Prime-pivots are selected using the farthest-away strategy (section 5.2), but with an additional requirement. Namely, that the smallest distance to a previously selected pivot is not lower than pp_{min}. If this minimum distance is exceeded, the prime-pivot selection is stopped. The goal of selecting prime-pivots is to divide the RBF space into non- or slightly-overlapping parts. The radius of a prime-pivot is the distance to the nearest prime-pivot $pp_{nearest}$ times c_{radius} (e.g. 0.5 or 0.75). This will allow the prime-pivots to fill up empty space.

As a next step, each record, from the database that was used to select the prime pivots, are assigned to one of the prime pivot (illustrated in figure 5.6). Prioritized in the following order: **1)** the record lies within the radius of the prime pivot; **2)** the record has the largest overlap with the prime pivot; **3)** the prime pivot is the nearest prime pivot. Based on the records that are assigned to a prime pivot, at most m_{sub} *sub-pivots* are selected using the farthest-away strategy. Again, with an additional requirement that the smallest distance to a previously selected pivot is not lower than sp_{min}. Similarly to prime pivots, the radius of a sub-pivot is the distance to the nearest sub-pivot $sp_{nearest}$ times the parameter c_{radius}. This prime-/sub-pivot scheme can be parallelized better than the original scheme (section 5.2). The prime-pivot selection still has to happen on a single node, but the amount is limited. The sub-pivot selection can be performed in parallel at different nodes for each prime-pivot.

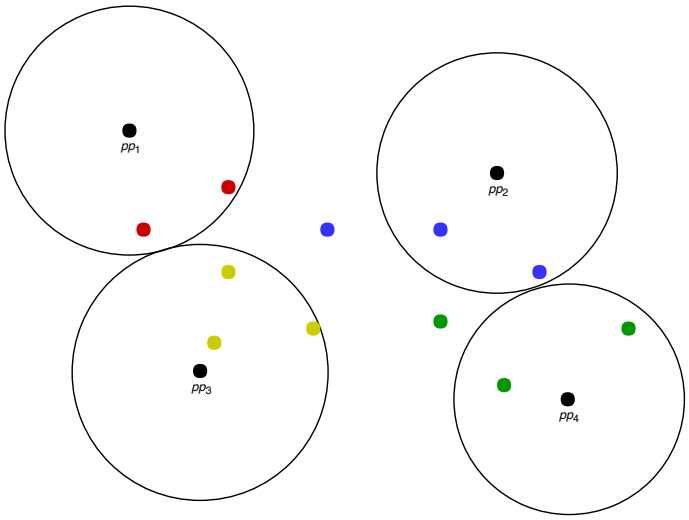

FIGURE 5.6: Clusters based on prime pivots (circles represent radii).

m_{prime}	m_{sub}	**Pivots**	**Records**	**Mean**	**Std.**	**Min.**	**Max.**
64	32	1,600	50,000	31	29	1	374
128	64	3,165	50,000	16	15	1	183
258	128	6,064	50,000	8	7	1	111
64	32	1,504	100,000	65	73	2	180
128	64	3,072	100,000	33	35	1	917
256	128	5,878	100,000	17	18	1	385

TABLE 5.4: Amount of records assigned to a sub-pivot.

It is still the case that increasing the pivots, through m_{prime} and m_{sub}, results in fewer assigned records to a single pivot (table 5.4). And with more pivots, there is also more overlap between query records and pivots (5.5). However, the amounts are much smaller than with the original pivot-based blocking approach (section 5.2).

5.2.2 Strict pivot-based (Euclidean)

With strict pivot-based blocking the RBF space is divided in several sub-spaces based on the prime-pivots. Within the sub-spaces there are only a modest amount of sub-pivots that a query record can quickly be compared to, as done in the strict pivot-based scheme (section 5.2.1). The distances from a query record to all the sub-pivots can be put into a single vector to create a *Euclidean* vector[12]. The idea is that this vector can substitute the RBF bit vector as the query record, since exact/similar RBFs will have exact/similar distances to sub-pivots and therefore also exact/similar Euclidean vectors. The same goes for distant RBFs that will have dissimilar Euclidean vectors. The idea is illustrated in figure 5.7, where an analogy can be made to the

[12]We mainly use this term to distinguish bit vectors (Hamming) and numeric vectors (Euclidean).

m_{prime}	m_{sub}	**Pivots**	**Records**	**Mean**	**Std.**	**Min.**	**Max.**
64	32	1,600	50,000	13	4	1	32
128	64	3,165	50,000	17	7	1	64
256	128	6,064	50,000	18	9	1	128
64	32	1,504	100,000	11	4	1	32
128	64	3,072	100,000	13	6	1	64
256	128	5,878	100,000	14	7	1	128

TABLE 5.5: Amount of overlapping sub-pivots to a query record.

global positioning system (GPS): sub-pivots are satellites that are used to position the query record in the space. Since the Euclidean vectors are in a different metric space, namely the Euclidean space $\mathbb{R}^n$, it opens up the possibility for more indexing structures such as K-D Tree [6] or Ball Tree [32]. These are out the scope of this work.

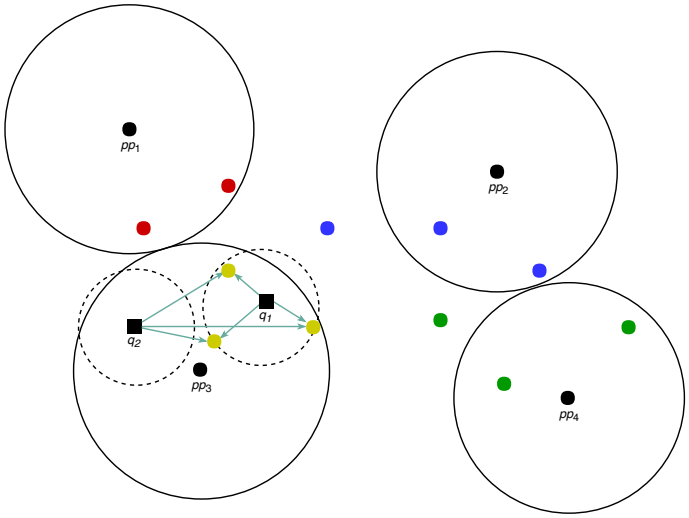

FIGURE 5.7: Euclidean vectors based on sub-pivots.

5.2.3 Spark implementation

Three pivot-based Spark implementations have been developed, these follow the original pivot-based blocking scheme (section 5.2), the strict pivot-based blocking scheme (section 5.2.1) and an implementation that utilizes the pivot-based Euclidean vector (section 5.2.2) respectively. These implementation consists of two phases, the first is executed by the involved organizations and the second by the linkage unit.

In the first phase (figure 5.8), records are loaded and cleaned, if required, by the involved organizations. Also, they are encoded into RBFs. Since the pivot-based schemes are data-independent no (pre-)indexing steps are performed, as all the records are record for that. This phase represents the second step of the PPRL process (Data

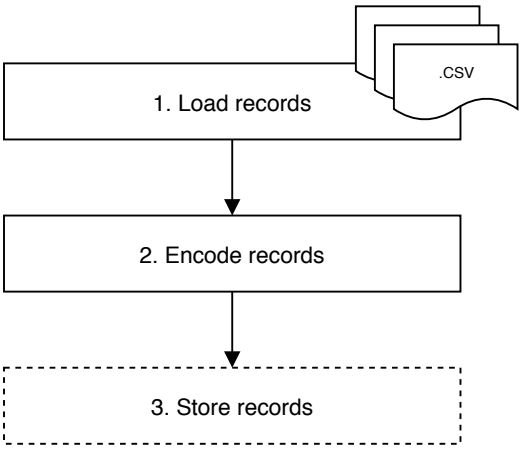

FIGURE 5.8: The first phase of the three Pivot-based implementations

Pre-Processing, section 1.1.1). The output is a collection with the (pseudo-)signature:

$$[\langle \text{Id}, \text{RBF} \rangle, \ldots]$$

The second phase (figure 5.9) differs in the two steps after loading the records (6 and 7). The remaining steps (8-10) are equal to the HLSH implementation, discussed in section 5.1.2. The original pivot-based scheme generates m pivots based on a $3m$ random sample. Both the strict pivot-based schemes generate prime-pivots on a sample up to 50,000. This larger sample is required so that as much as possible distant RBFs are included, so that the prime-pivots can be generated in such a way that they partition the space as much as possible. These pivots, for all implementations, are are *broadcasted*[13] and have the same signature as a regular RBF collection, i.e. it is a subset. The strict pivot-based schemes also generates and broadcasts sub-pivots based on the records that are assigned to the prime-pivots. This results in the collection:

$$[\langle \text{PrimeId}, [\langle \text{SubId}, \text{RBF} \rangle, \ldots] \rangle \ldots]$$

The strict pivot-based Euclidean vector approach creates *Annoy*[14] models for each of the prime-pivots. Annoy is a library for k-NN, it uses random projection [7], a form of LSH for the Euclidean space. These models are trained, using the default configuration, on the Euclidean vectors (section 5.2.2) created for each of the records assigned to a prime-pivot. After training the models are broadcasted. Because every worker node will have a local copy of all the models, every worker node will be process any arbitrary query. Since no joins are required, this can be highly parallelized.

[13] Spark's broadcasting mechanism is an efficient way of providing every worker node with a local copy of variables and collections.

[14] https://github.com/spotify/annoy

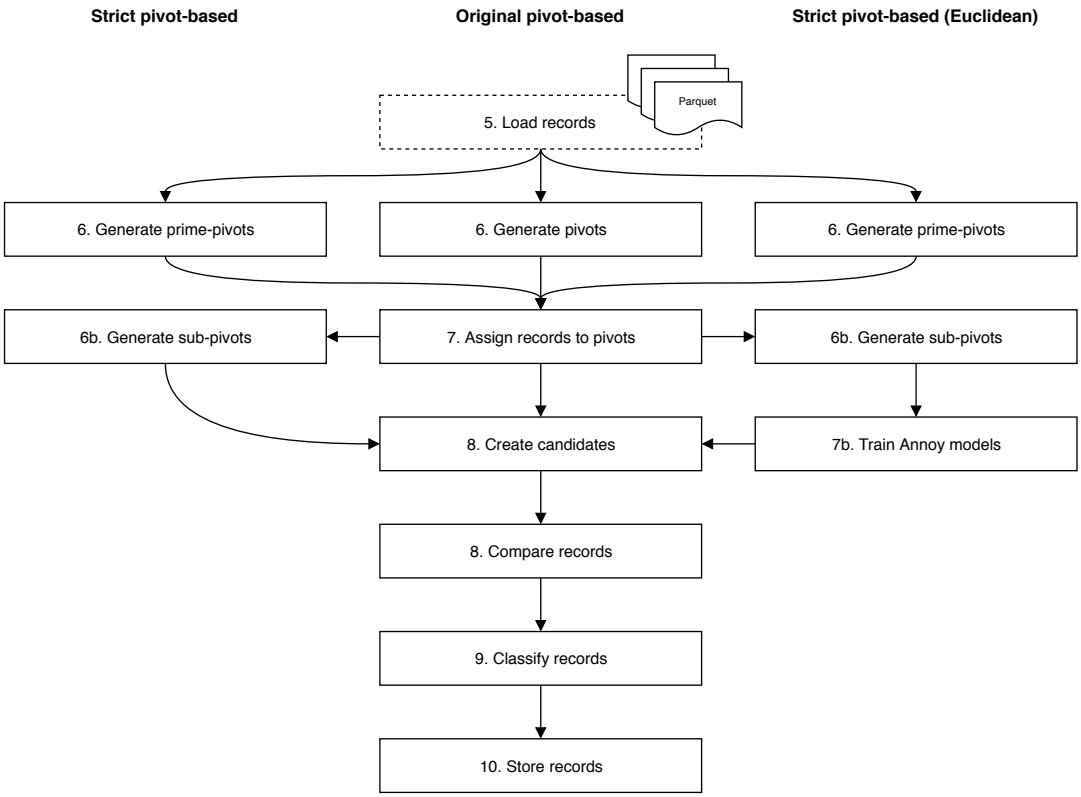

FIGURE 5.9: Steps for each of the Pivot-based implementations.

5.2.4 Single-node benchmark

Each of the pivot-based implementations have been benchmarked. The same dataset and single-node configurations have been used as the HLSH benchmark (section 5.1.3). The benchmark results have been plotted in figure 5.10. For the original pivot-based 1,200 pivots have been used. The runtime performance is a lot more worse than LSHDB and doesn't scale well. The more strict pivot selection has better performance up until 2,000,000 records, after that the runtime increases dramatically. Of all the implementations the one based on Euclidean has the best runtime performance, but is still significantly slower than LSHDB and both the HLSH implementations.

Implementation	Records	Precision	Recall	Accuracy	F1 Score
LSHDB	50,000	0.99	0.95	0.99	0.98
Pivots	50,000	0.99	0.89	0.99	0.94
Pivots (Strict)	50,000	0.99	0.79	0.98	0.88
Pivots (Euclidean)	50,000	0.99	0.87	0.98	0.89
LSHDB	100,000	0.99	0.95	0.99	0.97
Pivots	100,000	0.99	0.88	0.98	0.94
Pivots (Strict)	100,000	0.99	0.80	0.96	0.89
Pivots (Euclidean)	100,000	0.99	0.79	0.96	0.88

TABLE 5.6: Evaluation metrics for LSHDB and Pivot-based.

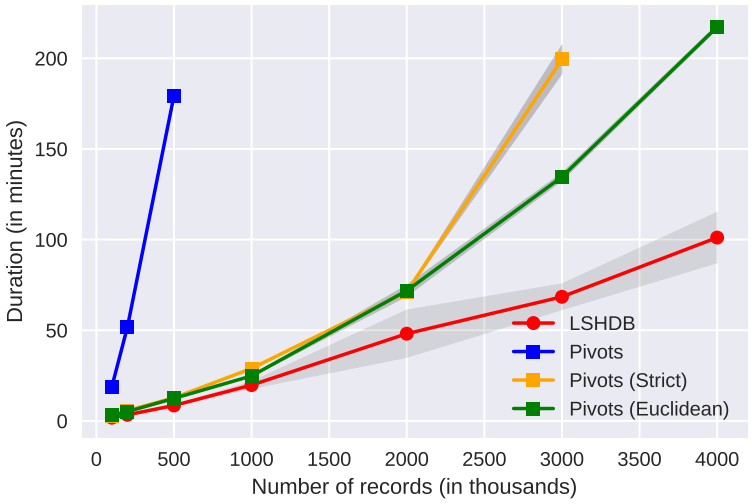

FIGURE 5.10: Singe-node runtimes of the Pivots implementations.

Based on the evaluation metrics in (table 5.6), all pivot-based implementations have worse recall compared to LSHDB. This is due to the pivot-based implementations not finding as much as true matching records as LSHDB. What is interesting, is the differences between the pivot-based implementations. The strict adjustments (section 5.2.1) to the original pivot-based technique, increases runtime at the cost of recall, i.e. finding less true matches. The Euclidean pivot-based implementation can again improve upon the strict pivot-based implementation, at the cost of recall.

5.2.5 Discussion

Despite being data-dependent and, especially the two strict variants, only blows up the dataset in the worst case[15], the pivot-based schemes all perform worse than both the LHSDB and HLSH implementations. This might have several causes. First, it may be that the particular computations of HLSH can be performed more efficient, despite requiring some joins and the associated coordination. Second, since the pivot-based schemes are data-dependant it might require fine-tuning the parameters for each (sub-)dataset used. On the same line, it might be that for some datasets not enough pivots can be generated that are sufficiently distant. The reliance on the distribution of the dataset is a major drawback of the pivot-based schemes.

Furthermore, the Euclidean vectors where consolidated in a model based on default configuration and tested with only a single indexing method. It might also be that for these specific vectors, that are in a relatively low dimension $n \leq 128$, another indexing structure is more performant. Tweaking these might improve the performance, in both runtime and accuracy of the Euclidean pivot-based implementation.

[15]With HLSH every record is guaranteed to be multiplied by L, where with pivots this depends on the prime pivots and may well be 1.

Chapter 6

Applications

6.1 The NCVR dataset

The *North Carolina Voting Regiser*[1] (NCVR) dataset is one of the few publicly available real-world dataset with around 7,000,000 records of personal data. This dataset is therefore commonly used as a benchmark dataset for PPRL [43]. For the benchmarks in this work, several sub-dataset have been generated based on the NCVR dataset. These sub-datasets mimic a two-database situation where each dataset pair (A & B) has a total record overlap of ≈ 10%. The dataset pairs are listed in table 6.1.

Name	Records A	Records B	Total records	True matches
50k	49,999	50,000	99,999	10,000
100k	99,993	99,992	199,985	19,998
200k	199,989	199,988	399,977	39,998
500k	499,980	499,986	999,966	99,998
1m	999,950	999,948	1,999,898	199,991
2m	1,999,908	1,999,897	3,999,805	399,984
3m	2,999,849	2,999,857	5,999,706	599,973
4m	3,999,791	3,999,825	7,999,616	967.950

TABLE 6.1: The generated NCVR sub-datasets.

As discussed in section 3.1, real-world personal data often is corrupted with, for example, typing errors. The NCVR dataset is in this perspective a relatively clean dataset, and contains only few corruptions. To make sure the PPRL techniques work properly with unclean data, i.e. not only on exact matches, the generated datasets have been corrupted using GeCo [39]. GeCo is a tool that can realistically corrupt personal data, such as names and addresses. Half of the overlapping records of each generated dataset pair have been randomly corrupted. This is around 5% of the total amount of records, evenly split among the A and B parts.

Encoding parameters for the used NCVR fields have been determined upfront and are listed in table 6.2. These have been used by all benchmarks, including LSHDB.

[1] https://s3.amazonaws.com/dl.ncsbe.gov/data/list.html

Field	**Tokens**	g	k	w
Firstname	Text	7.0	15	22
Lastname	Text	7.4	15	24
Birthyear	Numeric	6.0	15	5
Address	Text	18.0	15	29
City	Text	9.9	15	16
Zipcode	Text	6.1	15	4

TABLE 6.2: The used NCVR fields with parameters.

6.2 Cluster deployment

In chapter 5, the Spark implementations have been ran using a single-node configuration, so a fair comparison can be made with LSHDB. However, Spark applications primarily target clusters. To measure how well the created Spark implementations scale on a cluster, the same benchmark performed in chapter 5 has been repeated, also using the NCVR dataset, on a cluster of various sizes (10, 20, 40, 80 and 160 worker nodes). For this, only the HLSH implementations (section 5.1) have been considered, because only they showed a reasonable scaling curve with the single-node evaluation. The pivot-based implementation based on Euclidean vectors would also be interesting to deploy on a cluster, unfortunately the required native component (Annoy) has a compatibility issue[2] with the used cluster[3]. The worker nodes used had the ability of 4 CPUs (cores) and 16 GB RAM.

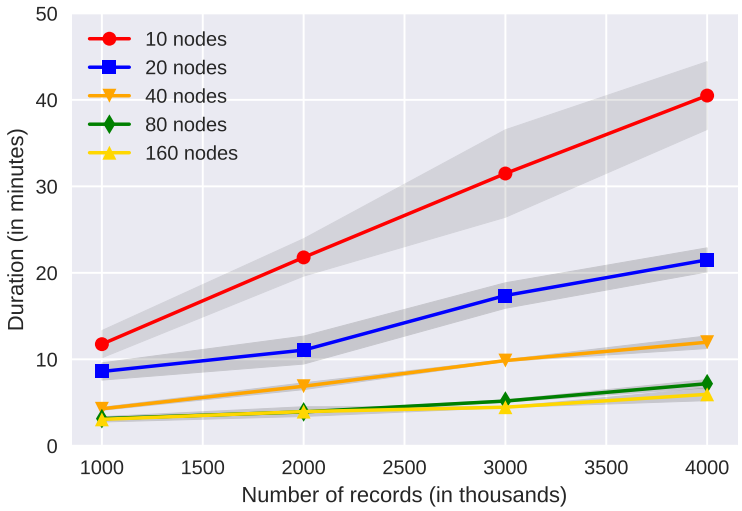

FIGURE 6.1: Cluster runtimes of the HLSH (RDD) implementation.

The averages of four runs for each configuration (number of nodes/records) for the RDD and Spark SQL implementations have been plotted in figure 6.1 and figure 6.2

[2] Linux containers are used, but within these containers `glibc`, required by Annoy, isn't available.
[3] A *Hortonworks Data Platform* (HDP) v2.3.4 deployment (also see acknowledgments in section 1.2.1).

respectively. Only runtime improvement is considered here, as it is assumed that the accuracy stays the same as in section 5.1.3, i.e. the computations are the same.

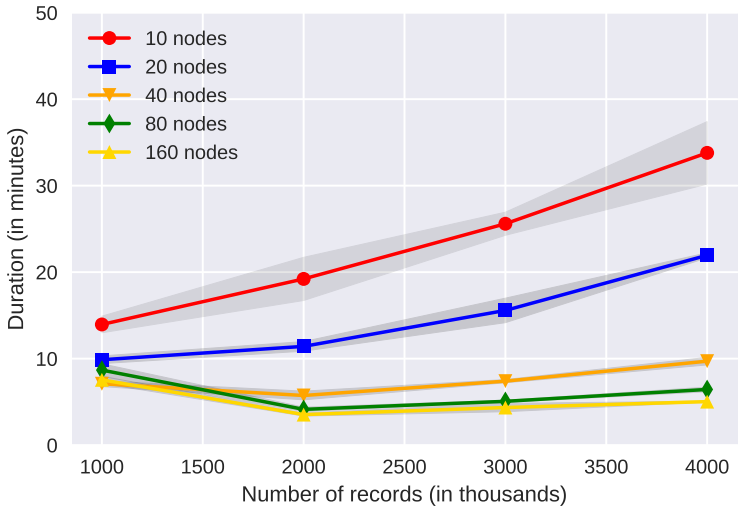

FIGURE 6.2: Cluster runtimes of the HLSH (SQL) implementation.

The runtime of the RDD implementation for the 1m dataset can be halved, compared to the single-node deployment, when using 10 nodes. This reduction steadily increases to two-third for the largest 4m dataset. Doubling the amount of nodes to 20, has only a modest effect for the 1m dataset (one-quarter runtime reduction), but for the other datasets again halves the runtime. Further doubling the amount of nodes decreases the runtime, but in lesser quantities. With 40 nodes the runtime is almost halved, compared to 20 nodes, with 45% on average across all datasets. This is reduced with on average 40% by again doubling the amount of nodes to 80 nodes. Using 160 nodes doesn't noteworthy decrease the runtime anymore, it even is slightly slower than using 80 nodes for the 2m dataset. The maximum reduction is 18% for the largest 4m dataset. We can conclude that for the RDD implementation, doubling the nodes up to 40 nodes has favorable runtime reductions, but beyond that the cost-benefit becomes effectively less tot none.

The Spark SQL implementation has a similar relation between runtime reduction and doubling the amount of worker nodes. However, the reduction between the single-node deployment and a 10-node cluster deployment is much greater with on average 58 % for the 2m, 3m and 4m datasets. The 1m dataset is the exception, with the Spark SQL implementation it seems that this size of dataset doesn't fully take advantage of the extra nodes. For amounts of nodes larger than 40, the 1m dataset is processed even slower than the 2m, 3m and 4m datasets. It is hard to pinpoint the exact reason, since Spark's SQL optimizer, Catalyst, is implemented as a black box.

We can extrapolate that for the larger cluster deployments (80 and 160 nodes) the runtimes will diverge, as the 20-node and 40-node lines, when using even larger datasets. In that case it becomes beneficial again to use the larger amount of nodes.

Chapter 7

Future work

This work considered a limited scope, in terms of tools, techniques and implementations. In each of these areas lie opportunities for further work. In this chapter some of these are mentioned.

Other Hadoop-ecosystem tools

Only Apache Spark has been used in this work as a framework for the various implementations. However, the Hadoop-ecosystem consists of multiple other candidates with each its advantages and disadvantages. Another good option is to use Apache Flink[1], as also suggested in recent other work [17, 43]. Flink provides a similar abstraction of computations as Spark, but mainly focuses on stream processing. This in contrast to Spark, that mainly focuses on batch processing. Streaming can be a good fit for PPRL, when there is a constant inbound of checks that have to be performed, as in the border security application (section 1.1.2).

Using a (big) data stores to store records and/or intermediate values is also worth exploring. In the implementations in this work, all data is managed by Spark itself. It might be the case that certain data operations can be performed more efficient when using a optimized data store, such as Apache HBase[2] or Apache Cassandra[3].

Other hashing/blocking techniques

The LSH technique used in this work relies on hashing of items, such that similar items collide by having the same hash function output (section 5.1). The effectiveness of LSH depends on the parameters used and on the specific dataset used. The same goes for the pivot-based schemes (section 5.2). A new development are hashing functions based on the *Learning to Hash* principle [45]. This is a technique that can

[1] https://flink.apache.org
[2] https://hbase.apache.org
[3] https://cassandra.apache.org

be used to learn a hashing function, based on the data characteristics and distribution, that most effectively accomplishes what LSH functions also try to accomplish. These methods are often based on *Machine Learning* and even *Deep Learning* [12].

In the same area, using (learned) dimensionality reduction functions can be used. The PPRL methods used in this work rely heavily on bit vectors (RBFs) in the Hamming space. A method for dimensionality reduction of these vectors is the, not commonly used, *Logistic Principal Component Analysis* [34]. Applying this technique can speed up PPRL applications, as it will be working in a lower dimension [19].

Other implementations

Implementation-wise there are also a few opportunities to explore. For instance, experimenting with native components on a larger cluster. The benefit of using native components is that they can perform operations on a much lower level[4]. The pivot-based implementation with Euclidean vectors relies on Annoy, a native component to train k-NN models (section 5.2.2). When these models are broadcasted, a higher degree of parallelism can be obtained. As every worker node is able to process any arbitrary incoming query record. Deploying this implementation, or similar, on a cluster can give insight in how, if so, favorable such a setup is.

Another option is to resort to *graphical processing unit* (GPU) computing for acceleration. Most of the required computations can be, in someway or another, be defined as vector or matrix operations. Such operations can be performed faster by exploiting the parallel nature of GPUs [30]. GPUs can also be used to accelerate the creation of RBFs, by performing the cryptographic hashes on GPUs instead of CPUs [26].

More than two involved organizations has not been considered in this work. However, the used two-party database scenario used could be adapted to a multi-database scenario. This works by assuming a transitive property for matches. For example, take three databases A, B and C. We first compare A and B, and then B and C. Then we consider the triplet $\langle a, b, c \rangle$ a match when $a = b$ and $b = c$, where a, b and c originate from A, B and C respectively and $=$ indicates a match between two records. This could be implemented using a graph processing library, such as GraphX[5], that takes the output matches from multiple two-database scenarios as input.

An similarity search (PPSS) implementation has not been considered in this work. Two possible ways to modify the created PPRL implementations for PPSS is to increase to distance-threshold (section 4.2) and allow for multiple matches to be made for a single record. This can be combined with the above description for supporting multiple organizations. In this case, using a graph processing library, clusters can be found based on graph cliques [33].

[4] For example when implemented with C or C++.

[5] `https://spark.apache.org/docs/latest/graphx-programming-guide.html`

Chapter 8

Conclusions

In this work we have made the case that PPRL is important, and that is it beneficial to have a PPRL implementation that runs on the widespread available Hadoop-ecosystem. We considered multiple Spark-based PPRL implementations, including HLSH and multiple pivot-based implementations. With the aim to find the implementation that scales sufficiently to a big data scale (at least multiple millions of records). As a baseline for the benchmark we have used the open-source LSHDB.

From the performed experiments we have learned that HLSH, especially the Spark SQL variant, is able to outperform LSHDB in terms of runtime for multiple millions of records. However, at the cost of finding fewer true matches. The same principle goes for the pivot-based implementations. But all of these implementations perform worse than LSHDB, in terms of runtime and achieved accuracy. We can therefore conclude that, when performance is essential, creating custom tailored applications in Java or even C++ is advisable when going to run it on a single-node. Using Spark for such deployments, brings along too much overhead to the table.

This doesn't mean Spark is not a good option for PPRL applications. When there is a cluster available, even with 10 modest worker nodes, using Spark can greatly improve the runtime of PPRL applications. Even if single-node deployments do not take tens of hours or days to complete even for a few million records, there might be scenarios where matches have to be made as quick as possible. For example in streaming scenarios with a constant influx of records that have to be checked.

Furthermore, the proposed PPRL solution does not under-perform in terms of accuracy compared to regular linkage libraries. Combined with being implemented in Spark, makes it a viable and accessible option when considering PPRL. We believe this can take away the threshold for start using PPRL, because in contrast to the obscure and scare non-Spark implementations, the Hadoop-ecosystem and Spark have the benefit of having a community with expertise and is offered by cloud providers.

Appendix A

Micro-Benchmarks

A.1 Benchmark setup

The micro-benchmarks have been implemented using Scala 2.11 (JDK9) and has been executed on a modest laptop (2 CPU cores, 8 GB RAM). The *UsePopCountInstruction*[1] option has been enabled to accelerate the bitwise implementations. Similarly, a native BLAS[2] library has been included to accelerate implementations based on vector/matrix algebra.

A.2 Calculating Hamming distances

The results, averages of four runs (figure A.1 and A.2), show that iteration is the slowest option and bitwise the fastest option. Still, all implementations increase linearly with the amount of comparisons. The runtime of the bitwise implementation does not seem to increase when doubling the bit vector dimension, in contrast to iteration and vector algebra. This can be explained by the fact that both the 32-bit and 64-bit vectors fit in a single numerical data type (Scala's Integer and Long) and thus requires the same amount of operations. If the dimension would grows larger, an array of numerical values have to be used and a runtime increase can be expected. Nevertheless, the bitwise implementation is chosen to calculate Hamming distances. Therefore, RBFs must also be stored using numerical data types.

A.3 Generating Hamming LSH keys

The results (figure A.3 and A.4) indicate only a small sub-second performance gain from using the matrix multiplication implementation for the runs with 500,000 vectors (for both 32- and 64-bit long vectors). The performance gain is more significant for the other two runs, where the runtime is reduced with up to 68% compared to

[1] Allows the JVM to use the low-level `POPCNT` instruction to count the number of bits set to 1.
[2] Basic Linear Algebra Subprograms; implemented using low-level routines.

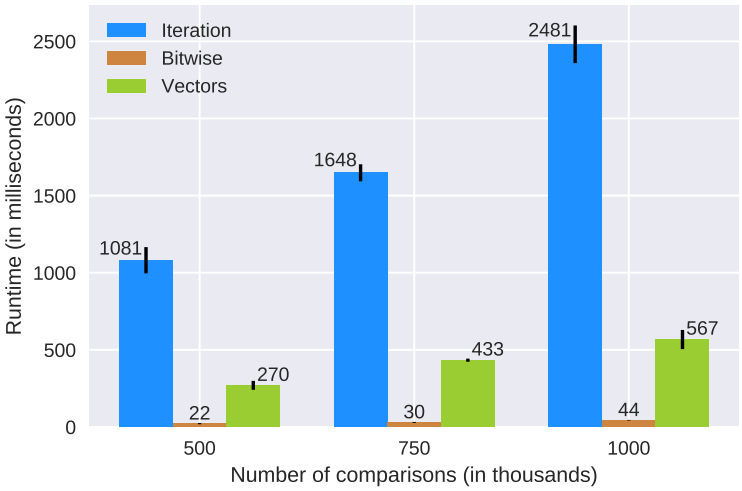

FIGURE A.1: Hamming distance calculations on 32-bit long vectors.

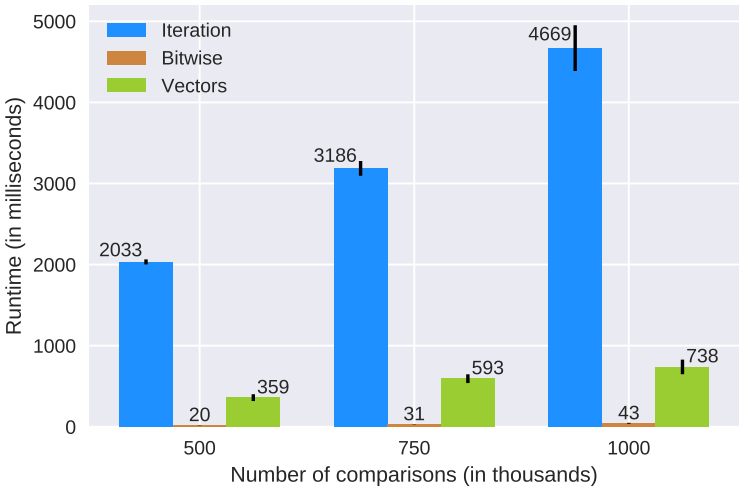

FIGURE A.2: Hamming distance calculations on 64-bit long vectors.

the iteration implementation. An explanation for the large discrepancy in runtime between the 500,000 and 1,000,000 runs for the iteration implementation might be the JVM's garbage collection mechanism kicking in when working with datasets that have a larger memory footprint. If this would be the case, it implies that the matrix multiplication implementation has a lower memory footprint. The matrix multiplication has been chosen as the to-use implementation.

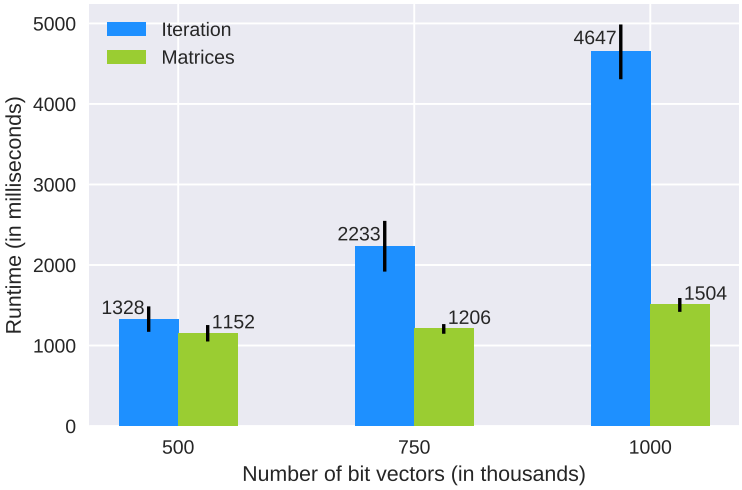

FIGURE A.3: Generating HLSH keys from 32-bit long vectors.

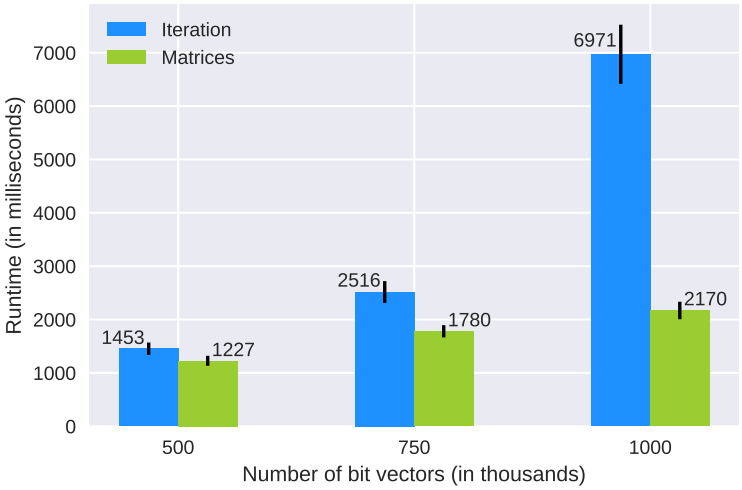

FIGURE A.4: Generating HLSH keys from 64-bit long vectors.

Bibliography

[1] Alexandr Andoni, Ilya Razenshteyn, and Negev Shekel Nosatzki. "LSH forest: Practical algorithms made theoretical". In: *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*. SIAM. 2017, pp. 67–78.

[2] Michael Armbrust et al. "Spark SQL: Relational data processing in spark". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, pp. 1383–1394.

[3] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. "LSH forest: self-tuning indexes for similarity search". In: *Proceedings of the 14th international conference on World Wide Web*. ACM. 2005, pp. 651–660.

[4] Rohan Baxter, Peter Christen, Tim Churches, et al. "A comparison of fast blocking methods for record linkage". In: *ACM SIGKDD*. Vol. 3. Citeseer. 2003, pp. 25–27.

[5] D J Baylis. *Error Correcting Codes: A Mathematical Introduction*. Vol. 15. CRC Press, 1997.

[6] Jon Louis Bentley. "Multidimensional binary search trees used for associative searching". In: *Communications of the ACM* 18.9 (1975), pp. 509–517.

[7] Erik Bernhardsson. "ANNOY: Approximate nearest neighbors in C++/Python optimized for memory usage and loading/saving to disk, 2013". In: *https://github.com/spotify/annoy* (2013).

[8] Burton H Bloom. "Space/time trade-offs in hash coding with allowable errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

[9] Sergey Brin. "Near neighbor search in large metric spaces". In: (1995).

[10] Peter Christen. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer Science & Business Media, 2012.

[11] Peter Christen et al. "Efficient cryptanalysis of Bloom filters for privacy-preserving record linkage". In: *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer. 2017, pp. 628–640.

[12] Thanh-Toan Do, Anh-Dzung Doan, and Ngai-Man Cheung. "Learning to hash with binary deep neural network". In: *European Conference on Computer Vision*. Springer. 2016, pp. 219–234.

[13] Elizabeth Ashley Durham. "A framework for accurate, efficient private record linkage". PhD thesis. Vanderbilt University Nashville, TN, 2012.

[14] Elizabeth A Durham et al. "Composite bloom filters for secure record linkage". In: *IEEE transactions on knowledge and data engineering* 26.12 (2014), pp. 2956–2968.

[15] Ivan P Fellegi and Alan B Sunter. "A theory for record linkage". In: *Journal of the American Statistical Association* 64.328 (1969), pp. 1183–1210.

[16] John Gantz and David Reinsel. "Extracting value from chaos". In: (2011).

[17] Marcel Gladbach et al. "Distributed Privacy-Preserving Record Linkage using Pivot-based Filter Techniques". In: *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*. IEEE. 2018.

[18] Lifang Gu and Rohan Baxter. "Decision models for record linkage". In: *Data mining*. Springer. 2006, pp. 146–160.

[19] Piotr Indyk and Rajeev Motwani. "Approximate nearest neighbors: towards removing the curse of dimensionality". In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM. 1998, pp. 604–613.

[20] Alexandros Karakasidis and Vassilios S Verykios. "Secure blocking + secure matching = secure record linkage". In: *Journal of Computing Science and Engineering* 5.3 (2011), pp. 223–235.

[21] Dimitrios Karapiperis, Aris Gkoulalas-Divanis, and Vassilios S Verykios. "LSHDB: a parallel and distributed engine for record linkage and similarity search". In: *Data Mining Workshops (ICDMW), 2016 IEEE 16th International Conference on*. IEEE. 2016, pp. 1–4.

[22] Dimitrios Karapiperis and Vassilios S Verykios. "A distributed framework for scaling up LSH-based computations in privacy preserving record linkage". In: *Proceedings of the 6th Balkan Conference in Informatics*. ACM. 2013, pp. 102–109.

[23] Dimitrios Karapiperis and Vassilios S Verykios. "A fast and efficient Hamming LSH-based scheme for accurate linkage". In: *Knowledge and Information Systems* 49.3 (2016), pp. 861–884.

[24] Dimitrios Karapiperis et al. "Efficient Record Linkage Using a Compact Hamming Space." In: *EDBT*. 2016, pp. 209–220.

[25] William J Krouse and Bart Elias. "Terrorist watchlist checks and air passenger prescreening". In: LIBRARY OF CONGRESS WASHINGTON DC CONGRESSIONAL RESEARCH SERVICE. 2009.

[26] Wai-Kong Lee et al. "Parallel and High Speed Hashing in GPU for Telemedicine Applications". In: *IEEE Access* (2018).

[27] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[28] Yehida Lindell. "Secure multiparty computation for privacy preserving data mining". In: *Encyclopedia of Data Warehousing and Mining*. IGI Global, 2005, pp. 1005–1009.

[29] Rui Mao et al. "Pivot selection for metric-space indexing". In: *International Journal of Machine Learning and Cybernetics* 7.2 (2016), pp. 311–323.

[30] John Nickolls and William J Dally. "The GPU computing era". In: *IEEE micro* 30.2 (2010).

[31] Mohammad Norouzi, Ali Punjani, and David J Fleet. "Fast exact search in hamming space with multi-index hashing". In: *IEEE transactions on pattern analysis and machine intelligence* 36.6 (2014), pp. 1107–1119.

[32] Stephen M Omohundro. *Five balltree construction algorithms*. International Computer Science Institute Berkeley, 1989.

[33] Satu Elisa Schaeffer. "Graph clustering". In: *Computer science review* 1.1 (2007), pp. 27–64.

[34] Andrew I Schein. "A generalized linear model for principal component analysis of binary data." In: 2003.

[35] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. "Privacy-preserving record linkage using Bloom filters". In: *BMC medical informatics and decision making* 9.1 (2009), p. 41.

[36] Ziad Sehili and Erhard Rahm. "Speeding up privacy preserving record linkage for metric space similarity measures". In: *Datenbank-Spektrum* 16.3 (2016), pp. 227–236.

[37] Konstantin Shvachko et al. "The hadoop distributed file system". In: *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. Ieee. 2010, pp. 1–10.

[38] Latanya Sweeney. "k-anonymity: A model for protecting privacy". In: *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems* 10.05 (2002), pp. 557–570.

[39] Khoi-Nguyen Tran, Dinusha Vatsalan, and Peter Christen. "GeCo: an online personal data generator and corruptor". In: *Proceedings of the 22nd ACM international conference on Information & Knowledge Management*. ACM. 2013, pp. 2473–2476.

[40] Dinusha Vatsalan and Peter Christen. "Privacy-preserving matching of similar patients". In: *Journal of biomedical informatics* 59 (2016), pp. 285–298.

[41] Dinusha Vatsalan, Peter Christen, and Vassilios S Verykios. "A taxonomy of privacy-preserving record linkage techniques". In: *Information Systems* 38.6 (2013), pp. 946–969.

[42] Dinusha Vatsalan et al. "An evaluation framework for privacy-preserving record linkage". In: *Journal of Privacy and Confidentiality* 6.1 (2014), p. 3.

[43] Dinusha Vatsalan et al. "Privacy-Preserving Record Linkage for Big Data: Current Approaches and Research Challenges". In: *Handbook of Big Data Technologies*. Springer, 2017, pp. 851–895.

[44] Vinod Kumar Vavilapalli et al. "Apache hadoop yarn: Yet another resource negotiator". In: *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM. 2013, p. 5.

[45] Jun Wang et al. "Learning to hash for indexing big data—a survey". In: *Proceedings of the IEEE* 104.1 (2016), pp. 34–57.

[46] William E Winkler. "Using the EM algorithm for weight computation in the Fellegi-Sunter model of record linkage". In: *Proceedings of the Section on Survey Research Methods, American Statistical Association*. Vol. 667. 1988, p. 671.

[47] Matei Zaharia et al. "Apache spark: a unified engine for big data processing". In: *Communications of the ACM* 59.11 (2016), pp. 56–65.