PSH: A Probabilistic Signature Hash Method with Hash Neighborhood Candidate Generation for Fast Edit-Distance String Comparison on Big Data

Joseph Jupin
CIS Dept.
Temple University
Philadelphia, PA, USA
joejupin@temple.edu

Justin Y. Shi
CIS Dept.
Temple University
Philadelphia, PA, USA
shi@temple.edu

Eduard C. Dragut
CIS Dept.
Temple University
Philadelphia, PA, USA
edragut@temple.edu

***Abstract*— Approximate string matching is essential because data entry errors are unavoidable. Approximately 80% of data entry errors are a single edit distance from the correct entry. We introduce Probabilistic Signature Hashing (PSH), a hash-based filter and verify method to enhance the performance of edit distance comparison of relatively short strings with proven no loss to accuracy. Our experiments show that the proposed method is almost 6800 orders of magnitude faster than Damerau-Levenshtein (DL) edit distance and produces the same exact results. This method combines prefix pruning, string bit signatures and hashing to provide very fast edit-distance comparison with no loss of true positive matches. PSH will provide substantial performance gains as a string comparison metric when used in place of DL.**

Keywords- Approximate String Comparison, Edit Distance, Deduplication, Record Linkage

I. INTRODUCTION

Combining data from different sources and/or deduplication are important first steps in the knowledge discovery process [1], whether data mining or statistical analysis of data is to be performed. The primary motivation for this research is to develop a faster method to compare relatively short strings as are typically found in demographic data. Our current project is to develop a specialized Record Linkage (RL) method for an urban health department. RL is a process that compares pairs of records from heterogeneous databases to find records that refer to the same entities [2]. Whether an RL system uses a deterministic or probabilistic [3] methodology, it is necessary to compare all the data values within each candidate pair of records.

The department needs to match client records across 11 independent health and social sciences databases without a reliable unique identifier. There are 1.5 million clients and 50 million records. Some of the clients have been in the system since birth. The system has to link records that span the clients' lives. The department currently uses a proprietary deterministic RL method but has experienced high false positive and false negative rates. We added the Damerau-Levenshtein (DL) edit distance algorithm to their method, which increased true positive matches but also increased the runtime by 500%. The data has to be updated daily, which currently requires approximately 8 hours per night, when it is not being queried for client matches. It would take approximately 40 hours to run the algorithm with DL. Each daily update would take more than a day. We could not use blocking methods to increase matching performance due to their dependence on blocking keys [4, 5, 6, 7, 8, 9, 10] selected from fields that have significant missing, erroneous or inconsistent data.

This paper's primary contribution is the development of an improved composite method, called the Probabilistic Signature Hash (PSH), which substantially decreases the computation required to compare short alphabetic, numeric and alphanumeric strings prior to evaluation with an edit distance metric.

II. BACKGROUND

Some of the methods used to optimize edit-distance are filtering, neighborhood generation [12], pruning [13], hashing [14] and tries [15]. We include descriptions of the Damerau-Levenshtein edit distance, Prefix Pruning—a performance enhancement for DL (PDL) and a description of the length filter. We also discuss Trie-Join [15], which is one of the fastest optimizations for edit-distance.

A. Damerau-Levenshtein Edit Distance

For effective string difference measurement, edit distance has an advantage over other metrics because it considers the ordering of characters and allows nontrivial alignment [16]. The Levenshtein edit distance algorithm is a dynamic programming solution for calculating the minimum number of character substitutions, insertions and deletions to convert one word into another [17]. Damerau extended Levenshtein distance to also detect transposition errors and treat them as one edit operation. Approximately 80% of data entry errors can be corrected using a one character substitution, one character deletion, one character insertion or a transposition of two characters [18]. The main problem with the DL algorithm is its complexity: $O(mn)$, where m and n are the lengths of the compared strings. This can require significant computation for comparisons of very large datasets—even if the compared strings are relatively short.

B. Prefix Pruning

Once a pair of strings has been determined to be sufficiently different for an edit distance measurement, the calculation should be terminated. A user-defined threshold can be added to decrease the computation required for DL. For a threshold k, it is only necessary to compute the elements from $j = i - k$ to $j = i + k$ (see algorithm below), which reduces the search space to a $2k + 1$ wide strip on the

diagonal [19]. The threshold can be used to force an early termination if $d_{i,*} > k$ [20]. The implementation, called *Pruning-Damerau-Levenshtein* (PDL), below is similar to an edit distance Prefix Pruning method for Trie-based string similarity joins [21]. There is a counter variable x added to count the $d_{i,j} \leq k$. If $x \leq 0$, for row i, the function terminates and returns a Boolean value *FALSE*. This decreases the complexity of DL from $O(mn)$ to $O(kl)$, where l is the length of the shortest string. If PDL completes and $d(m,n) \leq k$, the function returns $TRUE$, otherwise it returns $FALSE$. There are two lines that assign elements in the distance matrix d to 1000. This imposes a border of arbitrarily large integers just outside the $2k+1$ strip, ensuring the selection of a correct minimum value in the line $d_{i,j} = min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) + 1$ because d is initially zeros. The algorithm for PDL is:

Algorithm 1: PDL(s, t, k)

Input: s, t: strings of characters
k: integer threshold
Output: Boolean
$d = |s| + 1 \times |t| + 1$: array of integer zeros
m, n, x: integers
Begin
Step 1: Check for empty strings
$m = |s|$
$n = |t|$
if $m = 0$: **return** $FALSE$ **end-if**
if $n = 0$: **return** $FALSE$ **end-if**
Step 2: Create distance matrix d
for $i = 0$ *to* m: $d_{i,0} = i$ **end-for**
for $j = 1$ *to* n: $d_{0,j} = j$ **end-for**
Step 3: Calculate distance matrix
for $i = 1$ *to* m
$x = 0$
if $i < k + 1$: $d_{i,i-k-1} = 1000$
for $j = max(i-k, 1)$ *to* $min(i+k, n)$
if $s_{i-1} = t_{j-1}$
$d_{i,j} = d_{i-1,j-1}$
else
$d_{i,j} = min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) + 1$
if $i > 1$ *and* $j > 1$
if $s_{i-1} = t_{j-2}$ *and* $s_{i-2} = t_{j-1}$
$d_{i,j} = min(d_{i,j}, d_{i-2,j-2} + 1)$
end-if
end-if
end-if
if $d_{i,j} \leq k$: x+= 1 **end-if**
end-for
if $j < n$: $d_{i,j} = 1000$
if $x <= 0$: **return** $FALSE$ **end-if**
end-for
if $d_{m,n} <= k$: **return** $TRUE$
else: **return** $FALSE$
end-if
end

C. Filter and Verify Methods

In the '90s, methods to decrease data comparison using edit distance called "filter and verify" methods were introduced. Research on these methods is still active as filters can potentially discard many expensive comparisons [22]. One of the most current methods, length filtering [23], is based on the fact that if two strings *s* and *t* differ by within *k* or less edits, the difference in their lengths cannot be greater than *k*. Consider that "Joe" and "Jose"; and "Jose" and "Josef" are approximate matches for *k* = 1 but "Joe" and "Josef" are not. It should be noted that length filtering will typically not work on fixed-length strings, such as phone numbers and Social Security Numbers.

D. Trie-Join

Special purpose approximate string matching tree structures, referred to as "tries", have been used to facilitate DL type searching. In [13], the authors use trie structures with edit distance constraints to compress data and compute string similarity more efficiently. Their experimentation shows that their algorithm is much more efficient for shorter strings with low edit distances of 1 or 2 edits. The experiments included Ed-Join and some of the previous methods upon which Ed-Join [16] is based. The Trie-Join method combined the concept of the prefix pruning and a tree structure to index strings that approximately match from a dictionary, list or data store.

III. METHODOLOGY

PSH is a composite optimization that combines the PDL and length filter with signature filtering and signature hashing. We describe our method in this section.

A. Signature Filtering

Bitwise signature filtering has two parts: generating bitwise signature and filtering the signatures. The method takes advantage of CPU's ability to perform logical and arithmetic operations on unsigned integers very quickly. The main idea is that the signature is compressed into 32-bit unsigned integers, which have sufficient capacity to contain a checklist of numeric, alphabetic and alphanumeric characters in bits. The signature m for string s is a checklist of a subset of characters in s, where bit $m_0 = 1$ iff $'A' \in s$ and bit $m_1 = 1$ iff $'B' \in s$, etc. 32 bits is large enough to store all characters in the alphabet (A to Z) once that occur in a string and all numbers (0 to 9) that occur from 1 to 3 times. These do require some storage for the signatures for each string but these signatures can be created very fast and only require 4 bytes of memory. The unused bits are used to store additional information about the frequency of characters in a string. We describe the process for alphanumeric street addresses below. We omit the methods for numeric only and alphabetic character strings due to space constraints.

B. Generating FBF Signatures

Address strings contain both characters and numbers and present a problem because integers are limited to 32-bits and the $\Sigma = \{0,1,2,3,4,5,6,7,8,9,A,B,C,\ldots,X,Y,Z\}$ contains 36 characters. Our statistical analysis of addresses found that the characters 'S' and 'T' occurred in over 90% of addresses and the characters 'J', 'Q', 'X', and 'Z' occurred very infrequently. The 'S' and 'T' were removed because they were not a good measure of variance, 'J' and 'Q' were

combined to one bit position and likewise with 'X', and 'Z'. The figure below shows the address signature for "1801 N BROAD ST".

```
98765432 10YXWVUR PONMLKJI HGFEDCBA
         Z        Q
01000000 11000001 01100000 00001011
```

Figure 1: Address signature for "1801 N BROAD ST"

Algorithm 2: SetAddrBits(s)

Input: s: string of characters
Output: x: 32-bit unsigned integer
$\Sigma = \begin{Bmatrix} A, B, C, D, E, F, G, H, I, JQ, K, L, M, N, O, \\ P, R, U, V, W, XZ, Y, 0,1,2,3,4,5,6,7,8,9 \end{Bmatrix}$
(Note that JQ and XZ are treated as same character and ST are ignored)
c: integer to set signature bit
x: integer to contain signature
begin
  for each $s_i \in s$
    if $s_i \in \Sigma$
      $c = c|\Sigma_c = s_i$
      $x = x \vee (1 \ll c)$
    end-if
  end-for
  return x
end

C. Filtering FBF Signatures

The different characters between strings s and t can be found by using the exclusive disjunction on their signatures m and n, respectively. Algorithm FindDiffBits(m, n) uses a fast bit counting method to count the ones in the exclusive disjunction result of m and n. The loop only executes as many times as there are ones in the string [24]. The address strings are alphanumeric and the maximum length in our large list of real standardized local addresses is 25 characters. Because of the relatively short length of these strings, the FindDiffBits(m, n) algorithm produces sparse bit vectors for real word addresses. The loop will only execute x times for each of the x ones in the integers' exclusive disjunction's bits, which represents x members in a set, and is denoted as $x = |m \oplus n|$. The algorithm for FindDiffBits(m, n) is:

Algorithm 3: FindDiffBits(m, n)

Input: m: 32-bit vector signature for string s
  n: 32-bit vector signature for string t
Output: x: 32-bit integer count of different bits
d: 32-bit unsigned integer
begin
  $i = 0$
  $d = m_i \oplus n_i$
  $x = 0$
  while $d > 0$
    x+= 1
    $d = d \wedge (d - 1)$
  end-while
end

This method has two significant performance properties: it compresses the signatures into compact primitive data types, which means faster loading to CPU registers and storing from registers, and allows machine-level operations to be used to process the primitive data very quickly, which means much faster processing.

D. Naïve Signature Hashing

We could hash on the bitwise signatures but that would require the storage of 4G buckets, which is a waste of RAM. To save memory, we mapped the 32-bit signatures to 16-bits using the naïve mapping shown in the example below by combining two signature bits to one hash code bit. This was fast but many buckets were empty and some buckets had too many entries, which had a negative effect on performance.

```
8642 0XVR OMKI GECA
9753 1YWU PNLJ HFDB
     Z    Q
1000 1001 1100 0011
```

Figure 2: Naïve hash mapping for "1801 N BROAD ST"

E. Probabilistic Signature Hashing

To smooth the buckets in the hash table, we take a probabilistic approach to the hash mapping of signatures. A comprehensive description of the steps of the algorithm is not possible due to space constraints. We applied our algorithm to a list of 547,771 addresses in the City of Philadelphia, USA. The first step is to count the co-occurrence of character pairs for all strings in the data set. The figure below shows that the characters 'A' and 'B' occur together in 338,213 addresses.

AB	AC	AD		78	79	89
338213	350587	380903	...	241567	232658	225143

Figure 3: Counts of co-occurrence of characters in data

The second step is to sort the count by frequency. Notice that two identified infrequent characters, 'J' and 'X' are the minimum value.

JX	FX	FJ		E1	NR	EN
24759	54709	56896	...	476181	480206	485825

Figure 4: Sorted counts of co-occurrence of characters in data

Next, we iteratively select values closest to the mean without duplicating any characters until all characters in Σ are included. The figure below shows the selections. The count for "59" was closest to the mean. The count for "W4" was next, etc. These character groupings are used as the mapping for the hash function.

59	W4	LV	MO	B3	67	G0	DI
278361	279420	276807	276235	275005	272173	270427	286442
H8	J2	RX	AF	Y1	KN	EP	CU
260221	299857	303564	326097	338785	351388	352392	170173

Figure 5: Co-occurrence mapping for hash function

The mapping for addresses is shown below. This probabilistic hash method is approximately twice as fast as the naïve method. Note that the algorithm actually uses bit positions in signatures. We substituted characters because it

is easier to explain the process. The hash buckets contain the index of strings from the data set.

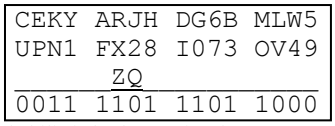

Figure 6: Probabilistic hash mapping for "1801 N BROAD ST"

F. Hash Neighborhood Generation for Candidate Buckets

It is not necessary to scan all 64K buckets to find hash candidates. A function using bitwise operators can be used to calculate the bucket codes that may contain approximate matches for a query exemplar. The algorithm below shows the calculation of hash candidates for codes that differ for up to two bits from the hash code exemplar h.

Algorithm 3: GetHashCodes(h)
Input: h: A hash code to neighborhood match
Output: H: The hash neighborhood for h
$b1$: Bit field for differs by 1 bit
$b2$: Bit field for differs by 2 bits
b: Number of bits in hash code (16 bits)
i, j: Iterators
k: Counter
Begin
  $i = 0$
  $k = 0$
  $H_{k++} = h$
  $b1 = 0000000000000001$
  $b2 = 0000000000000010$
  while $i < b$
    $H_{k++} = h \oplus b1$
    $j = i + 1$
    while $j < b$
      $H_{k++} = h \oplus (b1 \vee b2)$
      $b2 = b2 \ll 1$
      $j = j + 1$
    end-while
    $b1 = b1 \ll 1$
    $b2 = b1 \ll 1$
    $i = i + 1$
  end-while
  return H
end

IV. Sketch of Proof of Correctness

We show that the bitwise signature filter, hash filter and length filter do not discard any true positive matches from their respective candidate sets. Because none of these methods remove any true positives, their composite does not produce any false negative results.

A. Signature Filter

We define an approximate match as "strings s and t differ by k or less edits" where k is a user-defined threshold. Implementing this in PDL forces termination once a magnitude of distance less than or equal to k is no longer possible. To process two strings s and t, we create bitwise signatures m, n as: $m = \text{SetAddrBits}(s)$ and $n = \text{SetAddrBits}(t)$. There is a relation between PDL and bitwise signature comparison, $\text{FindDiffBits}(m, n)$, that the set of all string pairs $\langle s, t\rangle \in S \times T$, where S and T are lists of strings, with a $\text{FindDiffBits}(m, n) \leq 2k$, contains all string pairs that will pass PDL for a maximum of k edits, where $\text{PDL}(s, t, k) = TRUE$. In other words, if we select a set of pairs $G_{\leq 2k} = \forall(s, t) \in S \times T,\ \text{FindDiffBits}(m, n) \leq 2k$, where m is the signature of s and n is a signature of t, and set $H_{\leq k} = \forall(s, t) \in S \times T, \text{PDL}(s, t, k) = TRUE$, then we claim $G_{\leq 2k} \supseteq H_{\leq k}$.

Consider that PDL returns a Boolean value that is *TRUE* if the number substitutions, deletions, insertions and transpositions to convert string s into t is less than or equal to k edits and that $\text{FindDiffBits}(m, n) = |m \oplus n|$.

If a single edit operation found for a pair $(s, t) \in S \times T$ is a transposition, the filter will show a difference of zero because $|m \oplus n| = 0, \forall s_i \in s, \exists s_i \in t$ and $\forall t_j \in t, \exists t_j \in s$. Let s = "1**32**45" and t = "1**23**45". Since s and t have the same characters, $|m \oplus n| = 0$.

If a single edit operation is a delete, the worst case is $|m \oplus n| = 1$ if the member to be deleted $s_i \in s$ such that $s_i \notin t$. Let s = "12345**6**" and t = "12345". Since s and t differ by one delete operation, they differ by one character **6**, $|m \oplus n| = 1$.

If the single edit is an insertion, the worst case is $|m \oplus n| = 1$ if the character to be inserted $t_j \in t$ into s such that $t_j \notin s$. Let s = "1234" and t = "1234**5**". Since s and t differ by one insert operation, they differ by one character **5**, $|m \oplus n| = 1$.

If the single edit is a substitution, the worst case is $|m \oplus n| = 2$ if the member substituted s_i is changed to t_j such that $s_i \in s$, but $s_i \notin t$ and $t_j \in t$ but $t_j \notin s$. Let s = "1234**6**" and t = "1234**5**". Since s and t differ by one substitution operation, each differs from the other by one character. Notice that **5** is in t but not in s and **6** is in s but not in t, and we have $|m \oplus n| = 2$.

The worst case for a PDL of k is $|m \oplus n| = 2k$ if all k edits are substitutions and result in the worst-case condition above for each of the substitutions then $\forall h \in H_{\leq k}, \exists g \in G_{\leq 2k}, h = g$ and $\text{PDL}(s, t, k) = TRUE \Longrightarrow \text{FindDiffBits}(m, n) \leq 2k$. If k is increased, this same property holds because, by inductive reasoning, in the worst case for k edits, the signature will differ by at most $2k$ bits due to a substitution error.

B. Hash Filter

We claim that for $\forall s \in S$ with signature $m = SetAddrBits(s)$, $get\_hash\_codes(h)$ produces a set of candidate buckets H that contain the entries in S that are within k edit operations from s. This is evident because the hash codes are generated by a disjunction of two bits in the signature, which creates a dilation that cover all related signatures that is guaranteed to contain the correct candidates.

C. Length Filter

The length filter will not remove true matches from its candidate list because for k or less edits, the test $abs(|s| - |t|) \leq k$ will pass all string pairs that differ by $\leq k$ edits.

Substitution and transposition errors do not change the length of a string. If there are $\leq k$ insertions or deletions, the difference in string length will be $\leq k$.

D. PSH Composite

PSH combines signature hashing, length filtering, signature filtering and prefix pruning to increase the speed of DL edit-distance comparison. The algorithm below shows the application of filters. When a failing condition is found, the **if** statement terminates and continues to the next candidate in the hash bucket list.

Algorithm 4: PSH(S, M, T, q)

Input: S: A list of strings to search
M: A list of bit signatures for S
T: A hash table containing references to S and M
q: A query string
Output: $Match$: A list of matches
$m \in M, n$: Bit signatures
H: Hash neighborhood
$h \in H, y$: Hash codes
$T_h \in T$: Hash buckets with chaining for S and M
$s \in S$: A string
Begin
  $n = SetAddrBits(q)$
  $y = hash(n)$
  $H = GetHashCodes(y)$
  for each $h \in H$
    for each $s \in T_h$
      if $(L(s,q) \leq k \wedge F(m,n) \leq 2k \wedge PDL(s,q,k))$
        $Match \leftarrow (s,q)$
      end-if
    end-for
  end-for
  return *x*
end

where T is a hash table that contains buckets containing references to strings in S and signatures in M. T_h Are the candidate buckets that have $2k$ or less bits differing from hash code h.

V. Experiments

The data for the experiments was the 547,771 Philadelphia address data mentioned above. There were two datasets used in experiments. One dataset was clean address data and the other file had random single edit errors injected into the data from the clean file. The experiment matched all strings in the clean set with the error set. Since the strings in the files are known to match by index, this establishes a lower limit for a ground truth to detect false negative matches. There will be matches in the data that have different indices too. The hash candidate experiments are run 5 times and their average is recorded as the result. The other methods are only run once. These methods do not use hash candidate generation. Without the benefit of hashing each of these methods would have to compare over 300 billion string pairs. The string comparators used in the experiments include:

- Damerau-Levenshtein edit-distance (DL)
- Bitwise Signature Filtered PDL (FPDL)
- Length then Bitwise Signature Filtered PDL (LFPDL)
- Hybrid Signature Neighborhood Hashed Length and Fast Filtered PDL (GLFPDL)
- Probabilistic Signature Hash (PGLFPDL)

VI. Results

The experiments in the table below contains all 547,771 Philadelphia address strings per input file. DL requires more than 26.8 days to complete. Each pairwise comparison takes approximately 7.72 microseconds. The method using the bitwise filter and PDL (FPDL) takes approximately 48.7 thousand seconds (13.5 hours). The method with the length filter added (LFPDL) takes 23 thousand seconds (6.4 hours). The naïve hashing method (GLFPDL) takes 606 seconds (10 minutes). PSH (PGLFPDL) only takes 342 seconds (less than 6 minutes) to complete with each pairwise comparison taking 1.14 nanoseconds (given the search space), which is almost 6,800 times faster than DL. Notice that there are no false negative (Type 2) errors and that the Type 1 column are not actually false positives. They match by comparison but not by index. All optimizations produce the same results as DL.

Table 1: Experiments on all 547,771 addresses with PGLFPDL probabilistic hashing

Ad	Type1	Type2	Time ms	Speedup	Pair Time
DL	995,992	0	2,317,457,638.00	1	7.72E-03
FPDL	995,992	0	48,742,836.00	47.54	1.62E-04
LFPDL	995,992	0	23,027,915.00	100.64	7.67E-05
GLFPDL	995,992	0	606,182.00	3,823.04	2.02E-06
PGLFPDL	995,992	0	341,693.00	6,782.28	1.14E-06

A. Analysis of Hash Buckets

An analysis of the first naïve hash method found most hash buckets empty. The effects of this method are shown in the table below. The AdCle column is the hash table analysis for the naïve method and that there are 46,129 empty buckets and a max bucket size of 6,513. The PSH method decreases the empty buckets in the AdCleP column by 18,457 to 27,672 and the max bucket size by 5,429 to 1,084 entries. The last 2 columns are for the error injected street addresses, which have more variance in their data due to the random errors, resulting in less variance in individual hash buckets. Smaller bucket sizes mean less local string comparison and better performance.

Table 2: Bucket statistics for addresses

	AdCle	AdCleP	AdErr	AdErrP
Zero	46129	27672	27403	14658
Max	6513	1084	4725	853
Mean	8.358322	8.358322	8.358322	8.358322
StDev	56.08403	23.3084	43.01646	19.11679

B. Comparison to Trie-Join

The experiments in Table 3 below compare the new method to a more recently published novel string comparison optimization called Trie Join and a variant Bi Trie Join [21].

The software for the Trie Join methods was obtained from the authors project Website. The experiments were run five times for each method and their average runtime in milliseconds is shown below. The Type1 and Type2 error columns are calculated differently for these experiments. The ground truth is the *DL* edit distance algorithm's string matching results for a single edit. Notice that the Trie methods have Type2 errors. The Trie Join method was able to find 87.5% and Bi Trie Join found 90.9% of the strings that *DL* found. There are no false positives for any of the three methods. Trie Join is approximately 3.28 times faster and Bi Trie Join is approximately 2.62 times faster than *PGLFPDL*. The peak working set memory for each method is shown in the last column in kilobytes. Trie Join required 3.57 times the RAM and Bi Trie Join required 6.26 times the RAM that was required by *PGLFPDL*. The amount of memory used for an optimization is an important consideration for big data problems.

Table 3: Comparison of PGLFPDL and Trie Join with all 547,771 addresses

Ad	Type1	Type2	Time ms	Speedup	Pair Time	Memory K	Memory R
PGLFPDL	0	0	216889.4	1.00	7.23E-07	76,900	1.00
TJ	0	188029	82660.4	2.62	2.75E-07	274,600	3.57
BTJ	0	136610	66162.8	3.28	2.21E-07	481,604	6.26

VII. CONCLUSION

Our goal was to reduce unnecessary computation by identifying and filtering record pairs that are guaranteed not to match. Our computational results clearly show that there are superior performance gains while using the PSH over DL. The results also show that there is no loss to accuracy—PSH produces the same results as DL. PSH takes advantage of a computer's ability to quickly compare differences in primitive data types using a single instruction, exclusive disjunction, and an enhanced while loop to count ones. The computational cost of creating the hash table and signatures is very low (all experiments included the time to generate signatures and hash tables) and only 4 bytes are needed for each signature. Considering that addresses can be at least 25 characters (or 50 for 16-bit Unicode) bytes and a phone number is 10 characters (or 20 bytes), the space complexity is very efficient given the performance benefits. The record pair search space for the PSH is as exhaustive as DL but it completes each comparison, on average, much quicker by eliminating unnecessary work when comparing string pairs by filtering out pairs with no chance of matching. PSH delivers the same matches as DL, in significantly less time. Our results provide evidence that PSH can perform the same work in less than 6 minutes that would take DL nearly 27 days to complete. The 40 hour-a-day update for the RL project mentioned in the Introduction may now be completed in less than an hour. If this method maintains this level of performance, it should be able to compare two lists containing 10M addresses or 1 trillion pairs in 1.6 days and produce the same exact results as DL. Addresses are the longest string typically found in demographic data and require the most time to match using DL.

ACKNOLEDGEMENTS

We thank the City of Philadelphia's Office of Health and Opportunity for grant support for this project and access to a real-life record linkage problem. This work was supported in part by the U.S. NSF grant BIGDATA 1546480.

REFERENCES

[1] Fayyad, U., Piatetsky-Shapiro, G., Smyth, P.: Data Mining to Knowledge Discovery in Databases, In: AI Magazine 17(3) (1996)

[2] Dunn, H. L.: Record Linkage, In: American Journal of Public Health 36 (12): pp. 1412–1416. (1946)

[3] Fellegi, I., Sunter, A.: A Theory for Record Linkage, In: Journal of the American Statistical Association 64 (328): pp. 1183–1210. (1969)

[4] Jaro, M. A.: Advances in Record Linkage Methodology as Applied to Matching the 1985 Census of Tampa, Florida. In: Journal of the American Statistical Society, 84(406):414–420, (1989)

[5] Hernandez, M., Stolfo, S.: Real-world data is dirty: data cleansing and the merge/purge problem, In: Journal of Data Mining and Knowledge Discovery, 1(2), (1998)

[6] Christen, P., Churches T.: Febrl: Freely extensible biomedical record linkage: Manual, release 0.2 edition (2003)

[7] McCallum, A., Nigam, K., Ungar, L.: Efficient clustering of high-dimensional data sets, In: KDD , pp. 169–178 (2000)

[8] Cohen, W., Richman, J.: Learning to Match and Cluster Large High-Dimensional Data Sets for Data Integration. In: SIGKDD'02, (2002)

[9] George Papadakis, Jonathan Svirsky, Avigdor Gal, and Themis Palpanas. 2016. Comparative analysis of approximate blocking techniques for entity resolution. In PVLDB 9, 9 (May 2016), 684-695.

[10] Gu L., Baxter, R.: "Adaptive Filtering for Efficient Record Linkage", In SIAM 2004

[11] Campbell, K. M., Deck, D., Krupski, A.: Record Linkage Software in the Public Domain: A Comparison of Link Plus, The Link King, and a "Basic" Deterministic Algorithm, In: Health Informatics Journal, Vol. 14(1) (2008)

[12] Myers, E.: "A sublinear algorithm for approximate keyword searching", In Algorithmica, 12(4/5) pp. 345-374, 1994

[13] Wang, J; Feng, J; Li, G. "TrieJoin: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints". In PVLDB, Vol. 3, No. 1 (2010)

[14] Knuth, Donald E. (1997). The Art of Computer Programming: Volume 3, Sorting and Searching, 2nd ed.. Addison-Wesley.

[15] Apostolico, A.; Galil, Z.: "Combinatorial Algorithms on Words", Springer-Vertag, 1985.

[16] Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. In PVLDB, 1(1):933–944 (2008)

[17] Levenshtein, V. I.: Binary codes capable of correcting deletions, insertions, and reversals. In Soviet Physics Doklady, (1966)

[18] Damerau F. J.: A technique for computer detection and correction of spelling errors, In: Communications of the ACM, (1964)

[19] Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology, Cambridge, UK: Cambridge University Press, ISBN 0-521-58519-8 (1997)

[20] Sakoe, H., Chiba, S.: Dynamic programming algorithm optimization for spoken word recognition, pp. 159–165, (1990)

[21] Wang, J., Feng, J., Li, G.: Trie-Join: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints, In: PVLDB, 3, 1 2010

[22] Navarro, G.: A guided tour to approximate string matching", In CSUR, Volume 33 Issue 1, Pages 31 – 88, March 2001

[23] Gravano, L., Ipeirotis, H., "Approximate string joins in a database (almost) for free", In VLDB, pages 491 – 500, 2001

[24] Wegner, P.: A technique for counting ones in a binary computer, In: Communications of the ACM 3 (5): 322 (1960)