Privacy-Preserving Record Linkage with Spark

Onno Valkering
University of Amsterdam
Amsterdam, The Netherlands
onno.valkering@gmail.com

Adam Belloum
University of Amsterdam
Amsterdam, The Netherlands
a.s.z.belloum@uva.nl

***Abstract*—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 Apache Spark as an option to scale PPRL. Not only is it valuable to have a scalable PPRL implementation, but one based on the Spark would also be commonly deployable and could take advantage of further development of the ecosystem. Our results show that a PPRL solution based on Spark outperforms alternatives when it comes to handling multiple millions of records; can scale to dozens of nodes; and is on-par with regular record linkage implementations in terms of achieved results.**

***Index Terms*—linkage, PPRL, privacy, scalability, Spark**

I. Introduction

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 [13]. 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 applications. The goal of record linkage is to identify one and the same entities across multiple databases [7, 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 organizations. The only information that participating organization will learn is whether or not a certain record, i.e. an entity, is also in the database of another organization. From that point, optionally, the organizations can engage in targeted data exchange preserving the privacy of entities that are not shared among organizations. This is known as privacy-preserving record linkage (PPRL) [7, pp. 187-207].

A. Applications of PPRL

PPRL applications are typically found in the domains of crime and fraud detection, government services and healthcare [33]. An interesting example is outlined in [17]. This example describes 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 a linkage between personal data originating from airlines and hospitals. For this part PPRL can be used.

Another possible application [10] 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 [21]. Such, and similar, surveillance and screening can make use of PPRL to prevent privacy intrusions of those that are not on any watchlist.

PPRL can also be adapted to perform privacy-preserving similarity search (PPSS). In this case, an entity is no longer matched to a single other entity, but to multiple akin entities. This 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 patient groups [31].

B. The PPRL process

There are two major ways to perform PPRL, with a two-party protocol or three-party protocol [7, pp. 193-194]. When using a two-party protocol, the participating 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 than three-party protocols. This work only considers the more practicable three-party protocol PPRL. In particular the process [33] defined by the following six consecutive steps: parameter alignment, data pre-processing, indexing, comparison, classification and evaluation (figure 1).

1) Parameter alignment: Records are encoded so that original values of fields can't be recovered by other participants. Still, it must remain possible to perform meaningful comparisons of the encoded records 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 throughout the PPRL process. Determining and securely sharing all of these parameters is done in the first step.

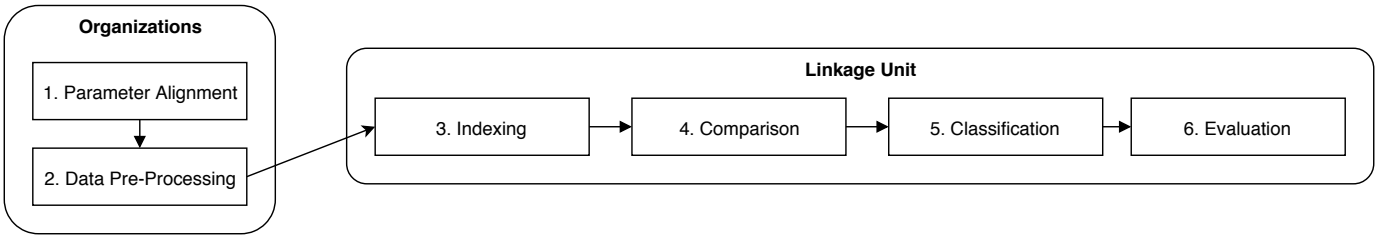

Fig. 1. Steps of the PPRL process (three-party protocol).

2) Data pre-processing: In this step, each organization encodes their 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 [29]. Before encoding, it is customary to first clean and format these QIDs in a uniform way. After encoding, the records are sent to the linkage unit.

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 grouped records the total number of comparisons that have to be performed can be reduced. This is a crucial step in terms of scalability, as the number of comparisons of a naive pairwise approach could be n^p in the worst case (where p is the number of participating organizations, each supplying n records).

4) Comparison: Through a 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.

5) Classification: Once it is known how (dis)similar two records are, a classification can be made: match or non-match. In contrast to regular record linkage, where machine learning is often applied [15], other decision models than a simple threshold function have not been thoroughly explored [33].

6) Evaluation: Metrics such as accuracy, precision, and recall can be used for evaluation. Measuring privacy itself is more difficult [32], and is considered out of scope in this work.

C. PPRL and Big Data

When applying PPRL on a Big Data scale and/or with an increasing amount of different databases, scalability becomes a challenge [33]. In recent papers [33, 28] Apache Hadoop and related tools and framework, combined considered an ecosystem, are suggested as an option for scaling PPRL. An important advantage of the Hadoop-ecosystem is its widespread adoption. As such, most public cloud providers, including Amazon Web Services[1] and Azure[2], offer fully managed Hadoop clusters. This allows organizations that don't possess the required infrastructure to still make use of the Hadoop-ecosystem, with a push of a button. This makes that a PPRL implementation based on the Hadoop-ecosystem will have the inherited advantage of being commonly deployable.

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

[2]https://azure.microsoft.com/solutions/hadoop/

It is known that Apache Spark, a prominent framework within the Hadoop-ecosystem, can be used to achieve great performance and scale to hundreds of nodes [35]. However, no clear picture exists how well it can handle PPRL [33]. With this work, we contribute to this missing evaluation. We present a scalable PPRL implementation based on Spark and show that it outperforms alternatives when it comes to handling millions of records; can scale to dozens of nodes; and is on-par with regular record linkage implementations in terms of achieved results. The implementation and datasets are publicly available under an open-source license[3]. Also, instructions and Docker images are provided to replicate the performed experiments[4].

The remainder of this paper is structured as follows. Related work is discussed in section II. Then, the theory underlying PPRL is described in more detail in sections III and IV. Followed by a description and evaluation of the created PPRL implementation in section V. Sections VI to VIII contain the discussion, future work and conclusions.

II. Related Work

A. Secure multi-party computation

With secure multi-party-computation (SMC) two or more parties engage, without an intermediary, in a joint effort solve computations [23]. The input 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 from different organizations. A drawback of SMC is the significant computational overhead [33], limiting its ability to be used for efficient large-scale applications, including PPRL.

B. LSHDB

LSHDB is an open-source[5] database and execution engine, developed using Java, supporting similarity search, regular record linkage and PPRL [18]. It supports batch processing, as well as online querying. To optimize query speed it relies on locality-sensitive hashing for indexing of records and has distributed and parallel capabilities. We have used LSHDB as a comparison to our Spark implementation, as it is one of the few, if not the only, existing open-source PPRL implementation. We used an optimized version (fork) for our evaluations[6].

[3]https://github.com/onnovalkering/pprl4s

[4]https://github.com/onnovalkering/pprl4s-extra

[5]https://github.com/dimkar121/lshdb

[6]https://github.com/onnovalkering/lshdb-star

III. Encoding

As mentioned in section I-B (The PPRL process), records have to be encoded before they are sent to the linkage unit. The Bloom filter data structure [6] has been adopted for this purpose [27]. Because of its effectiveness, good privacy protection and its relatively low computational costs, it has become a widely used encoding scheme for PPRL [7, 11, 17, 19, 28, 31, 33]. 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 [30]. This work makes use of two Bloom filters types: field-level Bloom filters (FBF) and record-level Bloom filters (RBF).

A. 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 [27]. First a bit vector v_{FBF} of length m_{FBF} is created. Next, the field f, padded with whitespace, is tokenized into the set T using *n*-grams of size two as tokens. Thus, for the value *hello*, T is $\{\_h, he, el, ll, lo, o\_\}$. Each token is then hashed using k independent hash functions. To reduce the success rate of dictionary attacks against FBFs, keyed cryptographic hash functions (e.g. HMAC) must be used [33]. 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. After this, v_{FBF} is the FBF for the field f.

Ideally, around 50% of the bits in a FBF are set to 1 (i.e. a uniform distribution). This maximizes entropy and thereby increases security in such a way that a malicious party might infer little about the length of the field's value and/or the value of the parameter k [11]. To accomplish this, the value of m_{FBF} can be dynamically determined to ensure that about half of the bits will remain 0. An equation for dynamically determining the value of m_{FBF} for FBFs of the same field, which is called dynamic FBF sizing, is provided in [11]:

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

This equation introduces two new variables: g and p. The first, g, denotes the average amount of tokens in T for the specific field (across all records). 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 2 illustrates the creation of a FBF.

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 [31]. Consider an integer value x, in this case, the tokens are the numbers in the range $[x - b, x + b]$, with interval b_{intv} (typically 1 for integers).

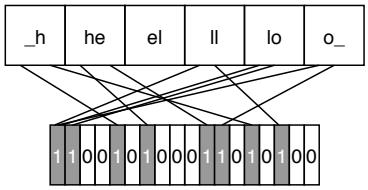

Fig. 2. Creation of a dynamic FBF ($g = 6$, $l = 18$, $k = 2$).

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 (all considering $b_{intv} = 1$) [31]. Tokenization of floating-point values is slightly different, as the method for integer values might result in differently aligned neighbors for such values. For example, take 5.5, with $b = 1$ and $b_{intv} = 0.5$, then the tokens will be $\{4.5, 5.0, 5.5, 6.0, 6.5\}$. The value 5.6 is very close to 5.5, however, the corresponding token set $\{4.6, 5.1, 5.6, 6.1, 6.6\}$ has no tokens in common with the token set of 5.5. As a result, the proximity of these two floating-point values will not be recognized during the comparison. As a solution, the neighboring numbers are based on x', which is determined for an arbitrary floating-point value x with the following function [31]:

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

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 is used to determine the neighboring numbers. Consider the floating-point value 5.6, with parameters $b = 1$ and $b_{intv} = 0.5$. Then $x' = 5.5$ and $T = \{4.5, 5.0, 5.6, 6.0, 6.5\}$.

B. Record-level Bloom filters

It is advisable to combine the FBFs of a record into a single RBF. Since the similarity of two records can be determined by a single comparison of the RBFs, instead of a pairwise comparison of each FBF. Also, it makes it more difficult to infer original field values and thereby improves the privacy protection [8].

The procedure for constructing a RBF is as follows [11]. 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. 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. The sample size for a FBF is its weight w_i times m_{RBF}. In case the i^{th} drawn bit is 1, then the bit in v_{RBF} with index i is also set to 1. As

the last step, the bits in v_{RBF} are randomly shuffled. The exact shuffling must be preserved between RBF constructions, across all the involved organizations. Otherwise, the RBFs loose their ability to be meaningfully compared. The weights of each field might be known in advance, or estimated in agreement by 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 [10, 12].

IV. Matching

After encoding, what remains for the linkage unit to operate on is exclusively a collection of RBFs. As described in section III (Encoding), 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 [16]. Since RBFs are essentially bit vectors, the applicable metric space can be defined as $\mathcal{H}^n = (\{0,1\}^n, d)$, where $n = m_{RBF}$. This corresponds to a Hamming space [5], therefore it makes sense to use the Hamming distance as the distance function d.

Finding matches within a Hamming space can be generalized to a k-nearest neighbour(s) (k-NN) [16] problem, where $k = 1$ for PPRL or $k \geq 2$ for similarity search (section I-A). The main characteristics that still set apart PPRL are:

- a Hamming space consisting of RBFs is typically high-dimensional, $n > 2000$ if not larger;
- few k-NN indexing structures are efficient for high-dimensional Hamming spaces [25];
- the nearest neighbour(s) must additionally be within some distance-threshold radius, instead of just being the closest.

A. Locality-sensitive hashing

To be able to efficiently search the collection of RBFs, i.e. $\mathcal{H}^n$, some preparation in the form of indexing is required. With PPRL, an indexing method to efficiently group similar RBFs (those in close proximity based on d) 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 [33]. To apply blocking to RBFs, blocking keys must be created based on RBFs in such a way that similar RBFs will have the same blocking key, and thus end up in the same block.

To create these blocking keys we use locality-sensitive hashing (LSH) [33]. 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 in close proximity within the metric space based on d [16]. This property makes the output of a LSH function suitable as blocking keys. 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.

B. Hamming LSH

A LSH technique suitable for Hamming spaces is Hamming LSH (HLSH) [10, 20]. In essence, HLSH is a function that samples bits from a given item [10]. 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 a Hamming space, with items X, based on HLSH is as follows [20]. We construct L number of hash tables, that each contain mappings from $x_i \in X$ 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. A technique to determine the optimal value of L is provided in [20].

Sampling k bits from a RBF, to create its blocking key(s), can be implemented in various ways. We now describe the used implementation based on matrix multiplication that we have found, through benchmarks[7], to be up to three times faster than an iteration-based implementation. Suppose we have two HLSH functions, h_1 and h_2, that sample the 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[8].

$$h_1(x_1) = 101 = 5$$

$$h_2(x_1) = 011 = 3$$

$$h_1(x_2) = 111 = 7$$

$$h_2(x_2) = 000 = 0$$

Bit sampling can also be implemented using matrix multiplication. For this, we need two matrices, $\mathbf{X}$ and $\mathbf{K}$. Where $\mathbf{X}$:

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

The matrix $\mathbf{K}$ in turn indicates column-wise which bits are to be sampled. To preserve to the order of sampled positions,

[7] A native BLAS library has been included to accelerate linear algebra computations, with the use of low-level routines.

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

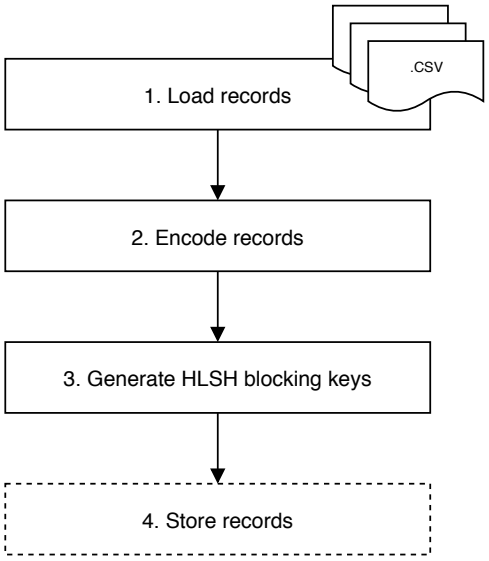

Fig. 3. The first phase of the Spark-based PPRL implementation.

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 as follows:

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

V. Implementation

We have created two Spark-based PPRL implementation variants. One using the resilient distributed dataset (RDD) API and the other using Spark SQL. The RDD API is a low-level API that provides extended control over performed operations. Spark SQL is a higher level API and allows Spark to perform optimizations using the Catalyst optimizer [2].

Both implementations consist of two phases. In the first phase (figure 3) the involved organizations load (and clean, if required) their records and encode them into RBFs. Also, for each RBF a set of L HLSH blocking keys is generated using the matrix multiplication method as described in section IV-B. Then, the output is stored on disk so that it can be transferred to the linkage unit[9]. This first phase corresponds to the second step of the PPRL process (Data Pre-Processing, section I-B).

The linkage unit performs the second phase (figure 4). The steps of this 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

[9]This step is omitted during benchmarks of the implementations.

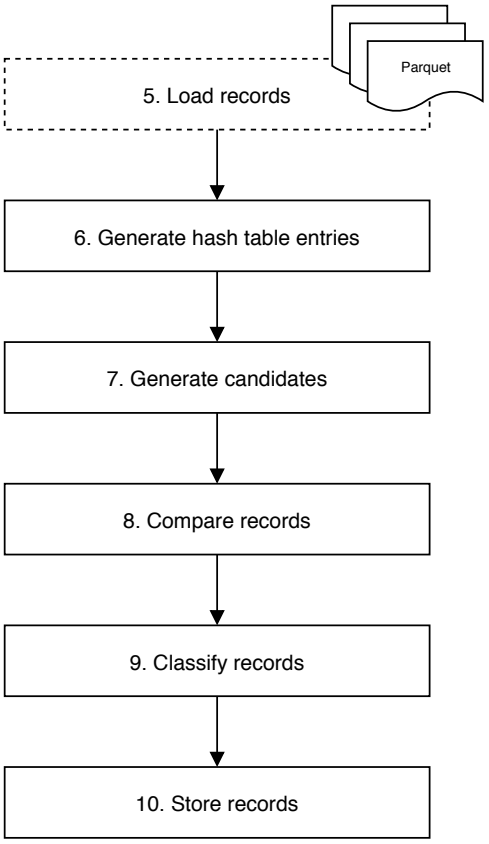

Fig. 4. The second phase of the Spark-based PPRL implementation.

steps. Before discussing the seventh step, let's first describe the preceding step that generates hash table entries.

The overarching idea is that we construct, in a distributed fashion, L hash tables. For this purpose, each record has a set of generated HLSH blocking keys ($k_1 \dots k_L$), where the ith key belongs exclusively to the ith hash table. In the sixth step, each record entry is duplicated into L separate (hash table) entries that each has a single HLSH blocking key associated to it. This approach allows Spark to process the entries independently and thereby the hash tables in parallel.

The goal of the next step (7) is to find record pairs that have at least one HLSH blocking key, with matching index, in common, i.e to find collisions in the L hash tables. Only these records will be considered for the future steps, and are therefore called candidate records. Based on the hash table entries, the RDD API implementation generates candidate record pairs by performing a cogroup on the HLSH blocking keys. This transformation results in a collection of groups (i.e. hash table), one for every distinct HLSH blocking key. All groups contain 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 Spark SQL implementation generates candidate record pairs differently based on the hash table entries. Namely, it performs an INNER JOIN based on the HLSH blocking keys. This results in the same candidate records pairs as the

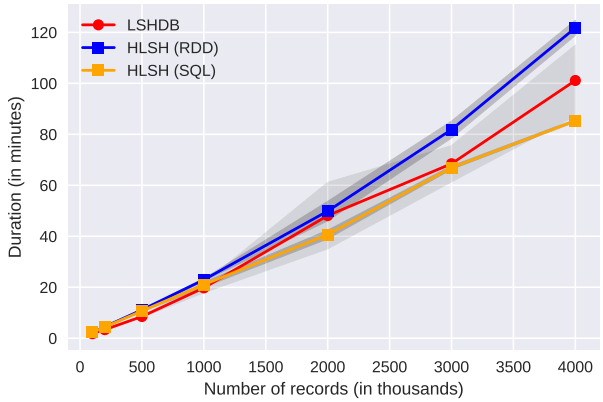

Fig. 5. Singe-node runtimes (records per database; two databases).

RDD API implementation, but they are not grouped by HLSH blocking keys and thus can be processed independently.

The remaining three steps (8-10) are very similar between the two implementations. Comparing records, i.e. calculating the Hamming distance, is done using optimized bitwise operations. Assigning classifications is done based on a threshold function, where the threshold is a parameter. Storing records is done using the Parquet format, but could be replaced by any other format compatible with Spark and Hadoop (HDFS).

A. Single-node evaluation

The two implementations variants have been benchmarked against LSHDB (section II-B), using a single-node setup[10] since LSHDB doesn't support cluster deployments. As the benchmark dataset we've used the North Carolina voting register (NCVR)[11]. From the NCVR dataset, different two-database configurations, of varying sizes, have been generated. Parts of the generated databases have been corrupted using GeCo to simulate dirty data [30]. The average of four runs is used as the final result for each configuration (figure 5).

What stands out is that LSHDB is faster than both the Spark variants 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 and other cluster-related tasks, even on a single-node setup. LSHDB is 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.

[10]Machine contained a Intel i7-6700, 32GB RAM and a SSD for storage.
[11]https://s3.amazonaws.com/dl.ncsbe.gov/data/list.html

TABLE I
EVALUATION METRICS FOR RL TOOLKIT, LSHDB AND SPARK.

Impl.	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
Spark	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
Spark	100,000	0.99	0.88	0.98	0.94

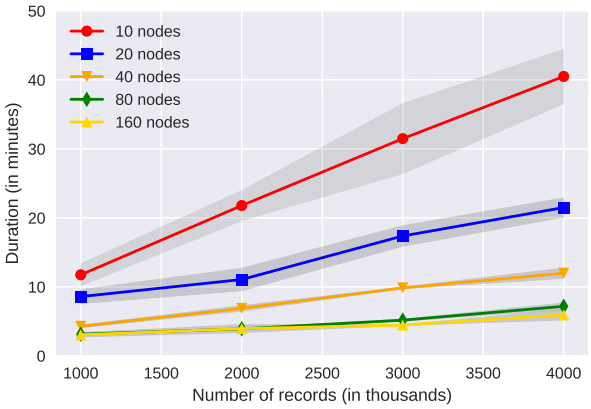

Fig. 6. RDD API cluster runtimes (records per database; two databases).

Between LSHDB and the Spark implementation variants, the precision, accuracy and F1-score are all on-par (table I). Except for recall. This means that although the Spark 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. Compared to the RL Toolkit[12], an open-source library for regular record linkage, both LSHDB and Spark are on-par in terms of the aforementioned evaluation metrics.

B. Cluster deployment evaluation

To measure how well the created Spark implementation variants can scale, they have been benchmarked after being deployed on a Hadoop cluster[13]. The same generated two-database configurations (NCVR) as before have been used. Different cluster sizes are considered, namely: 10, 20, 40, 80 and 160 worker nodes[14]. The averages of four runs for each configuration (number of records/nodes) for the RDD and Spark SQL implementation variants have been plotted in figure 6 and figure 7 respectively. Only runtime improvement is considered here, as it is assumed that the accuracy stays the same as in section V-A, since the computations are the same.

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-thirds for the largest 4m dataset. Doubling the number of nodes to

[12]https://github.com/J535D165/recordlinkage
[13]A Hortonworks Data Platform (HDP) v2.3.4 deployment.
[14]Each worker node (container) had 2 CPU cores and 8 GB RAM.

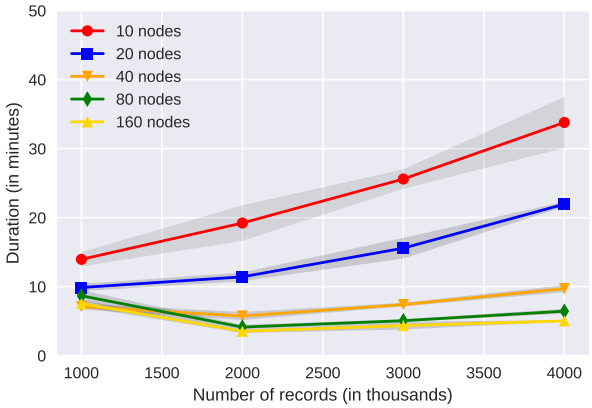

Fig. 7. Spark SQL cluster runtimes (records per database; two databases).

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 number 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 number 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 to none.

The Spark SQL implementation has a similar relation between runtime reduction and doubling the number 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 number 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 used as a black box.

For both implementations we don't observe a significant runtime improvement between the 80-nodes and 160-nodes clusters, despite doubling the number of nodes. We reason that this has to do with the used dataset being relatively small for these cluster sizes. When extrapolating these runtimes, we expect that the two lines will diverge, as the 20-node and 40-node lines, when using a sufficiently large dataset. In that case, it becomes beneficial again to use the larger amount of nodes.

VI. Discussion

A drawback of the HLSH approach is 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 through collision. 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 record to increase the probability of colliding with true matching records. Having multiple HLSH blocking keys blows-up, in terms of total amount of items that flow through the application, the dataset in the second phase of the Spark implementation, as duplicates are created so that hash tables can be processed in parallel. 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. However, these data-dependent indexing structures often rely heavily on iteration and recursion that are difficult to efficiently implement in Spark.

The runtime improvement over LSHDB in a single-node configuration is minimal. Still, in some scenarios even a runtime improvement of 15 minutes is advantageous. Think, for example, of streaming contexts with high throughput. However, Spark isn't meant for single-node deployments and comes out better when deployed on a cluster, as our results show. There even might be more runtime improvement to be gained by tuning the setup and parameters. The total runtime is also influenced by the dataset. For example, datasets that contain many similar but not equal entities will generate much more candidate record pairs than datasets where the majority is completely different and only a few entities are similar and/or equal. The first case will result in more comparisons that prolong the runtime, compared to the latter case.

VII. Future Work

This work considered a limited scope, in terms of tools, techniques, and implementations. In each of these areas lie opportunities for improvements. First, in this work only Spark has been used. However, the Hadoop-ecosystem consists of several other frameworks. An alternative, for example, is Apache Flink, as also suggested in recent other work [14, 33]. Flink provides a similar abstraction of computation as Spark, but focuses primarily on stream processing. This fits with use cases that have a real-time inbound flow of records, as, for example, in the border security application (section I-A).

Also, using a dedicated database to store records and/or intermediate values is worth exploring. In our implementations, all data is managed by Spark itself. However, it might be the case that certain data operations are performed more efficient when using an optimized big data store, such as Apache Cassandra[15] or Apache HBase[16].

[15]https://cassandra.apache.org
[16]https://hbase.apache.org

In term of techniques, the effectiveness of HLSH (section IV-B) depends on the parameters used in combination with the dataset. This can become an inconvenience when frequently dealing with new datasets. As each dataset will require manual parameter tuning. A new development are hash functions based on the Learning to Hash (L2H) principle [34]. Based on this principle the LSH functions most appropriate to the specific characteristics of the dataset can be learned, typically using deep learning [9]. Thus, by using L2H, manual parameter tuning can be avoided. In the same area, applying dimensionality reduction functions might be effective. Applying dimensionality reduction can speed up PPRL applications, as it will reduce the dimension of RBFs and thereby making the needed data operations to be able to run more efficient [16]. An applicable dimensionality reduction technique is Logistic Principal Component Analysis (LPCA) [26]

Implementation-wise there are also opportunities to explore. For instance, experimenting with native components (e.g. C/C++). The benefit of using native components is that operations can be performed more efficient, through low-level routines. This includes resorting to the graphical processing unit (GPU). Most of the required computations can be be defined as a vector- or matrix-operation. Such operations can be performed more efficient by utilizing the parallel nature of GPUs [24]. GPUs can also be used to accelerate the creation of RBFs, by performing the needed cryptographic hashing on GPUs instead of CPUs [22].

VIII. 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 adopted Hadoop-ecosystem. We considered two Spark-based implementations, with the aim to find an implementation that scales sufficiently to a big data scale. As a baseline, for the benchmark we have used the open-source LSHDB. From the performed single-node experiments, we have learned that a Spark-based implementation (Spark SQL) of HLSH is able to outperform LSHDB, in terms of total runtime, when processing multiple millions of records. However, it comes at the cost of finding fewer true matches. In terms of achieved accuracy, the two considered Spark implementations both perform slightly worse than LSHDB. However, the Spark-based implementations can be deployed on a cluster, which in turn reduces the runtime even further.

We can conclude that, when performance is essential, creating custom-tailored applications in Java or C++ is advisable when a single-node deployment is the only option. Using Spark for such deployments has non-negligible overhead. This doesn't mean Spark is not a good option for PPRL applications. When there is a cluster available, even with a few modest worker nodes, using Spark can greatly reduce the runtime of PPRL applications. Even if single-node deployments do not take tens of hours or days to complete processing millions of record, there might be scenarios where matches have to be made as fast as possible, justifying the use of a cluster. For example in streaming scenarios with an real-time inbound flow of records that have to be checked. Furthermore, both Spark implementations do not under-perform in terms of accuracy compared to regular linkage libraries. This makes it a viable and accessible option when considering PPRL. We believe this can take away the threshold to start using PPRL, because in contrast to the scarce non-Spark implementations, the Hadoop-ecosystem and Spark have the benefit of having a community with expert knowledge and is supported by cloud providers.

Acknowledgment

The authors would like to thank the EU PROCESS project (grant 777533), for supporting this work. The experiments in this work, have been made possible by SURFsara, that granted access to their Hathi Hadoop cluster for this purpose.

References

[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] Burton H Bloom. "Space/time Trade-offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426.

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

[8] 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.

[9] 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.

[10] Elizabeth Ashley Durham. "A Framework for Accurate, Efficient Private Record Linkage". PhD thesis. Vanderbilt University Nashville, TN, 2012.

[11] 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.

[12] 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.

[13] John Gantz and David Reinsel. “Extracting Value from Chaos”. In: (2011).

[14] 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.

[15] Lifang Gu and Rohan Baxter. “Decision Models for Record Linkage”. In: *Data mining*. Springer. 2006, pp. 146–160.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] William J Krouse and Bart Elias. “Terrorist Watchlist Checks and Air Passenger Prescreening”. In: Library of Congress Washington DC, Congressional Research Service. 2009.

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

[23] Yehida Lindell. “Secure Multiparty Computation for Privacy Preserving Data Mining”. In: *Encyclopedia of Data Warehousing and Mining*. IGI Global, 2005, pp. 1005–1009.

[24] John Nickolls and William J Dally. “The GPU Computing Era”. In: *IEEE micro* 30.2 (2010).

[25] 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.

[26] Andrew I Schein. “A Generalized Linear Model for Principal Component Analysis of Binary Data”. In: 2003.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] Dinusha Vatsalan and Peter Christen. “Privacy-Preserving Matching of Similar Patients”. In: *Journal of biomedical informatics* 59 (2016), pp. 285–298.

[32] Dinusha Vatsalan et al. “An Evaluation Framework for Privacy-Preserving Record Linkage”. In: *Journal of Privacy and Confidentiality* 6.1 (2014), p. 3.

[33] 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.

[34] Jun Wang et al. “Learning to Hash for Indexing Big Data”. In: *Proceedings of the IEEE* 104.1 (2016), pp. 34–57.

[35] Matei Zaharia et al. “Apache Spark: A Unified Engine for Big Data Processing”. In: *Communications of the ACM* 59.11 (2016), pp. 56–65.