Understanding Cloud Data Using Approximate String Matching and Edit Distance

Joseph Jupin, Justin Y. Shi, Zoran Obradovic
CIS Dept., Temple University
1801 N. Broad Street,
Philadelphia, PA 19122 USA
011-1-215-204-8450
{joejupin,shi}@temple.edu,zoran@ist.temple.edu

ABSTRACT

For health and human services, fraud detection and other security services, identity resolution is a core requirement for understanding big data in the cloud. Due to the lack of a globally unique identifier and captured typographic differences for the same identity, identity resolution has high spatial and temporal complexities. We propose a filter and verify method to substantially increase the speed of approximate string matching using edit distance. This method has been found to be almost 80 times faster (130 times when combined with other optimizations) than Damerau-Levenshtein edit distance and preserves all approximate matches. Our method creates compressed signatures for data fields and uses Boolean operations and an enhanced bit counter to quickly compare the distance between the fields. This method is intended to be applied to data records whose fields contain relatively short-length strings, such as those found in most demographic data. Without loss of accuracy, the proposed Fast Bitwise Filter will provide substantial performance gain to approximate string comparison in database, record linkage and deduplication data processing systems.

1 INTRODUCTION

The primary motivation for this research is to develop a faster method to compare relatively short strings as are typically found in demographic data collected in the cloud. This paper reports the findings in developing a distributed and cloud-based *Record Linkage* (RL) system for an urban health service department. Record Linkage is a process that compares pairs of records from heterogeneous databases to find similar or identical entities [1]. Whether the RL system uses a deterministic or probabilistic [2] methodology, it is necessary to compare the data within each 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 point and threshold RL method using a combination of the Soundex [3] for names, exact matches for gender, addresses and phone numbers, and other proprietary linear complexity approximate string matching algorithms for birthdates and Social Security Numbers in their record comparator but has experienced high false positive and false negative rates. We cannot provide the actual algorithm, real data or results based on real data due to a confidentiality agreement with the client and HIPPA requirements.

Studies have determined that the Soundex is highly prone to both low sensitivity (as low as 40%) and low precision (as low as 33%) [4][5]. We replaced the Soundex with the *Damerau-Levenshtein* (DL) edit distance algorithm, which increased true positive matches by more than 46% but also increased the runtime by 500%. The data has to be updated daily, which currently requires approximately 8 hours per night, when the system is not being queried for client matches. It would take approximately 40 hours to run the algorithm with DL. The system would not be able to keep up with updates and would also harm performance for client match queries.

Traditional blocking methods pre-compute candidate pairs of records and can decrease the accuracy of RL due to sensitivity to errors and inconsistencies in the data, thus ignoring potential matches and increasing the number of false negatives. Traditional blocking methods include standard blocking [7], sorted neighborhood [8], bigram indexing [9] and canopy clustering with tf-idf [10][11]. In [12] an analysis of each method is described. It has been noted that, in practice, sets of records based on blocking keys are usually very large [27]. Blocking on the last name "Smith" could create a very large block because it is the most common name in the U.S. Census.

In the reported study, data fields available for identity resolution include: First Name, Last Name, Address, Phone Number, Gender, Social Security Number and Birth Date. Although many databases have universal identifiers, like the Social Security Number, discrepancies from data entry errors, inconsistent data values and missing data fields can substantially depreciate its value to link entities. Data entry errors can include substitution, deletion, insertion and transposition of characters. There is a significant amount of missing and inconsistent data. As many as 32 % of Social Security Numbers have been reported missing from health related databases [13]. More than 40 % of SSNs are missing from our data. If the data fields selected as blocking keys contain missing, inconsistent or erroneous data, the records will not assigned to the correct blocks. All of the data fields have missing, inconsistent and erroneous data. This method is not offered as a replacement for blocking. In fact, it may increase performance in systems that both block and use our filter as a wrapper for the DL edit distance function.

The primary contribution of this paper is the development of an improved "filter and verify" method, called the *Fast Bitwise Filter* (FBF), which substantially decreases the computation required to compare short alphabetic, numeric and alphanumeric

978-0-7695-4956-9/13 $26.00 © 2013 IEEE
DOI 10.1109/SC.Companion.2012.149

strings for demographic data fields prior to evaluation with an edit distance metric. The FBF compresses the signature of a record's fields into 32-bit integers and allows the use of fast Boolean operations to quickly compare the string signatures. We combine FBF with an existing method, called length filtering to further increase computational efficiency of approximate string matching using the edit distance metric. This method substantially decreases the number of pair-wise comparisons using more computationally expensive DL edit distance by quickly disqualifying fields that are guaranteed not to match.

This paper is organized as follows. We discuss string distance metrics methods in the Background section. We describe FBF in the Methodology section. The Proof of Correctness section contains a proof that the FBF optimization has no false negatives with respect to DL. The Experiments section explains how experiments were performed and the data. The Results section discusses the computational results for both string and record experiments.

2 BACKGROUND

We discuss some traditional string comparison metrics below. In [14], the authors determined that token-based methods do not perform well for this type of data. Hence, we do not include token-based methods in our background or experiments. We include descriptions of the Damerau-Levenshtein edit distance, Prefix Pruning—a performance enhancement for DL, the Jaro [7] string similarity metric and the Jaro-Winkler [15] string similarity metric—an enhancement to Jaro.

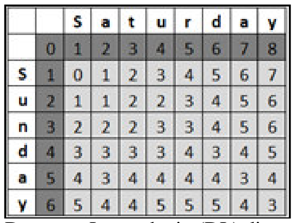
2.1 Damerau-Levenshtein Edit Distance

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. For example, the Levenshtein distance between the words "Saturday" and "Sunday" is 3 because the 'a' and 't' can be deleted and the 'r' can be substituted with an 'n' to convert "Saturday" to "Sunday".

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 [17]. The main problem with the DL algorithm is its complexity: $O(mn)$, where m and n are the lengths of the compared strings, s and t. This can require significant computation for comparisons of very large datasets—even if the compared strings are relatively short. The algorithm below is DL:

```
Algorithm 1: DL$(s, t)$
Input: s, t: strings of characters
Output: $d_{m,n}$: integer
$d = |s| + 1 \times |t| + 1$: array of integer zeros
$m, n$: integers
Begin
  Step 1: Check for empty strings
  $m = |s|$
  $n = |t|$
  if $m = 0$: return $n$ end-if
  if $n = 0$: return $m$ 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 d
  for $i = 1$ to $m$
    for $j = 1$ to $n$
      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
    end-for
  end-for
  return $d_{m,n}$
end
```

Fig. 1 contains the DL matrix. The dark grey cells show the initial cell numbers for rows and columns and the light grey area is where the edit distance is calculated for standard DL. The intersections of character columns and rows in the light grey area are the optimal calculated distances for all substrings. For example the distance between "Sat" and "Sun" is 2 because the intersection at 't' and 'n' is 2, which is the number of edits necessary to convert one to the other.



		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	3	4	5	6	7
u	2	1	1	2	2	3	4	5	6
n	3	2	2	2	3	3	4	5	6
d	4	3	3	3	3	4	3	4	5
a	5	4	3	4	4	4	4	3	4
y	6	5	4	4	5	5	5	4	3

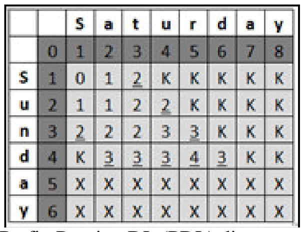
Figure 1. Damerau-Levenshtein (DL) distance matrix

2.2 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 in Alg. 1 Step 3 from $j = i - k$ to $j = i + k$, which reduces the search to a diagonal $2k + 1$ wide strip on the diagonal [18]. The threshold can be used to force an early termination if $d_{i,*} > k$ [19]. The implementation, called *Pruning-Damerau-Levenshtein* (PDL), below is similar to an edit distance *Prefix Pruning* method for Trie-based string similarity joins [20]. 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 is to impose 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 differs from the algorithm in [20] because it has been modified to perform DL instead of the standard Levenshtein distance and it returns a Boolean value:

```
**Algorithm 2:** 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 and string lengths*
  $m = |s|$
  $n = |t|$
  **if** $m = 0$: **return** $FALSE$ **end-if**
  **if** $n = 0$: **return** $FALSE$ **end-if**
  **if** $abs(m - n) > k$: **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** $d_{i,j} \le k$: $x += 1$ **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**
```

The matrix for PDL is in Fig. 2. The 15 'K' and 16 'X' cells are the computational savings for PDL with $k = 2$, a difference of 2 or less characters. PDL only required 17 cell evaluations of the 48 required by DL. For $k = 1$, PDL would terminate immediately because $abs(|s| - |t|) > k$. If this restriction was removed, only 8 cell evaluations would be required because the underlined cells would be skipped.



		S	a	t	u	r	d	a	y
	0	1	2	3	4	5	6	7	8
S	1	0	1	2	K	K	K	K	K
u	2	1	1	2	2	K	K	K	K
n	3	2	2	2	3	3	K	K	K
d	4	K	3	3	3	4	3	K	K
a	5	X	X	X	X	X	X	X	X
y	6	X	X	X	X	X	X	X	X

Figure 2. Prefix Pruning DL (PDL) distance matrix

2.3 Jaro Similarity Metric

Instead of increasing with the degree of difference between two strings, s and t, as is done in edit distance algorithms, the *Jaro* similarity metric [5] increases on the interval $[0, 1]$ for strings that have more characters in common within an n character position search width. In other words matching characters in strings s and t can be no more than n positions apart, where:

$$n = \left\lfloor \frac{max(|s|, |t|)}{2} \right\rfloor - 1$$

The Jaro similarity metric is calculated as:

$$jaro(s, t) = \frac{1}{3}\left(\frac{m}{|s|} + \frac{m}{|t|} + \frac{m - \frac{r}{2}}{m}\right)$$

where m is the number of matching characters and r is the number of transposed characters. An example: for $s = SMITH$ and $t = SMIHT$,

$$n = \left\lfloor \frac{5}{2} \right\rfloor - 1 = 1$$

$$m = 5$$

$$r = 1$$

$$jaro(s, t) = \frac{1}{3}\left(\frac{5}{|s|} + \frac{5}{|t|} + \frac{5 - \frac{1}{2}}{5}\right) = 0.967$$

The Jaro score for $s = SMITH$ and $t = JONES$ would be 0.0 because the *S*'s in the strings are more than one character apart.

2.4 Jaro-Winkler Similarity Metric

Winkler enhanced Jaro by awarding higher scores to strings that have longer matching prefixes. The Jaro-Winkler [15] string similarity metric is calculated:

$$wink(s, t) = jaro(s, t) + \ell p(1 - jaro(s, t))$$

where $jaro(s, t)$ is the Jaro score, ℓ is the length of the matching prefix and p is a scaling factor. Again for $s = SMITH$ and $t = SMIHT$ and $p = 0.1$,

$$wink(s, t) = 0.967 + 3 \times 0.1 \times (1 - 0.967) = 0.977$$

2.5 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 very active and filters have the potential to discard a very large number of more comprehensive and expensive comparisons [28]. One of the most common methods, *Length Filtering* [29], is based on the fact that if two strings *s* and *t* differ by *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 is obvious that length filtering will not work on fixed-length strings, such as phone numbers, Social Security Numbers and many other typed of IDs. The length filter from [29] is defined as shown in Algorithm 3:

```
**Algorithm 3:** LengthFilter$(s, t)$
**Input**: $s, t$: strings of characters
**Output**: Boolean
**Begin**
  **if** $abs((|s| - |t|)) > k$: **return** *FALSE*
  **else**: **return** *TRUE*
  **end-if**
```

```
end
```

3 METHODOLOGY

Our proposed optimization has two parts: generating FBF signatures and filtering the signatures. The FBF method takes advantage of a computer's ability to perform logical and arithmetic operations on unsigned integers very quickly [21]. The idea is that the filter signature is compressed into 32-bit unsigned integers, which have sufficient capacity to contain a checklist of numeric and alphabetic characters in bits as shown below in Fig. 3 and Fig. 4.

The signature x for string s is actually a checklist of a subset of characters in s, where bit $x_0 = 1$ iff $'\mathrm{A}' \in s$ and bit $x_1 = 1$ iff $'\mathrm{B}' \in s$, 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 1 to 3 times in a string as shown in the Figures below. These do require some storage for the signatures for each string but these signatures can be created very quickly. The unused bits can store extra information about the string (e.g. "Does any character in the string occur more than 2 times for an alphabetic string?" Or "Are 2 of the same character juxtaposed?")

```
ABCDEFGH IJKLMNOP QRSTUVWX YZ------
00000001 10001000 00110000 00000000
```

Figure 3. 32-bit alphabetic FBF bit signature for "SMITH"

```
00011122 23334445 55666777 888999--
11011011 00000001 11000000 10000000
```

Figure 4. 32-bit numeric FBF bit signature for "8005551212"

For SSNs, birthdates and phone numbers, one 32-bit integer should be sufficient for an FBF signature. This is because, like edit distance, FBF measures the difference between strings. If one of the compared numeric strings has many repeated characters, say a phone number "213-333-3333", the signature will only record three of the 3s. If the other compared number has *n* less 3s, than there are *n* different characters that will be revealed when the signatures are compared as described in Alg. 6. The FBF difference between "213-333-3333" and "213-333-4444", would be 3 because three of the 4s would be recorded. An FBF comparison is a fast approximation for edit distance. An FBF signature only need contain a subset of the characters in its string to be effective. To record more than one occurrence of each character in alphabetic strings, just add integers. A two integer vector can record 2 occurrences of alphabetic characters.

3.1 Generating FBF Signatures

Algorithm 4, $\mathrm{SetAlphaBits}(s)$, shows the process for generating an l-length vector of signatures to count l or less occurrences of each character for a string *s* containing only alphabetic characters. The algorithm, as implemented in code, should ignore case and will ensure an accurate count of all $\Sigma_c(\ldots\Sigma_c)^j \in s$, where $\Sigma = \{\mathrm{A,B,C,\ldots,X,Y,Z}\}$. Upon completion of $\mathrm{SetAlphaBits}(s)$, the following condition is true: $x_{j\ bit\ c} = 1 \Leftrightarrow \Sigma_c(\ldots\Sigma_c)^j \in s, c \in \mathbb{Z}|0 \le c < |\Sigma|,\ j \in \mathbb{Z}|0 \le j < l$. Note that $\ll$ below represents the bit shift operator.

```
Algorithm 4: SetAlphaBits(s, l)
Input: s: string of characters
       l: length of array x
Output: x: vector of 32-bit unsigned integers
Σ = {A, B, C, ..., X, Y, Z}
j: |Σ| integer vector of zeros
begin
  for each $s_i \in s$
    if $s_i \in \Sigma$
      $c = c|\Sigma_c = s_i$
      if $j_c < l$
        $x_j = x_j \vee (1 \ll c)$
      end-if
      $j_c$ += 1
    end-if
  end-for
end
```

Algorithm 5, $\mathrm{SetNumBits}(s)$, shows the process of generating a signature for a numeric only string, such as a Social Security Number, phone number or birth date. It uses a 32-bit unsigned integer, which can count up to three instances of numbers in the string s using 30 bits and ignores non-numeric characters. Since the strings are relatively short, only one integer is used for the signature. This algorithm can be modified to count more than 3 occurrences as is done in Alg. 4 above. For $\Sigma = \{0,1,2,\ldots,7,8,9\}$, the following condition is true upon completion of $\mathrm{SetNumBits}(s)$: $x_{bit\ 3c+j} = 1 \Leftrightarrow \Sigma_c(\ldots\Sigma_c)^j \in s, c \in \mathbb{Z}|0 \le c < |\Sigma|, j \in \mathbb{Z}|0 \le j < 3$.

```
Algorithm 5: SetNumBits(s)
Input: s: string of characters
Output: x: 32-bit unsigned integer
Σ = {0,1,2,3,4,5,6,7,8,9}
j: |Σ| integer vector of zeros
begin
  for each $s_i \in s$
    if $s_i \in \Sigma$
      $c = c|\Sigma_c = s_i$
      if $j_c = 2$
        $x = x \vee (4 \ll (3 * c))$
      else-if $j_c = 1$
        $x = x \vee (2 \ll (3 * c))$
      else-if $j_c = 0$
        $x = x \vee (1 \ll (3 * c))$
      end-if
      $j_c$ += 1
    end-if
  end-for
end
```

3.2 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 $\mathrm{FindDiffBits}(m,n,l)$ 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 [22]. The longest last names in the Census data are 15 characters. The longest numeric string is the phone number, which has 10 characters. The address strings are alphanumeric and the maximum length in a list of real standardized local addresses is 25 characters. Because of the relatively short length of these strings, they will most likely produce sparse bit vectors from the $\mathrm{FindDiffBits}(m,n,l)$ algorithm, described below. 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 $\mathrm{FindDiffBits}(m,n,l)$ is:

```
Algorithm 6: FindDiffBits($m, n, l$)
Input: $m$: 32-bit vector signature for string $s$
       $n$: 32-bit vector signature for string $t$
       $l$: 32-bit integer length of $m$ and $n$ vectors
Output: $x$: 32-bit integer count of different bits
$d$: 32-bit unsigned integer
begin
  $i = 0$
  $x = 0$
  while $i < l$
    $d = m_i \oplus n_i$
    while $d > 0$
      $x += 1$
      $d = d \wedge (d - 1)$
    end-while
    $i += 1$
  end-while
end
```

The length variable l is set to 1 for numeric signatures as defined in Alg. 4. Both the alphabetic and numeric signature and filter methods can be combined to process alphanumeric fields. This method has two significant performance properties: it compresses the signatures into compact primitive data types, which means faster loading to 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.

4 PROOF OF CORRECTNESS

We define an approximate match as "strings s and t differ by k or less edits" where k is a user-defined variable. Implementing this in PDL forces termination once a magnitude of distance less than or equal to k is no longer possible. To process numeric strings s and t, we create FBF signatures m, n as: $m = \text{SetNumBits}(s)$ and $n = \text{SetNumBits}(t)$. There is a relation between PDL and FBF signature comparison, $\text{FindDiffBits}(m, n)$, that the set of all string pairs $\langle s, t \rangle \in S \times T$ with a $\text{FindDiffBits}(m, n, l) \leq 2k$, 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 sets of pairs $G_{\leq 2k} = \forall (s, t) \in S \times T$, $\text{FindDiffBits}(m, n, l) \leq 2k$, where m is the signature of s and n is a signature of t, and $H_{\leq k} = \forall (s, t) \in S \times T, \text{PDL}(s, t, k) = TRUE$, 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 $\text{FindDiffBits}(m, n, l) = |m \oplus n|$.

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 $|m \oplus n| = 0$ since $\forall s_i \in s, \exists s_i \in t$ 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 $|m \oplus n| = 2$.

This is also valid for strings that contain multiples of the same character because each new occurrence of a character that is already in a string is recorded as a new character. Consider s = "12345**6**" and t = "12345**66**". The second 6 is considered different than the first and sets the "found a second 6 bit" to 1. Since there is no second '6' in s, we have $t_j \in t$ such that $t_j \notin s$ because the second '6' is considered different than the first.

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 $\forall h(s, t, k) \in H_{\leq k}, \exists g(m, n, l) \in G_{\leq 2k}$ and $\text{PDL}(s, t, k) = TRUE \Longrightarrow \text{FindDiffBits}(m, n, l) \leq 2k$.

Algorithm 7 below shows an example of an approximate string similarity join on the lists S and T using the FBF and PDL together as a *Filtered and Pruned Damerau-Levenshtein* distance (FPDL). Each string in the lists has a single numeric field. $\text{FindDiffBits}(m, n, l)$ uses signatures m and n for strings s and t, respectively, to decide which pairs to perform the computationally more expensive $\text{PDL}(s, t, k)$ using the aforementioned threshold relationship. To process numeric strings, create FBF signature arrays M, N as: $\forall s \in S, \exists m \in M, m_i = \text{SetNumBits}(s_i)$ and $\forall t \in T, \exists n \in N, n_j = \text{SetNumBits}(t_j)$ and follow the process in $\text{MatchStrings}(S, T, M, N, k, l)$:

```
Algorithm 7: MatchStrings($S, T, M, N, k, l$)
Input: $S, T$: Strings lists to be matched
       $M, N$: FBF signature lists for $S$ and $T$
       $k$: Integer threshold
       $l$: Integer length of signatures
$i, j$: Integers
begin
  $\forall (s, t) \in S \times T$
    if FindDiffBits$(m_i, n_j, l) \leq 2k$
      if PDL$(s_i, t_j, k) = TRUE$
        $match(s_i, t_j)$
      else
        $unmatch(s_i, t_j)$
    else
      $unmatch(s_i, t_j)$
    end-if
  end-if
end
```

The filter can also be used to determine the *unmatch* condition as shown in the second else statement above for FPDL because it is already established that the strings can't match since $G_{\leq 2k} \supseteq H_{\leq k}$ and $G_{\leq 2k} \cap G_{>2k} = \emptyset$, where $G_{>2k} = \forall (s, t) \in S \times T, |m \oplus n| > 2k$, then $H_{\leq k} \cap G_{>2k} = \emptyset$, $\forall h(s, t, k) \in H_{\leq k}, \nexists g(m, n, l) \in G_{>2k}$ and $\text{FindDiffBits}(m, n, l) > 2k \Longrightarrow \text{PDL}(s, t, k) \neq TRUE$. This alphabetic and alphanumeric FBF proofs are similar to this proof because the signature bits are generated and processed in the same way. The Venn diagram in Fig. 5 below shows a typical relation between the four sets: $G_{\leq 2k}$, $G_{>2k}$, $H_{\leq k}$ and $H_{>k} = \forall (s, t) \in S \times T, \text{PDL}(s, t, k) \neq$

$TRUE$. The large black circle is $H_{>k}$ and the small black circle is $G_{\leq 2k}$. The intersection is the $(s,t) \in S \times T$ in which $\text{FindDiffBits}(m,n,l) \leq 2k$ but $\text{PDL}(s,t,k) \neq TRUE$. It may also be the case that $G_{\leq 2k} \cap H_{>k} = \emptyset$ if $\text{FindDiffBits}(m,n,l)$ and $\text{PDL}(s,t,k)$ are in perfect agreement.

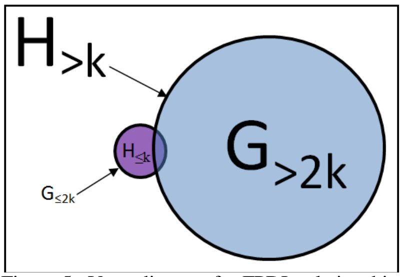


Figure 5. Venn diagram for FPDL relationship

5 EXPERIMENTS

The experiments include only comparable string distance metrics, meaning that the compared methods are likely to be effective on short strings. The primary use of this method is intended for linking fields in the previously described demographic type data (and possibly other relatively short strings). According to the 1990 US Census data obtained from [23], the minimum, maximum and average character lengths for first names are: 2, 11 and 5.96, respectively. These statistics are 2, 15 and 6.89 for last names. The birth date, Social Security number and phone number fields are fixed-length at 8, 9 and 10 characters, respectively. The string comparators used in the experiments include:

1. Damerau-Levenshtein (DL)
2. Prefix Pruned Damerau-Levenshtein (PDL)
3. Jaro string similarity (Jaro)
4. Jaro-Winkler string similarity (Wink)
5. Hamming Distance (Ham) [24]
6. FBF Filtered DL (FDL)
7. FBF Filtered PDL (FPDL)
8. FBF Filtered only (FBF)
9. Length Filtered DL (LDL)
10. Length Filtered PDL (LPDL)
11. Length Filter only (LF)
12. Length then FBF Filtered DL (LFDL)
13. Length then FBF Filtered PDL (LFPDL)
14. Length then FBF only (LFBF)

The experiments were run 5 times and their average was recorded as the result. The data for the experiments included randomly selected strings from:

1. 5,163 Census first names (FN)
2. 151,670 Census last names (LN)
3. 547,771 local addresses (Ad)
4. 12,000 synthetic phone numbers (Ph)
5. 35,525 random birthdates (Bi)
6. 12,000 synthetic Social Security Numbers (SSN)

The first names were merged from the male and female first names lists from the 1990 U.S. Census. The last names were from the 2000 U.S. Census. The addresses were from local tax records containing 3,874 unique streets. The phone numbers were synthetically generated based on the numbering scheme of the North American Numbering Plan (NANP) [25]. The birthdates were randomly selected over 100 years between 2/25/1912 and 2/24/2012 or 36,525 unique dates. The Social Security Numbers were synthetically generated by the same rules that the Social Security Administration uses to issue actual SSNs to clients [26]. Each entry in the initial or "clean" data sets were injected with single edit errors to produce a second "error" data set to simulate data entry errors, where the clean entries match the error entries by index position in each list to maintain a ground truth. Samples of 5,000 were selected from each list and matched using the string comparison algorithms.

Our second set of results generates runtime curves for all of the methods from the first experiment. When executed on two same-size lists, the FBF algorithm is clearly $O(n^2)$ as are all of the others but the work for each string pair comparison is, on average, significantly reduced. We ran experiments using a variable n from 1000 to 18000 strings in each of the two datasets to be merged from the 151,671 last names from the Census data. We randomly selected 5 clean datasets for each n (1000 to 18000 last names or 90 clean datasets) and created 90 error datasets by injecting single edit distance error to a set copied from the clean data. We ran each experiment 5 times, discarding the fastest and slowest times from each and averaging the remaining times.

Our third set of experiments perform the same experiments as the second set but using the length filter from [29] and the combination of length filtering with FBF using the Census last names. The length filter was used as a wrapper for FBF as FBF is used as a wrapper for DL and PDL as shown in Alg. 7, basically adding another if-else statement. We also ran the experiments for the first set of experiments with the local last names, addresses and the first names. We did not perform any length-based filter experiments on the fixed-length numeric strings (birthdates, SSNs and phone numbers) because, as mentioned, the length filter is useless for fixed-length data strings.

The test computer is an Acer Aspire 7745G notebook, has a 64-bit Intel i7 processor 720QM, 16 GB PC1333 RAM and Windows 7 Ultimate 64-bit operating system. The code was written and compiled in 32-bit GCC.

6 RESULTS

The experimental results show significant performance gains using FBF. The results for SSN are shown below in Table 1 with the edit distance threshold $k = 1$ and the Jaro/Wink threshold set to 0.8 (0.75 for FN). Only 5000 of the 25,000,000 pairs actually match. The DL method is used as the baseline for all of the other methods. The first column is the method, the second is Type 1 errors (false positives), the third is Type 2 errors (false negatives), the fourth is time in milliseconds and the last is the performance gain over DL. Notice that all of the DL-based methods have very few Type 1 errors compared to Jaro and Wink. Ham is the only method that has Type 2 errors. The time had to be recorded in milliseconds because the FBF functions run very quickly.

As shown in the last row of Table 1, Gen, the $\text{SetNumBits}(s)$ described in the Methodology section function processes 10,000 SSNs in 0.6 milliseconds or 60 nanoseconds per FBF SSN signature. The FBF row shows the results for matching the strings using only the FBF filter. Notice that the filter has 123,318 Type 1 errors. This shows that FBF removed 12,369,182 unnecessary pair-wise comparisons before processing the strings with DL or PDL. FPDL is 62.24 times faster than DL. The average FBF comparison for an SSN was less than 58 nanoseconds per pair using $\text{FindDiffBits}(m,n,l)$.

Table 1. Accuracy and performance results for SSN string experiment

SSN	Type 1	Type 2	Time ms	Speedup
DL	42	0	52,807.2	1.00
PDL	42	0	17,449.2	3.03
Jaro	93,658	0	16,043.6	3.29
Wink	239,922	0	17,720.2	2.98
Ham	41	2,352	3,571.6	14.79
FDL	42	0	1,060.8	49.78
FPDL	42	0	848.4	62.24
FBF	123,318	0	729.0	72.44
Gen			0.6	88,012.00

Table 2 shows the experimental results for SSN with $k = 2$. Notice that even with a more relaxed match threshold, the edit distance methods still have less Type 1 errors and the FBF filter still provides significant performance gain with no loss to true positive matches. Notice that the FBF performance gain is less that with $k = 1$. The reason is that the FBF passed 1,344,669, 10.9 times as many candidate pairs. FPDL is still faster than Hamming distance.

Table 2. Accuracy and performance results for SSN experiment *k*=2

SSN2	Type 1	Type 2	Time ms	Speedup
DL	1,229	0	51,523.4	1.00
PDL	1,229	0	22,441.4	2.30
Jaro	93,658	0	15,473.6	3.33
Wink	239,922	0	17,120.0	3.01
Ham	1,014	0	3,518.4	14.64
FDL	1,229	0	3,625.6	14.21
FPDL	1,229	0	2,097.0	24.57
FBF	1,344,669	0	713.2	72.24
Gen			0.8	64,404.25

Fig. 6 shows the plot of the average runtime for an FBF filter comparison on a single pair of SSNs by total number of pairwise comparisons performed. The performance gain is stable and consistent with and average time of 58 nanoseconds for a FBF only pairwise comparison, 67.9 nanoseconds for FPDL and 84.9 nanoseconds for FDL. The average time for DL was 4,122.7 nanoseconds per comparison. The plot for LN was more erratic because the strings are not of fixed length as in SSN but seemed to converge on runtimes between 56 and 58 nanoseconds per comparison.

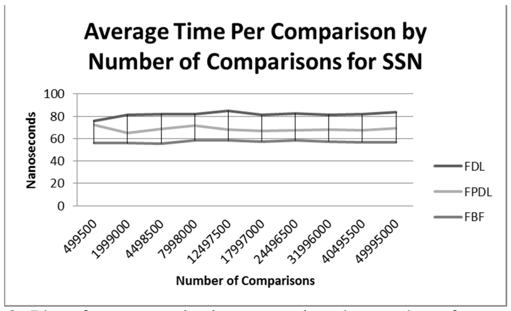


Figure 6. Plot of average pairwise comparison by number of comparisons

Table 3 shows the results for Census last names, LN, with edit distance threshold $k = 1$ and the Jaro/Wink threshold set to 0.8. There are also two lists with each containing 5,000 strings. The performance gain is still evident, the edit based methods continue to have lower Type 1 errors and the FPDL method is also approximately 3 times faster than Ham. The remaining results are similar and are reported in the Appendix.

Table 3. Accuracy and performance results for LN string experiment

LN	Type 1	Type 2	Time ms	Speedup
DL	766	0	31,073.2	1.00
PDL	766	0	6,201.0	5.01
Jaro	18,615	44	10,707.2	2.90
Wink	47,195	28	12,242.6	2.54
Ham	559	3,011	3,344.0	9.29
FDL	766	0	1,154.4	26.92
FPDL	766	0	1,138.6	27.29
FBF	20,174	0	1,142.6	27.20
Gen			0.8	38,841.50

Our best performance results were realized in the street address, Ad, experiments, shown in Table 4. The FDL method achieved a 78.18 times speedup and FPDL reached 79.6.

Table 4. Accuracy and performance results for Ad string experiment

Ad	Type 1	Type 2	Time ms	Speedup
DL	120	0	135,098.8	1.00
PDL	120	0	15,887.4	8.50
Jaro	103,368	0	35,034.8	3.86
Wink	192,108	0	36,587.8	3.69
Ham	69	3,444	5,537.8	24.40
FDL	120	0	1,728.0	78.18
FPDL	120	0	1,697.2	79.60
FBF	3,452	0	1,664.6	81.16
Gen			2.0	67,549.40

Table 5 shows the performance gains for FPDL compared to all non-filtered methods. FPDL is almost 80 times faster than DL for comparing address strings and almost 5 times faster than Ham for phone numbers—Hamming distance has an $O(n)$ computational complexity and FPDL is faster and has the matching power of DL. The results show that FBF yields better performance on longer strings. Street addresses are the longest, phone numbers are the second and SSNs are the third longest. The difference is due to DLs $O(mn)$ complexity. Table 4 shows the performance speedups from the smallest average length, FN on the left, to the longest, Ad on the right.

Table 5. Runtime and speedup for FPDL versus all other methods

FPDL	FN	LN	Bi	SSN	Ph	Ad
DL	23.23	26.10	42.46	62.24	75.00	79.60
PDL	6.04	5.22	15.91	20.57	22.63	9.36
Jaro	8.76	9.52	14.08	18.91	23.87	20.64
Wink	10.08	11.06	15.80	20.89	25.98	21.56
Ham	2.89	3.00	3.86	4.21	4.71	3.26

Table 6 shows the results from an RL experiment using FBF. The RL method was a simple deterministic point and threshold based algorithm. There were two datasets—one with 1000 clean records and the other with 1000 single edit error injected records. The table shows that FDL is 45 times faster and FPDL is 48.9 times faster than the baseline DL-based RL.

Table 6. Performance results for RL experiment

RL	DL	PDL	FDL	FPDL	FBF	Gen
Time ms	13762.0	3464.6	305.6	281.6	273.2	2.0
Speedup	1.0	4.0	45.0	48.9	50.4	6881.0

For completeness, we provide results from experiments using the Soundex (SDX) in Table 7. These experiments were performed using the same datasets as described for the previous experiments—one clean dataset and one with single edit errors injected. DL is 2.3 times slower for first names and 2.6 times slower for last names. When implemented in aforementioned RL system, DL-based RL was 5 times slower than the department's proprietary system. FPDL is 10.3 time faster than the Soundex for first names and 10.8 times faster for last names. We did not compare the Soundex to the other data because it is primarily a name matching phonetic algorithm. The accuracy of the Soundex is much worse than DL. In both cases, the Soundex found less than half of the true positive matches and the number of false positives are 6.4 times that of DL for first names and almost 40 times greater than DL for last names. The Soundex even has false negative results.

Table 7. Soundex vs. DL with error injected

Error	**TP**	**FN**	**FP**	**TN**	**Time ms**
FN-DL	5,000	0	6,458	24,988,542	24,586
FN-SDX	2,259	2,741	47,137	24,947,863	10,664
LN-DL	5,000	0	766	24,994,234	32,308
LN-SDX	2,499	2,501	30,606	24,964,394	12,344

Table 8 shows the results of matching the clean dataset against itself. Both found all true positives and both have higher false positives than the previous experiment but the false positives are much higher for the Soundex than for DL. These results show that the presence of single edit data entry errors can cripple the Soundex's ability to find true positive matches. In both experiments, the Soundex suffers from substantially higher false positive matches.

Table 8. Soundex vs. DL with clean data

Clean	**TP**	**FN**	**FP**	**TN**	**Time ms**
FN-DL	5,000	0	18,268	24,976,732	24,464
FN-SDX	5,000	0	70,476	24,924,524	10,936
LN-DL	5,000	0	1,760	24,993,240	31,586
LN-SDX	5,000	0	37,654	24,957,346	11,938

Fig. 7 shows the results of the runtime curves from the second set of experiments. Notice that the greatest growth rate is DL and the smallest growth rates are the FBF methods (FDL and FPDL). The FBF methods grow slower than Hamming distance and almost appear linear when compared to DL in this context.

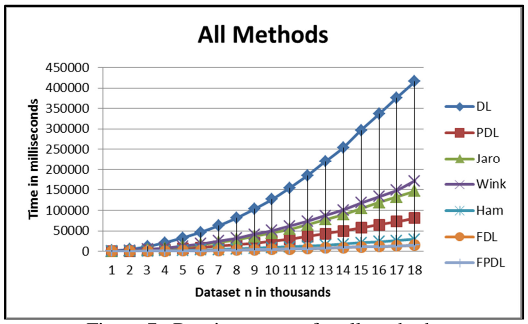


Figure 7. Runtime curves for all methods

The runtimes and n for each method were analyzed with Matlab's polyfit function for a second degree polynomial solution in the form $an^2 + bn + c$. Table 9 below shows the coefficients for the second and first degree terms, and the constant. The FBF methods' growth rates (a for FDL and a for FPDL) are two orders of magnitude smaller than DL (a for DL).

Table 9. Coefficients and constants for speedup polynomials for FBF

	DL	**PDL**	**Jaro**	**Wink**	**Ham**	**FDL**	**FPDL**	**Fil**
a	1.32E-03	2.57E-04	4.68E-04	5.48E-04	9.30E-05	4.69E-05	4.67E-05	4.57E-05
b	-0.374	-0.080	-0.171	-0.496	-0.039	-0.008	-0.013	-0.012
c	512.739	127.316	247.971	1134.396	71.392	12.328	28.035	27.081

The actual speedups for the experiments comparing speedup for FPDL against DL are shown in Table 10 below. Based on the polynomial, the expected speedup of FPDL over DL for very large n (500,000 or more) is approximately 28.3 and is stable. DL would take approximately 3.8 days to merge 2 datasets with 500,000 strings each. FPDL would only require 0.13 days.

Table 10. Speedups for last name data by *n*

n	**speedup**	***n***	**speedup**
1,000	27.6	10,000	28.2
2,000	27.3	11,000	28.3
3,000	27.7	12,000	28.6
4,000	28.3	13,000	28.4
5,000	27.9	14,000	28.0
6,000	27.8	15,000	28.2
7,000	27.8	16,000	28.2
8,000	28.0	17,000	28.5
9,000	28.1	18,000	28.1

Fig. 9 shows the comparison curves for FBF, length filtering and both methods combined for last name. The bottom curve is LFPDL, which is also obscuring the LFDL curve. These are the fastest methods. The Length filter methods LDL and LPDL were the slowest. The results for the FBF methods, FDL and FPDL from Table 3, are the middle two curves.

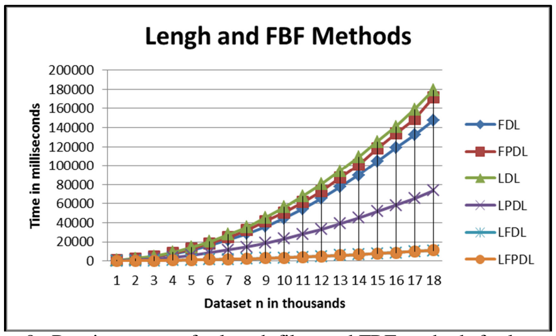


Figure 9. Runtime curves for length filter and FBF methods for last name

The results of polyfit on the runtime curves from Fig. 9 are shown in Table 11. Notice that the coefficient a for LFPDL, 3.41E-05, is about 27% smaller than the a coefficient for FPDL in Table 7.

Table 11. Coefficients and constants for speedup polynomials for length filter

	LDL	**LPDL**	**Len**	**LFDL**	**LFPDL**	**LFil**
a	5.38E-04	2.21E-04	9.23E-06	3.34E-05	3.41E-05	3.21E-05
b	0.263	0.119	0.004	0.012	0.001	-0.003
c	-531.126	-244.743	-9.159	-10.796	6.730	14.420

Table 12 shows the last name results for DL and FPDL from Table 3 and new results for length filtering (LF) and the combination of length filtering and FBF. Notice that the FPDL method is 27.29 times faster than the traditional DL method but the combination of both methods is 36.01 times faster than the DL method—an improvement of about 32% over FPDL. Also notice that LF or the length filter only method was 127.52 times faster than DL. The LF row shows that the length filter passed 11,196,547 string pairs of 12,497,500 possible pairs. This is why the LDL and LPDL methods did not perform as well as the FBF or combined filter methods. The combined decreased the number of calls to DL and PDL to 12,735 as shown in the LFBF row. None of the methods in table 10 or any of the other experiments that use these filtering methods have any type 2 errors.

Table 12. Accuracy and performance results for LN string experiment using length filter

LN	Type1	Type2	Time ms	Speedup
DL	766	0	31,073.2	1.00
FPDL	766	0	1,138.6	27.29
LDL	766	0	13,599.0	2.28
LPDL	766	0	5,666.7	5.48
LF	11,196,547	0	243.7	127.52
LFDL	766	0	890.7	34.89
LFPDL	766	0	863.0	36.01
LFBF	12,735	0	795.3	39.07

The main reason for this performance gain is that the minimum length of a last name in the Census data is 2 and the maximum length is 15 with an average string length of 6.89. There is increased efficiency by limiting calls to FBF's FindDiffBits(m, n, l) function for every possible string pair. The FBF row in Table 3 shows 20,174 calls passed to be verified to DL and PDL and LFBF only processed 12,735—a savings of 7,439 calls. The increased precision is from adding FBF to length filtering. Table 11 shows the counts for string lengths for the Census last name data used in our experiments.

Table 13. Counts of Census last name string lengths

Length	Frequency	Length	Frequency
2	175	9	14424
3	1585	10	7772
4	8768	11	3215
5	23238	12	1190
6	34025	13	442
7	33256	14	177
8	23380	15	23

Table 14 shows the street address results for DL and FPDL from Table 4 and new results for length filtering and the combination of length filtering and FBF. The combined method increased the speedup of FPDL from 79.6 times faster than DL to 130.83 times faster for LFPDL.

Table 14. Accuracy and performance results for Ad string experiment using length filter

Ad	Type1	Type2	Time ms	Speedup
DL	120	0	135,098.8	1.00
FPDL	120	0	1,697.2	79.60
LDL	120	0	48,879.3	2.76
LPDL	120	0	14,343.3	9.42
LF	9,623,583	0	237.3	569.24
LFDL	120	0	1,164.0	116.06
LFPDL	120	0	1,032.7	130.83
LFBF	3,200	0	985.3	137.11

7 CONCLUSION

As more data are stored in clouds, identity resolution is a commonly required function for health and human services, fraud detection and other security services. Our goal was to reduce unnecessary edit distance 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 FBF over even the leanest string distance metrics. The results also show that there is absolutely no loss to accuracy—FBF, when combined with DL or PDL, produces the same results as DL. FBF 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 FBF signatures is very low and only 4 bytes are needed for a numeric string, 8 bytes for an alphabetic string (to count two occurrences of each character) and 12 for an alphanumeric string. 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 not unreasonable given the performance benefits. The record pair search spaces for FDL and FPDL are as exhaustive as DL but completes each comparison, on average, much quicker by eliminating unnecessary work when comparing string pairs. FBF and PDL work efficiently together to deliver the same resulting accuracies as DL, in significantly less time. Our results provide evidence that FPDL can perform the same work in as little as one day that would take DL nearly 80 days to complete. The 40 hour record linkage (RL) update mentioned in the Introduction can now be completed in an hour or two. This performance is far superior to using low complexity Soundex alone with more than 46% yield in true positive matches. Our goal is to implement a distributed in-memory data graph to process demographic data and resolve entities within the data in a cloud.

Our methodology substantially differs from SSJoin [30] in similarity handling. SSJoin is a DBMS operator that uses edit similarity and other string comparators for approximate joins. It also uses a prefix similarity filter based on matching tokens. The similarity filter cannot guarantee zero accuracy loss. FBF instead focuses on the difference between strings, which allows it to use a single machine instruction, the exclusive or, to find a subset of differing characters—in nanoseconds.

The Appendices contain tables for experiments with first names, phone numbers and birthdates. Sample code and datasets (used in experiments) can be downloaded at: http://astro.temple.edu/~joejupin/FBF.zip

8 REFERENCES

[1] Dunn, H. L.: Record Linkage, In: American Journal of Public Health 36 (12): pp. 1412–1416. doi:10.2105/AJPH.36.12.1412. (1946)

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

[3] Knuth, D. E.: The Art of Computer Programming: Volume 3, Sorting and Searching. Addison-Wesley. pp. 391–92. ISBN 9780201038033. OCLC 39472999 (1973)

[4] Stanier, A.:, Computers in Genealogy, Vol. 3, No. 7 (September 1990)

[5] Lait, A.J. and Randell, B.: AN Assessment of Name Matching Algorithms, Technical Report Series, University of Newcastle Upon Tyne Computing Science (1996)

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

[7] 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)

[8] 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)

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

[10] McCallum, A., Nigam, K., Ungar, L.: Efficient clustering of high-dimensional data sets, In: Proc. of 6th ACM SIGKDD Int. Conf. on KDD, pp. 169–178 (2000)

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

[12] Baxter, R., Christen, P., Churches, T.: A Comparison of Fast Blocking Methods for Record Linkage, In: First Workshop on Data Cleaning, Record Linkage and Object Consolidation, KDD 2003, Washington DC, August 24-27 (2003)

[13] 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)

[14] Cohen, W., Ravikumar, P., Fienberg, S.: A Comparison of String Metrics for Matching Names and Records . In Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web, 2003.

[15] Winkler, W. E.: String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage, In: Proceedings of the Section on Survey Research Methods, American Statistical Association: pp. 354-359 (1990)

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

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

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

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

[20] Wang, J., Feng, J., Li, G.: Trie-Join: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints, In: Proc. of the VLDB Endowment, V.3, No.1 (2010)

[21] Microsoft MSDN: http://msdn.microsoft.com/en-us/library/x79h55x9.aspx

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

[23] US Census: http://www.census.gov/

[24] Hamming, R. W.: Error detecting and error correcting codes, In: Bell System Technical Journal 29 (2): 147–160, MR0035935 (1950)

[25] NANP: North American Numbering Plan: http://www.nanpa.com/

[26] Social Security Administration: The SSN Numbering Scheme: http://www.socialsecurity.gov/

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

[28] Navarro, G.: A guided tour to approximate string matching", ACM Computing Surveys (CSUR) Volume 33 Issue 1, Pages 31 – 88, March 2001

[29] Gravano, L., Ipeirotis, H., "Approximate string joins in a database (almost) for free", In Proc. 27th Intl. Conf. on VLDB, pages 491 – 500, 2001

[30] Chaudhuri, S., Ganti, V., Kaushik, R.: "A Primitive Operator for Similarity Joins in Data Cleaning", In Proc. 22nd Intl. Conf. on Data Engineering, Page 5, 2006

9 APPENDIX: OTHER EXPERIMENTAL RESULTS

Table 9. Accuracy and performance results for FN string experiment

FN	Type 1	Type 2	Time ms	Speedup
DL	6,458	0	24,081.4	1.00
PDL	6,458	0	6,257.0	3.85
Jaro	215,874	102	9,080.0	2.65
Wink	314,994	102	10,450.4	2.30
Ham	4,539	2,972	3,000.8	8.02
FDL	6,458	0	1,102.0	21.85
FPDL	6,458	0	1,036.6	23.23
FBF	91,072	0	996.2	24.17
Gen			0.6	40,135.67

Table 10. Accuracy and performance results for Ph string experiment

Ph	Type 1	Type 2	Time ms	Speedup
DL	7	0	63,311.6	1.00
PDL	7	0	19,102.6	3.31
Jaro	82,748	10	20,153.8	3.14
Wink	567,118	10	21,930.0	2.89
Ham	7	2,272	3,976.0	15.92
FDL	7	0	961.6	65.84
FPDL	7	0	844.2	75.00
FBF	61,277	0	738.8	85.70
Gen			0.4	158,279.00

Table 11. Accuracy and performance results for Bi string experiment

Bi	Type 1	Type 2	Time ms	Speedup
DL	7,899	0	42,121.0	1.00
PDL	7,899	0	15,786.8	2.67
Jaro	597,466	7	13,971.2	3.01
Wink	1,470,453	7	15,673.6	2.69
Ham	6,152	3,006	3,833.8	10.99
FDL	7,899	0	1,368.8	30.77
FPDL	7,899	0	992.0	42.46
FBF	355,860	0	711.4	59.21
Gen			1.0	42,121.00