TEMPORAL GRAPH RECORD LINKAGE AND *K*-SAFE APPROXIMATE MATCH

A Dissertation
Submitted to
the Temple University Graduate Board

In Partial Fulfillment
of the Requirements for the Degree
DOCTOR OF PHILOSOPHY

By
Joseph Jupin
December 2016

Examining Committee Members:

Dr. Justin Yuan Shi, Advisory Chair, CIS Department
Dr. Yuhong Guo, CIS Department
Dr. Edward Dragut, CIS Department
Dr. Bo Ji, CIS Department
Dr. Boleslaw K. Szymanski, External Member, RPI CS Department

ABSTRACT

Temporal Graph Record Linkage and *k*-Safe Approximate Match:

Graph Aggregated Temporal and Relational Data Enhanced Record Linkage with Iterative Matching and Hash Optimized Edit-Distance String Comparison

By
Joseph Jupin
email: joejupin@temple.edu
Temple University, 2016

Committee:
Dr. Justin Yuan Shi, Advisor and Chair
Dr. Yuhong Guo
Dr. Edward Dragut
Dr. Bo Ji
Dr. Boleslaw K. Szymanski

Since the advent of electronic data processing, organizations have accrued vast amounts of data contained in multiple databases with no reliable global unique identifier. These databases were developed by different departments for different purposes at different times. Organizing and analyzing these data for human services requires linking records from all sources. RL (Record Linkage) is a process that connects records that are related to the identical or a sufficiently similar entity from multiple heterogeneous databases. RL is a data and compute intensive, mission critical process. The process must be efficient enough to process big data and effective enough to provide accurate matches.

We have evaluated an RL system that is currently in use by a local health and human services department. We found that they were using the typical approach that was

offered by Fellegi and Sunter with tuple-by-tuple processing, using the Soundex as the primary approximate string matching method. The Soundex has been found to be unreliable both as a phonetic and as an approximate string matching method. We found that their data, in many cases, has more than one value per field, suggesting that the data were queried from a 5NF data base. Consider that if a woman has been married 3 times, she may have up to 4 last names on record. This query process produced more than one tuple per database/entity apparently generating a Cartesian product of this data. In many cases, more than a dozen tuples were observed for a single database/entity. This approach is both ineffective and inefficient. An effective RL method should handle this multi-data without redundancy and use edit-distance for approximate string matching. However, due to high computational complexity, edit-distance will not scale well with big data problems.

We developed two methodologies for resolving the aforementioned issues: PSH and ALIM. PSH – The Probabilistic Signature Hash is a composite method that increases the speed of Damerau-Levenshtein edit-distance. It combines signature filtering, probabilistic hashing, length filtering and prefix pruning to increase the speed of edit-distance. It is also lossless because it does not lose any true positive matches. ALIM – Aggregate Link and Iterative Match is a graph-based record linkage methodology that uses a multi-graph to store demographic data about people. ALIM performs string matching as records are inserted into the graph. ALIM eliminates data redundancy and stores the relationships between data.

We tested PSH for string comparison and found it to be approximately 6,000 times faster than DL. We tested it against the trie-join methods and found that they are up to 6.26

times faster but lose between 10 and 20 percent of true positives. We tested ALIM against a method currently in use by a local health and human services department and found ALIM to produce significantly more matches (even with more restrictive match criteria) and that ALIM ran more than twice as fast.

ALIM handles the multi-data problem and PSH allows the use of edit-distance comparison in this RL model. ALIM is more efficient and effective than a currently implemented RL system. This model can also be expanded to perform social network analysis and temporal data modeling. For human services, temporal modeling can reveal how policy changes and treatments affect clients over time and social network analysis can determine the effects of these on whole families by facilitating family linkage.

ACKNOWLEDGEMENTS

I would like to thank my advisor, committee chair and associate chair of the CIS department, Dr. Justin Yuan Shi for all of his help and guidance with this project. Dr. Shi provided me with this research problem after being contacted by a local health department to assess their Record Linkage system. Through his efforts, we had the very rare privilege of access to a real system with real data. This access is usually very restricted due to HIPPA privacy regulations. He was always there to help me when meeting with health department officials and in developing a methodology to assess and report in the health departments RL system. He has been very patient and has invested time and effort into our project. He always made time to meet and discuss strategies and solutions to problems that would arise during the development of the solution to or problem.

I also thank the other members of my dissertation committee, Dr. Yuhong Guo, Dr. Edward Dragut, Dr. Bo Ji and Dr. Boleslaw K. Szymanski for their time and consideration and for agreeing to participate and review in my work. I am glad to have had faculty with experience and interest in data problems accept involvement in my project.

I would also like to thank the members of my former project, analyzing data from the ProDES project. Dr. Zoran Obradovic was my former advisor and committee chair. He provided me with experience working on data collected from multiple sources and all the issues that these data represent. Dr. Philip W. Harris was our primary investigator and provided the problem and the data along with valuable domain advice for working with

Criminal Justice and Social Sciences data. Dr. Alan Izenman, Dr. Jeremy Mennis and Dr. Heidi Grunwald were other members of the group that also provided insight into processing data, analyzing data and reporting on results obtained from our research. My fellow graduate students: Brian Lockwood and Yilian Qin. Hope that their post Temple careers are both successful and satisfying.

I would also like to thank officials from the Philadelphia Department of Health. Dr. Donald Schwarz, Deputy Mayor of Health and Opportunity, Susan Kretsge, Chief of Staff for the Office of Health and Opportunity, Swamy Bodige, Analyst for the Office of Health and Opportunity, LaVern Wright, Program Director for Integrated Systems for the Office of Health and Opportunity, and Dr. Zheng Wen, IFI, for their advice and support in assessing and understanding their RL system. I would also thank the City of Philadelphia's Office of Health and Opportunity for grant support for this project.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF EQUATIONS

PART I

PREFACE

CHAPTER 1

INTRODUCTION TO RECORD LINKAGE

The task of merging heterogeneous data sources has many names: record linkage, merge/purge, entity resolution, entity identification, deduplication, data cleansing, etc. Combining such data is an important first step in the knowledge discovery process [Fayyad 1996], whether aggregation, data mining or statistical analysis of data is to be performed. Many modern organizations have databases that were designed and developed by different people, at different times and for different purposes. As organizations evolve and grow, their information needs change. Corporate mergers and departmental consolidations create conditions where it is necessary to link or combine multiple heterogeneous data sources. These organizations, such as corporations and governmental departments, need to generate information from data analysis to make decisions. An unconscious patient may be admitted to a hospital emergency room, unable to provide information about drug allergies, medical conditions or prior procedures. Hospital Emergency Rooms and Trauma Centers can benefit greatly from integrated patient data reports from multiple hospitals and doctors' offices—the patient's life depends on medical professionals having complete information, especially if the patient is unconscious and cannot provide it. If a caseworker for a social services agency is evaluating services for a client and the client is receiving services from multiple agencies, the caseworker needs to acquire complete information from all databases to best serve the client. If an analyst is preparing an aggregate report of all clients within

the system, the analyst needs accurate linkages between each client's records within the system to generate accurate results. Federal, State and Municipal governments all have databases that contain a myriad of social, health and financial data about citizens. Many of these databases are not linked because they were designed and implemented by different departments or organizations with different schemas and may not have a reliable universal unique identifier.

The primary objective of *Record Linkage* (RL) is to identify records of a single entity from within one (deduplication) or across two or more (linking) data sources. Due to the potential for differing database schemas, inconsistent data, and no available or reliable universal global identifier, RL can be a very challenging task. The Social Security Number, if available, can be used as a universal identifier but has some weaknesses, such as that it is neither self-checking nor self-correcting as described in [Kuch 1972]. As many as 32 percent of SSNs have been reported missing from health related databases [Campbell 2005]. We have observed health and social services databases with more than 46% missing SSNs. Data entry errors, inconsistencies, changes in data over time, such as change of the last name, due to marriage, or address, due to the entity moving its physical location, and missing data create challenging hurdles to accurately match a single entity's records in one or more data sources.

We are currently engaged in research that aims to improve some of the tools required to perform RL. We have performed research on a deterministic RL model, currently being used by a local government health department and found modifications and innovations that can improve RL accuracy. The current system uses the Soundex for approximate string matching. We have demonstrated that an edit distance based method will perform

better and has been found to reveal approximately 80% of data entry errors [Damerau 1964]. However, edit distance is approximately five times slower than the Soundex. We have evaluated the literature, an existing record linkage system, data from a local health department, and collected information from the system's data, related to this issue. This research has revealed some valuable insights into the health and social sciences record linkage problem.

A perfect solution for RL is unlikely—even humans cannot match some records that refer to the same entity due to errors, inconsistencies and missing data. Just like many other computer science problems, each step in the RL process is only as dependable as the last. The current RL technologies generally focus on a part of the problem and ignore certain important qualities of health and social sciences data that should really define a better methodology than currently exists within this domain.

1.1 Challenges and Principals of Record Linkage

To summarize the challenges we have found in the literature and our own experiments, we offer our principals for the development of an improved RL model. An effective RL system must:

1. Handle the temporal data problem (aggregate records)
2. Remove data and processing redundancies
3. Ensure transitive closure on RL (Iterative RL)
4. Handle synonymous terms (aliases), phonetic terms and minor spelling errors

5. Guarantee that all optimizations are lossless with respect to accuracy
6. Provide a mechanism to allow system users to correct mistakes in RL
7. Make use of available contextual data (family members)
8. Provide a mechanism for the future implementation of family linkage (SNA)
9. Handle both RL and deduplication operations, concurrently
10. Perform approximate string matching faster with our new and novel hybrid hash method
11. Create better quality comparison vectors than is possible with other methodologies

1.2 Proposed Methodology

Record linkage attempts to assemble all of a given individual's data from multiple heterogeneous data sources into a comprehensive and complete report of his or her experiences (services and treatments) within the system. Social workers and health professionals require these reports to best serve their clients, efficiently and effectively. We have found that it is useful and beneficial to consider all of a client's historical identifying data (temporal data) and relationships (family and social network) with other entities in the system and developed *Aggregate-Link-Iterative-Match* (ALIM), an efficient way to process, store, query and match the data to develop a more accurate comparison vectors for enterprise database entities that can be used for deterministic, probabilistic and data mining record matching algorithms. ALIM is based on a temporal data graph structure that links all entities' historic and family relational data and pre-

computes and stores all alias, approximate and phonetic matches for data elements and relationship links for records to provide a more efficient and effective model. This is offered as a partial solution to the Record Linkage (RL) problem.

1.3 Envisioned Contributions

The primary contribution of this work is an innovative method for handling RL data, the efficient generation of comparison vectors that capture the essence of entities' histories, storage of family relationships and use previously discovered matched records to iteratively facilitate a transitive closure for new matches. This method addresses a recognized but unsolved temporal dimension that will increase the value of the comparison vectors used with the selected matching algorithm. This method can perform pre-computed RL for speed and efficiency and real-time RL to provide the most up-to-date results based on the current state of the data. An ancillary contribution is the development of a probabilistic hybrid hashing, filtered and pruned optimization for the Damerau-Levenshtein edit distance algorithm, called Probabilistic Signature Hashing (PSH), that has been found to be almost 6,000 times faster than using Damerau-Levenshtein alone. PSH also produces the exact same string matching results as DL. This is significant because DL can detect up to 80% of data entry errors.

1.4 Organization of Dissertation

This report is organized into five major parts. The Introduction section below introduces the Record Linkage problem, proposed methodology, contributions to domain and the notation and formalization in this document. The Background section is arranged to present the material as it occurs in the RL process. The topics are preprocessing by data standardization, reduction of data processing by blocking, field-level comparison using phonetic and string distance metrics, optimizations for string distance metrics, a survey of existing RL methods. The Findings section discusses our findings based on the review of the literature, assessment of an RL system currently in use by an urban health department and experimental results. The Methodologies, Experiments and Results section describes a history and current status for our string matching optimization, PSH, and our RL system, ALIM. It chronicles our methodologies, experiments and results. The Concluding Remarks section discusses our contributions to the domain and future work.

1.5 Notation and Formalization

The table below shows the formal notation used in this document unless otherwise proximally stated. All variables, acronyms for functions and acronyms for composites of functions will be *italicized*. Acronyms for methodologies will not be italicized.

Concept	**Notation**
Alphabet	Σ
Set of all strings over Σ	Σ^*
Sets of strings in Σ^*	S, T
Reduced alphabet of Σ	σ
The length of a set of strings in Σ^*	$\|S\|$
Individual strings in S and T, $s \in S$, $t \in T$	s, t
Individual characters in strings s, and t, $s_i \in s$, $t_j \in t$	s_i , t_j
The length of a string	$\|s\|$
Individual character variables	a, b, c, d
A function to calculate $\|s\|$	$l(s)$
A length filter comparison function	$L(s, t)$
A signature filter generation function	$f(s)$
Signatures of $f(s)$ or $f(t)$	x, y
Individual bits in signatures x and y, $x_i \in x$ and $y_j \in y$	x_i, y_j
A signature filter comparison function	$F(x, y)$
Maximum edit distance between approximately matched strings	k
Damerau-Levenshtein edit distance	$DL(s, t)$
Prefix Pruned Damerau-Levenshtein edit distance	$PDL(s, t, k)$
A function to generate a hash code	$h(s)$
Hash codes of $h(s)$ and $h(t)$	u, v
Hash tables that contain u and v	U, V
An exhaustive search function to find a set of candidate hash buckets for pattern u in table V	$H(u, V)$
A function to generate all related bucket codes related to edit distance k for pattern u in table V	$G_k(u, V)$
A set of candidate hash buckets	C
An individual hash bucket containing a list of strings	C_i
Sets of records	Q, R
Individual records in Q and R, $q \in Q$ and $r \in R$	q, r
Individual records $r_i \in R$	r_i
A comparison vector	γ
A set of comparison vectors	Γ
A set of matching record or string pairs	M
A set of unmatched record or string pairs	U
A set of possible match record pairs	P

CHAPTER 2

BACKGROUND

Record Linkage (RL) is a process that compares pairs of records from heterogeneous databases to find similar or identical entities [Dunn 1946]. We use RL to describe this operation in a general sense whether it refers to merging entities from heterogeneous databases or purging duplicate entities from the same database.

If RL is applied to a single data source, the process is typically an attempt to detect duplicate entries and is referred to as duplicate detection or deduplication. If RL is applied to two or more data sources, the process is likely an attempt to match same entities from multiple data sources. This process is known under a many names, which has caused a great deal of confusion when cross-referencing articles. Statisticians, health services and historians primarily refer to RL as record linkage. Database professionals generally refer to RL as merge/purge processing or list washing. Computer and information scientists may refer to it as data matching, entity resolution, entity identification, record matching or as the object identity problem. Other names for RL are: entity disambiguation, instance identification, co-reference resolution, reference reconciliation and database hardening [Elmagarmid, Ipeirotis and Verykios 2007] [Christen and Churches 2003].

Halbert L. Dunn of the U.S. National Bureau of Statistics conceived the concept of record linkage (RL) in 1946. He described the process in terms of a person's life:

"Each person in the world creates a Book of Life. This Book starts with birth and ends with death. Record Linkage is the name of the process of assembling the pages of this Book into a volume." [Dunn 1946]

Howard Newcombe, a Canadian geneticist, and his associates Kennedy, Axford, and James were the first to perform record linkage using a computer in 1959. They took advantage of a computer's speed, accuracy and consistency to arrange personal records into family histories. Newcombe compiled a manual, the "Handbook of Record Linkage Methods for Health and Statistical Studies," in 1988. [Newcombe 1988]

The first formal mathematical theory of probability for record linkage was published in an innovative publication, "A Theory for Record Linkage," by Ivan Fellegi and Alan Sunter in 1969, which uses the relative frequencies of strings to be compared. They proved that their probabilistic decision rule model produced optimal results when the data's attributes are conditionally independent. This method is a decision-theoretic approach to RL, which established the validity of the method first applied by Newcombe et al. [Fellegi and Sunter 1969]. Methods based on this work generally have three steps to link records: searching, decision-making and grouping. The searching step generally uses a very fast but less precise comparator to decrease the number of pair-wise comparisons than can be done with a slower but more accurate comparator. This is usually referred to as blocking. The decision-making step compares records with a more accurate comparator, which produces a probability that two records represent the same entity. This method compares each record from one file to those in another, which produces 3 categories: records that are definitely matched *M*, records are definitely not matched *U*, and records that are possibly matched *P*. The possible matches are set aside

for further consideration by a domain expert because not enough data exists to compute a positive of negative match. We describe a traditional approach to RL as described by Fellegi and Sunter in their seminal RL work. Given two datasets *Q* and *R*, we compare all unique record pairs in *Q* and *R*:

Equation 1: Fellegi and Sunter Domain

$$Q \times R = \{(q,r) : q \in Q, r \in R\}$$

Use a combination of blocking methods, string comparison metrics and matching rules to determine the set of matches *M*:

Equation 2: Fellegi and Sunter Matches

$$M = \{(q,r) : q = r, q \in Q, r \in R\}$$

Determine the set of un-matches *U*:

Equation 3: Fellegi and Sunter Un-matches

$$U = \{(q,r) : q \neq r, q \in Q, r \in R\}$$

Record the possible matches *P* for human comparison:

Equation 4: Fellegi and Sunter Possible Matches

$$P = \{(q,r) : (q,r) \notin M, (q,r) \notin U, q \in Q, r \in R\}$$

Generally there are two thresholds that are used to decide which record pairs are assigned to *M* and *U*, i.e. all pairs with a threshold greater than or equal to *x* are matches and all pairs with a threshold less than or equal to *y* are un-matches. The possible matches are those that fall between the thresholds. The set of matches *M* are (q,r) that have Record a comparison vector γ containing the agreements and disagreements for each record's attributes (first name, last name, address, etc.) numbered from 1 to *n*:

Equation 5: Fellegi and Sunter Comparison Vector

$$\gamma\left[\alpha(q),\beta(r)\right] \equiv \left\{\gamma^1\left[\alpha(q),\beta(r)\right],\gamma^2\left[\alpha(q),\beta(r)\right],\ldots,\gamma^n\left[\alpha(q),\beta(r)\right]\right\}$$

Γ is the set of all function γ on $Q \times R$.

Equation 6: Fellegi and Sunter Set of Comparison Vectors

$$\Gamma = \forall\gamma(q,r) \in Q \times R$$

The conditional probability of observing a given γ for $(q,r) \in M$ is:

Equation 7: Fellegi and Sunter Conditional Probability for a Match

$$m(\gamma) = P\left\{\gamma\left[\alpha(q),\beta(r)\right] \mid (q,r) \in M\right\} = \sum_{(q,r)\in M} P\left\{\gamma\left[\alpha(q),\beta(r)\right]\right\} \cdot P\left[(q,r) \mid M\right]$$

The conditional probability of observing a given γ for $(q,r) \in U$ is:

Equation 8: Fellegi and Sunter Conditional Probability for a Un-Match

$$u(\gamma) = P\left\{\gamma\left[\alpha(q),\beta(r)\right] \mid (q,r) \in U\right\} = \sum_{(q,r)\in U} P\left\{\gamma\left[\alpha(q),\beta(r)\right]\right\} \cdot P\left[(q,r) \mid U\right]$$

The set comparison vectors Γ is used to assign candidate pairs from $Q \times R$ into *M*, *P* and *U* using a linkage rule *L*. Most probabilistic RL methods are wrappers for deterministic methods and simply adjust weight values awarded to each field comparison to consider probabilities revealed by the data. An example is that "James Smith" is the most likely first name and last name in the 1990 Census data and "Alonso Cooke" is the least [U. S. Census 1990]. A match on the former would carry less weight than the later because there are likely to be many more James Smiths in the population. The matched pairs are stored for future queries. A model for a traditional RL system for merging databases is shown in Figure 1: Record Linkage (RL) process to merge two databases.

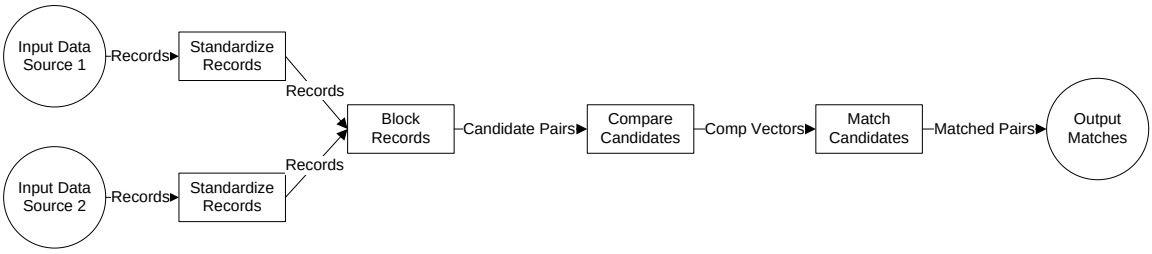


Figure 1: Record Linkage (RL) process to merge two databases

2.1 Standardization

The first step in the process is to standardize the records. A standard form for an address may be *{number direction street designation}* as in "1234 S BROAD ST", where direction is a single letter N, S, E, and W for North, South, East and West, and designation is abbreviated as AVE, ST, RD, etc. for Avenue, Street, Road, etc. A popular and useful string distance metric is the Damerau-Levenshtein edit distance [Damerau 1964], which counts the number of edits (character substitutions, insertions, deletions and transpositions) that are required to transform a string *s* into string *t*. Standardization is necessary because if *s* = "1234 S BROAD ST" and *t* = "1234 South Broad Street", the edit distance between *s* and *t* is 8, which is a significant difference.

This report considers an RL solution based in findings (discussed in the Findings section) for a large historical data warehouse for a local urban health department that uses seven demographic fields shown in Table 1: First Name (FN), Last Name (LN), Address (Ad), Phone Number (PN), Gender (Gn), Birth Date (BD) and Social Security Number (SSN) because these are used in a local government system that we have evaluated. Also, the

data has errors, inconsistencies and missing fields. Our objective is to develop a method to match records across multiple heterogeneous databases for a local city government's health department. These databases were developed by different departmental agencies for different purposes at different times. They all have the aforementioned fields in common but there are inconsistencies, errors and missing data that will cause exact string matching to fail to link records that belong to the same entity. There are approximately 50 million records for 1.5 million entities because there is more than one record in each database for many of the entities that record how the entity has changed over time. In forty percent of the records, the SSN is missing and there are errors in some of the SSNs that are not missing. Phone numbers and addresses can change, as well as last names for women due to marriage or divorce. Aliases or nicknames and initials may be used instead of first names. Any of these records may have data capture or entry errors and all strings are relatively short. A cursory scan of a few of these data sets immediately revealed individuals that had moved from one address to another and had changed phone numbers.

2.2 Blocking

Blocking is a method for decreasing the number of pair wise comparisons performed by record matching algorithms. Processing every record pair in very large data sets may not be practical or even possible using a comprehensive RL algorithm, such as one based on DL edit distance. The idea is that some key fields are used to decide candidate pairs for future record matching, while non-candidates are immediately disqualified. The blocking

method should be much faster than the matching method. The main idea is to perform a fast pass over the data to substantially decrease future work. The blocking algorithm should remove a significant number of compared record pairs from the set of unblocked pairs. Blocking produces a set of candidate pairs:

Equation 9: Blocking

$$\langle q, r \rangle \subset Q \times R, q \in Q, r \in R, |\langle q, r \rangle| \ll |Q \times R|$$

which are passed to the Compare Candidates step.

Traditional blocking methods include standard blocking [Jaro 1989], sorted neighborhood [Hernandez and Stolfo 1998], bigram indexing [Christen and Churches 2003] and canopy clustering with tf-idf [McCallum, Nigam and Ungar 2000] [Cohen and Richman 2002]. In [Baxter, Christen and Churches 2003] an analysis of each method is described. Below is a brief description and how they relate to the problem above.

Standard blocking uses one or more attributes to create a blocking key that is used to cluster records into blocks by identical blocking keys. An example would be to create blocks based on the first three characters of the last name and the zip code. Last name changes due to marriage or divorce and zip code changes from families that move from one neighborhood to another would have the potential to cause an unacceptable amount of false negative matches. The only immutable fields in the set are SSN, birth date and gender. Any of the other fields may change over time. All have the potential for errors.

Sorted neighborhood blocking is similar to standard blocking, except that it sorts the data set on a sort key selected from the attributes and slides a *w* record size window over the data set, selecting candidate pairs only from within the window. This method has the

same problems as standard blocking and if *w* is too small and the key has many records with an identical key value, than the window will miss candidate pairs. An example is if the first three characters of the last name are used, for the key "smi", which matches "Smith". In the 2000 Census, 2,376,206 out of 242,121,073 respondents had "Smith" as a last name [U. S. Census 2000]. If the window size is not large enough, viable candidates and true matches may be missed. A single character transposition error "Simth" would be completely missed as would any other single character error that affects the key value.

Bigram indexing converts blocking key values to bigrams, sorts them and inserts them into an inverted list with a threshold value between 0.0 and 1.0. The set of bigrams for "Smith" is {"sm", "mi", "it", "th"}. After sorting we have {"it", "mi", "sm", "th"}. The threshold removes a subset of bigrams from the set. If the threshold is 0.75, then 75% of the bigrams are retained and 25% are removed generating 4 blocks, one block for each permutation of 3 members selected from the bigram set: {"it", "mi", "sm"}, {"it", "mi", "th"}, {"it", "sm", "th"} and {"mi", "sm", "th"}. Other strings that have matching bigram blocks will be assigned to one or more of these blocks. This substantially increases the required string pair comparisons that are necessary for standard blocking but can result in more blocks. This method has been found to be resilient for handling small errors in data [Cohen and Richman 2002]. However, substitution and transposition errors will hurt candidate generation on small strings. Consider "Smith" and "Simth" with bigrams {"si", "im", "mt", "th"} and a 0.75 threshold: {"im", "mt", "si"}, {"im", "mt", "th"}, {"im", "si", "th"} and {"mt", "si", "th"}. The "Simth" blocks only match one, "th", of those for "Smith". Bigram indexing is not resilient to transposition errors

for relatively short strings because the effects of transposition or substitution errors are multiplied in the bigrams. In this case, three of four bigrams are changed by a single transposition but the threshold only removes one bigram from each set. Deletion and insertion errors will change the number of bigrams and also affect two bigram values.

Canopy clustering with tf-idf uses tf-idf as a measure of distance on bigram tokens to create canopy clusters. A random record is selected and other records are clustered with it based on a loose distance threshold. The random record and any records within a tight threshold are removed from the cluster and the process is repeated a number of times. If errors exist in key values, potential matches will be ignored.

With all of these blocking methods, the selection of reliable blocking keys is the most important factor for their success. For example, blocking on the SSN would create a large number of very small blocks and blocking on gender would create two very large blocks. SSN, birth date and gender are considered immutable strings or fields because they do not change over time. The mutable fields, names, addresses and phone numbers, would not be reliable for blocking because, if used, candidate pairs may be mismatched. Mutable fields should be treated as variable and not as constant data fields because of their potential to change over time.

2.3 Comparing and Matching

A common and straight-forward way to compare two candidate records is to compare each record's field-by-field string distance value using some string distance metric. For

records *q* and *r*, the fields from the each are used to generate pair-wise comparison features, which are assembled into a comparison vector. Table 1 below shows two records and the comparison vector produced by the approximate match rule "the fields do not differ by more than 1 edit operation", which can identify approximately 80% of data entry errors [Damerau 1964]. The comparison vector in the bottom row shows 1 for an exact or approximate match field, 0 for unknown due to a missing value and -1 for an unmatch field. The comparison vectors are passed to the Match Candidates step.

Table 1: Two compared records and their resulting comparison vector

	First Name	Last Name	Address	Phone	Gender	Birth Date	SSN
Rec 1	John	Doe	1234 N Broad	2155551234	M	08205245	123456789
Rec 2	John	Smith	1234 S Broad	2675559876	M	06151978	
Comp Vec	1	-1	1	-1	1	-1	0

The process to match candidates uses the comparison vector to decide whether or not a record pair matches. Records that have a sufficient number of similar fields that fulfill a quantitative or rule-based requirement are said to match. For a deterministic RL system, points can be assigned to fields based on importance or uniqueness and summed to a threshold; or a set of rules can require that certain fields match. There are also probabilistic models that can be applied to RL.

2.4 Field-level Comparison

The methods discussed below are either phonetic or string distance matching algorithms. These field-level comparators are used to generate each field's pairwise agreement entry in the comparison vector.

2.4.1 Field Comparison with Phonetic Matching

Phonetic comparison of character strings is performed by converting the string into a phonetic code, which is used as an index to group homophonic strings with minor spelling differences into classes. The phonetic algorithms basically map single or multiple characters in a pattern string s to single or codes that create a phonetic code string s', which is typically used as an exact match with the codes t' of strings t in a dictionary T, where $s' = t'$. The following are descriptions of the most common phonetic code and index methods that are applicable to RL.

The Russell Soundex

An early phonetic string matching method called the Soundex was developed and patented by Robert C. Russell and Margaret K. Odell in 1918. This method was intended to be an improvement on indexing systems to phonetically accommodate spelling errors and inconsistencies for names. This phonetic indexing scheme is commonly referred to as "Soundexing." Russell's reasons for developing the Soundex were stated on his patent:

"There are certain sounds which form the nucleus of the English language, and those sounds are inadequately represented merely by the letters of the alphabet, as one sound may sometimes be represented by more than one letter or combination of letters, and one letter or combination of letters may represent two or more sounds. Because of this, a great many names may have two or more different spellings which in an alphabetic index, or an index which separates names according to the sequence of their contained letters in the alphabet, necessitates their filing in widely separate places." [U.S. Patent 1,261,167 1918]

The list below shows how Russell separated characters onto phonetic categories. Each category is represented by a number.

1. The vowels or "oral resonants":	a, e, i, o, u, y
2. The labials and labio-dentals:	b, f, p, v
3. The gutterals and sibilants:	c, g, k, q, s, x, z
4. The dental-mutes:	d, t
5. The palatal-fricative:	l
6. The labio-nasal:	m
7. The den to or lingua-nasal:	n
8. The dental fricative:	r

Figure 2: Russell Soundex Description

The *Russell Soundex* algorithm parses a name and generates a code beginning with the first letter in the name followed by three digits. The algorithm below demonstrates the rules for generating a Russell Soundex code. If two compared strings have the same code, they are considered to be a phonetic match.

1. Convert all characters to upper case
2. The first character in the string is the first character in the code
3. Drop the combination "GH"
4. Drop 'S' or 'Z' if the last character in the string
5. {A, E, I, O, U, Y} → 1
6. {B, F, P, V} → 2
7. {C, G, J, K, Q, S, X, Z} → 3
8. {D, T} → 4
9. {L} → 5
10. {M} → 6
11. {N} → 7
12. {R} → 8
13. Double adjacent code characters like "44" are reduced to '4'
14. Other characters (H and W) are ignored
15. Only the first occurrence of a vowel from category 1 is recorded
16. The code is padded with zeros if there are less than three numbers in the code: "C###"

Figure 3: Russell Soundex Mapping

The Russell Soundex code for "Michael" would be "M135" and "Bobby" would be "B120".

The American Soundex

The *American Soundex* [Knuth 1973] is a variant of the Russell Soundex and is the most popular phonetic algorithm. It is a standard function in many database systems, including Oracle [Oracle 2012], Microsoft SQL Server [Microsoft 2012], MySQL [MySQL 2012] and PostgreSQL [PostgreSQL 2012]. It was applied to in 1930's U.S. Census data from 1890 to 1920 to phonetically link entities and is currently in use today. The primary differences with the original Soundex are:

- All vowels are ignored unless the first character in the string
- The combination "GH" is not dropped
- The characters 'S' and 'Z' are not dropped as the last characters
- The characters 'M' and 'N' are merged to one category

The official variant of the Soundex used by the U.S. government is maintained by the National Archives and Records Administration (NARA) [NARA 2007]. The algorithm below shows the rules for generating the American Soundex code for a string.

1. Convert all characters to upper case
2. The first character in the string is the first character in the code
3. {B, F, P, V} → 1
4. {C, G, J, K, Q, S, X, Z} → 2
5. {D, T} → 3
6. {L} → 4
7. {M, N} → 5
8. {R} → 6
9. Double adjacent code characters like "44" are reduced to '4'
10. Other characters are ignored
11. The code is padded with zeros if there are less than three numbers in the code: "C###"

Figure 4: American Soundex Mapping

The code for "Michael" would be "M240" and "Bobby" would be "B100".

Refined Soundex

Some suggested refinements from the Apache Commons Codec to the Soundex are shown below [Apache Commons 2012]. The modifications help to create a greater number of smaller size groups to hedge against false positive matches.

1. Convert all characters to upper case
2. The first character in the string is the first character in the code
3. {B, P} → 1
4. {F, V} → 2
5. {C, K, S} → 3
6. {G, J} → 4
7. {Q, X, Z} → 5
8. {D, T} → 6
9. {L} → 7
10. {M, N} → 8
11. {R} → 9
12. Double adjacent code characters like "44" are reduced to '4'
13. Other characters are ignored
14. The code is padded with zeros if there are less than three numbers in the code: "C###"

Figure 5: Refined Soundex Mapping

The code for "Michael" would be "M370" and "Bobby" would be "B100".

Daitch-Mokotoff Soundex (D-M Soundex)

The *Daitch-Mokotoff Soundex*, also called the D-M Soundex, variant was developed by Jewish genealogists Gary Mokotoff and Randy Daitch in 1985 to provide better accuracy with Yiddish and Slavic names [Avotaynu 2012]. Mokotoff states that the D-M Soundex has the following improvements over the conventional American Soundex:

1. Information is coded to the first six meaningful letters rather than four.
2. The initial letter is coded rather than kept as is.
3. Where two consecutive letters have a single sound, they are coded as a single number.
4. When a letter or combination of letters may have two different sounds, it is double coded under the two different codes.

5. A letter or combination of letters maps into ten possible codes rather than seven.

The rules for the D-M Soundex are:

1. Names are coded to six digits, each digit representing a sound listed in the Coding Chart below.
2. The letters A, E, I, O, U, J and Y are always coded at the beginning of a name, as in Augsburg (054795). In any other situation, they are ignored except when two of them form a pair and the pair comes before a vowel, as in Breuer (791900), but not Freud. The letter "H" is coded at the beginning of a name, as in Halberstadt (587943) or preceding a vowel as in Mannheim (665600), otherwise it is not coded.
3. When adjacent sounds can combine to form a larger sound, they are given the code number of the larger sound, as in Chernowitz, which is not coded Chernowi-t- z (496734) but Chernowi-tz (496740).
4. When adjacent letters have the same code number, they are coded as one sound, as in Cherkassy, which is not coded Cherka-s-sy (495440) but Cherkassy (495400). Exceptions to this rule are the letter combinations "MN" and "NM" whose letters are coded separately, as in Kleinman which is coded 586660 not 586600.
5. When a name consists of more than one word, it is coded as if one word, such as Nowy Targ, which is treated as Nowytarg.
6. Several letters and letter combinations pose the problem that they may sound in one of two ways. The letter and letter combinations CH, CK, C, J and RZ (see chart below), are assigned two possible code numbers. Be sure to try both possibilities.
7. When a name lacks enough coded sounds to fill the six digits, the remaining digits are coded "0" as in Berlin (798600) which has only four coded sounds (B-R- L-N).

Figure 6: Rules for D-M Soundex

This variant produces a 6 character code based on the transformations in the table below.

Table 2: D-M Soundex Code Mapping

Letter combinations found	At the beginning	After a vowel	Else
AI, AJ, AY, EI, EY, EJ, OI, OJ, OY, UI, UJ, UY	0	1	
AU	0	7	
IA, IE, IO, IU	1		
EU	1	1	
A, UE, E, I, O, U, Y	0		
J	1	1	1
SCHTSCH, SCHTSH, SCHTCH, SHTCH, SHCH, SHTSH, STCH, STSCH, STRZ, STRS, STSH, SZCZ, SZCS	2	4	4
SHT, SCHT, SCHD, ST, SZT, SHD, SZD, SD	2	43	43
CSZ, CZS, CS, CZ, DRZ, DRS, DSH, DS, DZH, DZS, DZ, TRZ, TRS, TRCH, TSH, TTSZ, TTZ, TZS, TSZ, SZ, TTCH, TCH, TTSCH, ZSCH, ZHSH, SCH, SH, TTS, TC, TS, TZ, ZH, ZS	4	4	4
SC	2	4	4
DT, D, TH, T	3	3	3
CHS, KS, X	5	54	54
S, Z	4	4	4
CH, CK, C, G, KH, K, Q	5	5	5
MN, NM		66	66
M, N	6	6	6
FB, B, PH, PF, F, P, V, W	7	7	7
H	5	5	
L	8	8	8
R	9	9	9

The code for “Moskowitz” would be “645740” and “Auerbach” would be “097500”.

Cologne Phonetic (Kölner Phonetik)

The *Cologne Phonetic* was developed and published by Hans Joachim Postel in 1969 primarily for Germanic names. It maps strings to phonetic codes in a process that is similar to the Soundex and its variants [Postel 1969]. The codes are not limited to 4 or 6 characters as in the Soundex methods. However, multiple adjacent matching code characters are reduced to one character and vowels are ignored, even if they are the first character in the string. The characters ‘H’ and ‘J’ are ignored. The rules for generating a phonetic code are:

1. Convert string to upper case
2. Transform characters in string from left to right according to the conversion table.
3. Remove all multiple adjacent matching codes.
4. Remove all the code "0" except at the beginning.

Figure 7: Rules for Cologne Phonetic

The table below contains the transformations for the Cologne Phonetic.

Table 3: Cologne Phonetic Code Mapping

Letter	Context	Code
A, E, I, J, O, U, Y		0
H		-
B		1
P	not before H	
D, T	not before C, S, Z	2
F, V, W		3
P	before H	
G, K, Q		4
C	at onset before A, H, K, L, O, Q, R, U, X	
	before A, H, K, O, Q, U, X except after S, Z	
X	not after C, K, Q	48
L		5
M, N		6
R		7
S, Z		8
C	after S, Z	
	at onset except before A, H, K, L, O, Q, R, U, X	
	not before A, H, K, O, Q, U, X	
D, T	before C, S, Z	
X	after C, K, Q	

The code for “Brezhnev” is 17863.

NYSIIS

The New York State Identification and Intelligence System (*NYSIIS*) was a project headed by Robert L. Taft in 1970. Taft published a paper that compared the Soundex and NYSIIS, where he stated that NYSIIS was designed through rigorous empirical analysis (of names) [Taft 1970]. The algorithm is shown below.

1. Convert all characters to upper case
2. Perform these translations if they are the first characters in the string
   MAC → MCC
   PH → FF
   KN → NN
   PF → FF
   K → C
SCH → SSS
3. Perform these translations if they are the last characters in the string
   EE → Y
   IE → Y
   {DT, RT, RD, NT, ND} → D
4. The first character of the code is the first character of the string
5. Translate remaining characters
   EV → AF
   {A, E, I, O, U} → A
   Q → G
   Z → S
   M → N
   KN → N
   K → C
   SCH → SSS
   PH → FF
   If previous or next character is a consonant, H → ‘’
   If previous character is a vowel, W → A
6. Translate last characters in string
   S → ‘’
   AY → Y
   A → ‘’
7. The code can optionally be truncated to 6 characters

Figure 8: NYSIIS Mapping

Metaphone

The *Metaphone* was published by Lawrence Phillips in 1990 as an alternative phonetic index to the Soundex. The original Metaphone algorithm uses 16 symbols, “0BFHJKLMNPRSTWXY”, to represent consonant sounds. The ‘0’ (zero) represents the “th” sound and the ‘X’ represents the “ch” or “sh” sounds. All other symbols

represent the typical English pronunciation [Philips 1990]. The algorithm below shows the transformation rules for the Metaphone.

1. Convert all characters to upper case
2. Drop all duplicate adjacent characters except for 'C' (i.e. CC → CC)
3. If the string begins with the following, perform these transformations
   KN → N
   GN → N
   PN → N
   AE → E
   WR → R
End-If
4. Replace 'C' as
   SCH → SKH
   CIA → XIA
   CH → XH
   CI → SI
   CE → SE
   CY → SY
   C → K
5. Replace 'D' as
   DGE → JGE
   DGE → JGE
   DGY → JGY
   DGI → JGY
   D → T
6. Replace 'G' as
   If "GH" is not at the end of the string or before a vowel
      GH → H
   End-If
   If not at end of string
      GN → N
      GNED → NED
   End-If
   GI → JI
   GE → JE
   GY → JY
   G → K
7. If 'H' follows a vowel and is not before a vowel, 'H' → ''

Translate the following
CK → K
PH → F
Q → K
V → F
Z → S
8. Replace 'S'
SH → XH
SIO → XIO
SIA → XIA
9. Replace 'T'
TIA → XIA
TIO → XIO
TH → 0 (Zero)
TCH → CH
10. If "WH" is at the beginning of the string
WH → W
End-If
11. If 'W' does not precede a vowel
W → ''
End-If
12. If 'X' is at the beginning of the string
X → S
Else
X → KS
End-If
13. If 'Y' does not precede a vowel
Y → ''
End-If
14. Remove all vowels except those at the beginning of the string

Figure 9: Metaphone Mapping

Caverphone

The *Caverphone* was developed by David Hood at the University of Otago for the Caversham Project to match data in old and new electoral lists for New Zealand pronunciation of names in 2002. It was intended to match electoral rolls for the 19th and early 20th centuries, when names were recorded in a "commonly recognizable form"

[Caversham 2012]. The algorithm below shows the rules to generate the code from a string.

1. Convert string to lowercase
2. Remove characters not a-z
3. If the name begins with
   cough → cou2f
   rough → rou2f
   tough → tou2f
   enough → enou2f
   gn → 2n
4. If the name ends with
   mb → m2
5. Perform transformations
   cq → 2q
   ci → si
   ce → se
   cy → sy
   tch → 2ch
   c → k
   q → k
   x → k
   v → f
   dg → 2g
   tio → sio
   tia → sia
   d → t
   ph → fh
   b → p
   sh → s2
   z → s
   any initial vowel → A
   all other vowels → 3
   3gh3 → 3kh3
   gh → 22
   g → k
   groups of the letter ‘s’ → S
   groups of the letter ‘t’ → T
   groups of the letter ‘p’ → P

groups of the letter ‘k’ → K
groups of the letter ‘f’ → F
groups of the letter ‘m’ → M
groups of the letter ‘n’ → N
w3 → W3
wy → Wy
wh3 → Wh3
why → Why
w → 2
any initial ‘h’ → A
all other occurrences of h → 2
r3 → R3
ry → Ry
r → 2
l3 → L3
ly → Ly
l → 2
j → y
y3 → Y3
y → 2

6. Remove
   2 → ‘’
   3 → ‘’
7. Put six 1s on the end
8. Take the first six characters as the code

Figure 10: Caverphone Mapping

2.4.2 Field Comparison with String Distance Metrics

There are a variety of string distance metrics from simple algorithms, such as Hamming Distance and Jaccard Distance to more elaborate dynamic programming solutions as is used in the Levenshtein edit distance and its variants.

Hamming Distance

Hamming Distance (Ham) [Hamming 1950] is the de facto standard for comparing fixed-length numeric strings. Its strength is in its simplicity. It has an $O(n)$ complexity, where

$n = min(|s|, |t|)$. Its weakness is its inability to detect insertion, deletion and transposition errors. Given the algorithm below, the Hamming distance between strings "Smith" and "Simth" would be two, which would result in only a 60% similarity for a single transposition error.

Algorithm 1: Ham(s, t)

Input: *s*, *t*: strings of characters
Output: *n*: 32-bit integer
Begin
  Step 1: Check for empty or different length strings
  if $|s|$ = 0: **return** -1 **end-if**
  if $|t| \neq |s|$: **return** -1 **end-if**
  Step 2: Calculate distance
  $n \leftarrow 0$
  for i = 1 to $|s|$
    if $s(i) \neq t(i)$
      $n \leftarrow n + 1$
    end-if
  end-for
  return *n*
end

Jaro

The *Jaro Distance* [Jaro 1989] is a string similarity metric that defines a distance measure normalized between 0 and 1, to determine if two strings, *s* and *t*, are likely to match. The Jaro distance is defined as:

Equation 10: Jaro Distance

$$d_{Jaro} = \frac{1}{3}\left(\frac{m}{|s|} + \frac{m}{|t|} + \frac{m - x}{m}\right)$$

where *m* is the number of matching characters and *x* is half the number of transpositions. Two characters are said to match if their distance is less than:

Equation 11: Jaro Match

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

Consider an example comparison: *s* = "Smith" and *t* = "Simth". The number of matching characters $m = 5$ because both strings have the same 3 characters 'S', 't' and 'h' in the same positions and the characters "m" and "i" are less than $d_{Match} = 1.5$ character positions apart. Half the number of transpositions $x = 0.5$ because there is one transposition: "mi" to "im". Then $d_{Jaro} = 0.967$.

Jaro-Winkler

Jaro-Winkler Distance [Winkler 1990] is an extension of Jaro that awards higher scores for longer matching prefix lengths *l* using a predetermined prefix scale *p*. The Jaro-Winkler distance is defined as:

Equation 12: Jaro-Winkler Distance

$$d_{Wink} = d_{Jaro} + lp(1 - d_{Jaro})$$

Consider the previous strings and that $p = 0.2$. Then:

$$d_{Wink} = 0.967 + 1 * 0.2 * (1 - 0.967) = 0.968$$

The score is slightly higher due to the matching first character. If the strings were "Smith" and "Smiht", the score would be 0.987 because the matching prefix is three characters: "Smi".

Jaccard Similarity and Distance

The Jaccard Similarity Coefficient [Jaccard 1901] is a statistical measure for comparing the difference and similarity between sets of entities. Similarity is measured as the intersection of sets divided by the union of sets. The sets for string similarity being the

strings *s* and *t* and the elements in the sets are their respective characters. Jaccard Similarity is defined as:

Equation 13: Jaccard Similarity

$$J(s,t) = \frac{|s \cap t|}{|s \cup t|}$$

Jaccard Difference is defined as:

Equation 14: Jaccard Difference

$$J_\delta(s,t) = 1 - J(s,t) = \frac{|s \cup t| - |s \cap t|}{|s \cup t|}$$

For the "Smith" and "Simth" example, we have $|"Smith" \cap "Simth"| = 5$ and $|"Smith" \cup "Simth"| = 5$:

$$J("Smith","Simth") = \frac{5}{5} = 1$$

$$J_\delta("Smith","Simth") = 1 - J("Smith","Simth") = \frac{5-5}{5} = 0.0$$

For "Smith" and "this", we have:

$$J("Smith","this") = \frac{4}{5} = 0.8$$

$$J_\delta("Smith","this") = 1 - J("Smith","this") = \frac{5-4}{5} = 0.2$$

Jaccard does well with transposition errors because all the characters are the same. Jaccard's weakness is that it ignores the order and frequency of characters. Consider that the strings "Smith" and "this" would be considered an 80% match. Both Jaccard and Hamming have been used as a pretest or filter to increase edit distance comparison due to their low computational complexity [Arasu, Ganti and Kaushik 2006].

Dice's Coefficient

Dice's Coefficient [Dice 1945] is a statistical measure of sets and is similar to Jacquard except that the function ranges (0, 1). Dice's Similarity is defined as:

Equation 15: Dice's Coefficient

$$D_s(s,t) = \frac{2|s \cap t|}{|s| + |t|}$$

Dice's Difference is defined as:

Equation 16: Dice's Difference

$$D_d(s,t) = 1 - \frac{2|s \cap t|}{|s| + |t|}$$

If we compare "Smith" and "Simth", we have:

$$D_s("Smith","Simth") = \frac{2*5}{5+5} = 1$$

$$D_d("Smith","Simth") = 1 - \frac{2*5}{5+5} = 0.0$$

For "Smith" and "this", we have:

$$D_s("Smith","this") = \frac{2*4}{5+4} = 0.89$$

$$D_d("Smith","this") = 1 - \frac{2*4}{5+4} = 0.11$$

Tversky Index

The Tversky Index [Tversky 1977] is an asymmetric measure of similarity that is useful for information retrieval. The function returns a number (0, 1) and is defined as:

Equation 17: Tversky Index

$$S(s,t) = \frac{|s \cap t|}{|s \cap t| + \alpha|s - t| + \beta|t - s|}$$

where, $\alpha, \beta > 0$ are parameters that assign weights to the magnitude of difference between sets. Consider parameters $\alpha = 0.5$ and $\beta = 0.5$, then:

$$S(\text{"}Smith\text{"},\text{"}this\text{"}) = \frac{4}{4 + 0.5*1 + 0.5*0} = \frac{4}{4.5} = 0.89$$

With these parameters, Tversky will consistently produce the same results as Dice but require more computation.

Overlap Coefficient

The Overlap Coefficient computes the ratio of overlap of two sets given the minimum size of the two sets.

Equation 18: Overlap Coefficient

$$O(s,t) = \frac{|s \cap t|}{\min(|s|,|t|)}$$

For “Smith” and “this”, we have:

$$O(\text{"}Smith\text{"},\text{"}this\text{"}) = \frac{4}{4} = 1$$

N-Gram/Q-Gram

N-Grams, also referred to as Q-Grams, are a statistical or probabilistic string comparison technique that segments strings into *n*-size (or *q*-size) parts for efficient processing. If *n* =1, the parts of the string are referred to as unigrams. For *n* = 2, they are referred to as bigrams and for *n* = 3, trigrams, and so on. Bigrams and trigrams are typically used for RL string comparison. We previously discussed bigram indexing in the section on

blocking. We will describe trigrams with prefix and suffix markers for approximate string field comparison for RL.

The set of trigrams for “Smith” is {“--s”, “-sm”, “smi”, “mit”, “ith”, “th-”, “h--”}. The hyphens are the prefix markers that show the beginning of the string (“--s” means the string begins with ‘s’) and the suffix markers that mark the end of the string (“h--” means the string ends with ‘h’). A lexicographical sort will place a hyphen (ASCII decimal 45) before all characters (uppercase A is 65 and lowercase a is 97). The sort for “Smith” is {“--s”, “-sm”, “h--”, “ith”, “mit”, “smi”, “th-”}. The sorted trigrams for “Simth” are {“--s”, “-si”, “h--”, “imt”, “mth”, “sim”, “th-”}. If these sorted trigrams are matched, as shown below, only three of the seven trigrams match, yielding a 3/7 = 0.43 match ratio. If prefixes and suffixes are ignored, the ratio decreases to 0/7 = 0.0. This shows the same problem noted in the blocking section that n-grams are not especially useful for matching relatively short strings for RL field-level string comparison.

{“**--s**”, “**-sm**”, “**h--**”, “**ith**”, “**mit**”, “**smi**”, “**th-**”}

{“**--s**”, “**-si**”, “**h--**”, “**imt**”, “**mth**”, “**sim**”, “**th-**”}

Figure 11: Q-Grams for "Smith" and "Simth"

Levenshtein Distance

Levenshtein Distance is commonly referred to as edit distance. For effective string similarity measurement, edit distance has an advantage over other metrics because it considers the ordering of characters and allows nontrivial alignment [Xiao, Wang and Lin 2008]. 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 [Levenshtein 1966]. 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”. The main problem with the *DL* algorithm is its complexity: *O*(*mn*), where *m* and *n* are the lengths of the compared words. This can require significant computation for comparisons of very large datasets.

Algorithm 2: Lev(s, t)

Input: *s*, *t*: strings of characters
Output: *n*: 32-bit integer
m: $|s| + 1 \times |t| + 1$: array of 32-bit integers
Begin
  Step 1: Check for empty strings
  if $|s|$ = 0: **return** $|t|$ **end-if**
  if $|t|$ = 0: **return** $|s|$ **end-if**
  Step 2: Create distance matrix d
  for i = 0 to $|s|$: *m*(*i*,0) ← *i* **end-for**
  for j = 1 to $|t|$: *m*(0,*j*) ← *j* **end-for**
  Step 3: Calculate distance matrix
  for i = 1 to $|s|$
    for j = 1 to $|t|$
      if $s(i-1) = t(j-1)$
        $m(i,j) \leftarrow m(i-1, j-1)$
      else
        $d(i,j) \leftarrow min(d(i-1,j),$
          $d(i,j-1), d(i-1,j-1)) + 1$
      end-if
    end-for
  end-for
  return *m*($|s|$, $|t|$)
end

Figure 12 contains the Levenshtein computation 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 Levenshtein. 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.

		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 12: Calculation of Levenshtein distance

Damerau-Levenshtein Edit Distance

Damerau-Levenshtein (*DL*) edit distance extends Levenshtein distance to also detect transposition errors and treat them as one edit operation. Approximately 80 percent of data entry errors can be corrected using a one character substitution, one character deletion, one character insertion or a transposition of two characters [Damerau 1964]. The algorithm below shows DL for character strings (Damerau’s contribution to *DL* is underlined):

Algorithm 3: $DL(s, t)$

Input: *s*, *t*: strings of characters
Output: *n*: 32-bit integer
m: $|s| + 1 \times |t| + 1$: array of 32-bit integers
Begin
  Step 1: Check for empty strings
  if $|s|$ = 0: **return** $|t|$ **end-if**
  if $|t|$ = 0: **return** $|s|$ **end-if**
  Step 2: Create distance matrix d
  for i = 0 to $|s|$: $m(i,0) \leftarrow i$ **end-for**
  for j = 1 to $|t|$: $m(0,j) \leftarrow j$ **end-for**
  Step 3: Calculate distance matrix
  for i = 1 to $|s|$
    for j = 1 to $|t|$
      if $s(i-1) = t(j-1)$

```
            m(i,j) ← m(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 m(|s|, |t|)
end
```

The two computation matrices in Figure 13 below demonstrate the subtle differences in computing Levenshtein (Left) and Damerau-Levenshtein (Right). The strings "SUNDAY" and "SUDNAY" were processed using both algorithms and only differ by a single transposition error. The differences in computation are shown in bold red. At the second intersection of 'D' and 'N', Damerau detects the transposition and does not add an extra point. This quality is important for both alphabetic and numeric strings. The most likely error that occurs when recording Social Security Numbers and phone numbers is a single transposition error.

		S	U	D	N	A	Y
	0	1	2	3	4	5	6
S	1	0	1	2	3	4	5
U	2	1	0	1	2	3	4
N	3	2	1	1	1	2	3
D	4	3	2	1	**2**	**3**	**4**
A	5	4	3	2	2	**2**	**3**
Y	6	5	4	3	3	**3**	**2**

		S	U	D	N	A	Y
	0	1	2	3	4	5	6
S	1	0	1	2	3	4	5
U	2	1	0	1	2	3	4
N	3	2	1	1	1	2	3
D	4	3	2	1	**1**	**2**	**3**
A	5	4	3	2	2	**1**	**2**
Y	6	5	4	3	3	**2**	**1**

Figure 13: Calculation of Levenshtein (left) and Damerau-Levenshtein (right) distances

2.4.3 Optimizations for Edit-Distance Comparison

Approximate string matching is much more computationally expensive than exact string matching and has been a hot research topic for decades. The most common and effective way to compare strings is the Damerau-Levenshtein (*DL*) edit distance algorithm, which counts the number of characters that must be changed to convert a string s into another string t. Equation 19 below shows the computation for each element in the *DL* matrix, which is based on the concept of dynamic programming. The first line determines that two compared characters are equal and simply copies the previously computed distance. The second line tests for a transposition error and adds 1 to the distance two steps back, which is Damerau's contribution to the original Levenshtein edit distance [Levenshtein 1966]. The third line adds 1 to the distance if a substitution, insertion or deletion error is detected.

$$
\begin{aligned}
&If\ s_j = t_i, C_{j,i} \leftarrow C_{j-1,i-1} \\
&Else\ If\ s_{j-1} = t_i\ AND\ s_j = t_{i-1}, C_{j,i} \leftarrow C_{j-2,i-2} + 1 \\
&Else\ C_{j,i} \leftarrow min\big(C_{j-1,i-1} + 1, C_{j-1,i} + 1, C_{j,i-1} + 1\big)
\end{aligned}
$$

Equation 19: DL computation

These edit operations include character substitutions, insertions, deletions and transpositions. This algorithm has a computational complexity of $O(mn)$, meaning that for two compared strings, s and t, where $m = |s|$ and $n = |t|$, the algorithm requires mn time and space to compute the edit distance. It takes more than 9 ½ hours just to compare the complete list of 151,671 last names (surnames) that occur 100 or more times from the 2000 U.S. Census [U.S. Census 2000]. Given the complexity of *DL*, it would take a very long time to pairwise compare every street address in the country. Addresses are longer strings and every address in the country is a much longer list. There are almost 550,000 street addresses in the City of Philadelphia alone. Approximate matching aims to find all

strings in a set T for a given pattern s that match for k or less edits, where $k \leq DL(s,t)$. String pairs that have a k or less edit distance are said to be k-related. Any optimization for edit distance must guarantee that it does not violate the k-relation between strings that approximately using *DL*. There are some optimizations available to speed up *DL* for approximate string matching. Some of these are filtering, neighborhood generation [Myers 1994], pruning [Wang, Feng and Li 2010], hashing [Knuth 1997] and tries [Apostolico and Galil 1985].

Prefix Pruning

Prefix Pruning is based on the premise that 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 [Gusfield 1997]. The threshold can be used to force an early termination if $d_{i,*} > k$ [Sakoe and Chiba 1990]. The implementation, called *Pruning-Damerau-Levenshtein* (*PDL*), below is similar to an edit distance *Prefix Pruning* method for Trie-based string similarity joins [Wang, Feng and Li 2010]. 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 $d_{i,j} \leftarrow min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) + 1$ because d is initially zeros. The algorithm for *PDL* differs from the algorithm in [Wang, Feng and Li 2010] because it has been modified to perform *DL* instead of the standard Levenshtein distance and it returns a Boolean value.

Algorithm 4: $PDL(s, t, k)$

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
$m \leftarrow |s|$
$n \leftarrow |t|$
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: $d_{i,0} \leftarrow i$ **end-for**
for $j = 1$ *to* n: $d_{0,j} \leftarrow j$ **end-for**
Step 3: Calculate distance matrix
for $i = 1$ *to* m
$x \leftarrow 0$
if $i < k + 1$: $d_{i,i-k-1} \leftarrow 1000$
for $j = max(i - k, 1)$ *to* $min(i + k, n)$
if $s_{i-1} = t_{j-1}$
$d_{i,j} \leftarrow d_{i-1,j-1}$
else
$d_{i,j} \leftarrow min(d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}) + 1$
if $i > 1$ *and* $j > 1$
if $s_{i-1} = t_{j-2}$ *and* $s_{i-2} = t_{j-1}$
$d_{i,j} \leftarrow min(d_{i,j}, d_{i-2,j-2} + 1)$
end-if
end-if
end-if
if $d_{i,j} \leq k$: $x \leftarrow x + 1$ **end-if**
end-for
if $j < n$: $d_{i,j} \leftarrow 1000$
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 Figure 14. 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 14: Prefix Pruning DL (PDL) distance matrix

Filter and Verify Methods

In the 1990s, methods to decrease data comparison using edit distance called "*filter and verify*" methods were introduced. Basically, filtering uses a computationally cheap method to filter out pairwise comparisons that have no chance of matching using the more expensive *DL*. 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 [Navarro 2001]. One of the earliest known and most common methods, Length Filtering [Gravano and Ipeirtis 2001]. The idea is based on the fact that two strings, s and t, cannot differ by k or less characters or edit operations if the difference in their lengths is 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 [Gravano and Ipeirtis 2001] is defined as shown in Algorithm 5:

Algorithm 5: 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

Given a function to compute string length, $L(s, t)$, we filter as shown below as length filtered *DL* (*LDL*). If the first term, $L(s, t) \leq k$, fails the second, $k \leq DL(s, t)$, is not called, which saves unnecessary computation because it takes much less work to count characters than to perform *DL*.

$$if\big(L(s,t) \leq k \; AND \; k \leq DL(s,t)\big), M \leftarrow (s,t)$$

Equation 20: Composition of Length Filter and DL

Neighborhood Generation

Neighborhood generation [Myers 1994] generates all approximate strings for a pattern s given an alphabet Σ and perform exact matching, usually using an index or binary search, to find all approximate matches in a list T. Consider that the alphabet $\Sigma = \{a, b, c\}$ and the pattern $s = aba$. There will be exactly three deletion matches $\{ba, aa, ab\}$, nine insertion matches $\{aaba, baba, caba, abba, acba, abaa, abca, abab, abac\}$, and six substitution matches $\{bba, cba, aaa, aca, abb, abc\}$. One can easily see that the processing and number of neighbors generated can explode for larger alphabet $|\Sigma|$ and

longer strings $|s|$. The strings generated are called the *k*-neighborhood of s and are denoted as $N_k(s) = \{x \in \Sigma^*, DL(x,s) \leq k\}$ [Navarro, Baeza-yates, Sutinen, and Tarhio 2000]. The growth of the *k*-neighborhood is estimated to be $|N_k(s)| = O(m^k|\Sigma|^k)$ [Ukkonen 1985].

Hashing

Hashing converts strings into a numeric code that arranges a list of strings T into a hash table V of buckets each of which contains strings that may be spelled similarly. Consider that for a function $h()$ that converts strings into hash codes, we have $u = h(s)$ for a search pattern s and a search pattern hash code u, a hash table V, where $\forall v \in V$ and $\forall t \in T, v = h(t)$ and a function $H(u, V)$ that finds candidate matches C from the bucket codes in V. The exact match string will always be in the same bucket because if $s = t$ than $u = v$. If the hash coding method is well designed for approximate matching, if $s \approx t$ than $u \approx v$. Hashing methods increase approximate matching performance because they substantially decrease the search space by selecting candidate hash buckets that may have some similarity with the search pattern and ignoring those that do not. Hashing will be discussed further in subsequent topics.

Signature Hashing

Signature Hashing is referred to as a simple folklore approximate dictionary search optimization method—meaning that it is not really published in the literature but is reasonably well known in the domain [Boytsov 2011], though hashing itself is extensively covered [Knuth 1997]. The strings *s* from dictionary set *S* are converted to numeric codes, representing hash buckets *H*, via some hash function $h()$ to reduce the size of the alphabet by mapping $\Sigma \rightarrow \mathfrak{s}$, where $|\Sigma| < |\mathfrak{s}|$, such that $s' = h(s)$. An example

of a hash map operation is shown in Figure 15 below. Individual characters from the left-hand-side are mapped to the right-hand-side, basically reducing the size of the alphabet from 26 to 13 characters by grouping characters in pairs and storing them in binary signatures. This reduces the search space to 2^{13} (8,192) buckets. Without the reduction, 2^{26} (67,108,864) buckets would be needed to store un-hash coded binary signatures.

a→{ab}	b→{ab}
c→{cd}	d→{cd}
…	
w→{wx}	x→{wx}
y→{yz}	z→{yz}

Figure 15: Sample Hash Map

The mapping in Figure 16 below shows the hash code for the name "JAMES", which would map to bucket number 597 (binary 0001001010101= decimal 597). Any recurring characters are dropped and if 'A' and 'B' (or any other grouped pair) are present they only flip one bit. This scheme would result in a one bit difference between the words "MISS" (0001001010000 = 592) and "MISSISSIPPI" (0001011010000 = 720) because the {OP} bit would be '1' in the latter.

YWU**S**QO**M**KIG**E**C**A**

ZXVTRPNL**J**HFDB

000**1**00**1**0**1**0**1**

Figure 16: 13-bit Hash Code for "JAMES"

Each $t \in T$ will hash to one-and-only-one bucket. If two strings are assigned to the same bucket because they have the same hash code, they are said to have had a collision. This problem is handled by chaining or allowing more than one string index in a bucket by using an array to contain the indices of all t_i that hash to any given code. The hash codes

are the indices to the buckets represented by arrays of candidate matches. For a match pattern string s, an exact match in the hash table, $t = s$, will be in the bucket that matches the hash code for s, $s' = h(s)$, if it is a member of T. To find approximate matches for s that differ by k edits, the binary signatures of the hash table are scanned to find all signatures that differ by less than or equal to $2k$ bits. Any $t \in T$ in the buckets that are returned are candidates for an approximate match of k or less edits. [Boytsov 2011]

Full Neighborhood Generation

A *Full Neighborhood Generation* generates all strings within k edits of a pattern string s, which is referred to as a k -neighborhood, which is denoted as $N_k(t) = \{s \in \Sigma^*, DL(t,s) \leq k\}$. Each of the strings s in the k -neighborhood is exact matched with the dictionary set T. The major drawback is potential size of the generated neighborhood. Consider the full 1-neighborhood for "JAMES":

The string "JAMES"

4 single transpositions

5 single character deletions

6 x 26 single character insertions

5 x 25 single substitutions

This is 291 elements in the 1-neighborhood list. A rough estimate for this growth is $O(n^k|\Sigma|^k)$ [Ukkonen 1993]. The full 2-neighborhood for "JAMES" has approximately 16,900 elements and the full 3-neighborhood would have nearly 2.2 million elements. The growth of the space and computation is dependent on the number of edits k, the size of the string m, and the size of the alphabet Σ. The value in this method is that candidates can be generated, albeit in large sets for greater values of k, m, and $|\Sigma|$ [Boytsov 2012].

Reduced Alphabet Neighborhood Generation Hashing

Reduced Alphabet Neighborhood Generation Hashing uses a character-by-character based coding to generate a reduced neighborhood set *C* of candidate strings by converting string $s \in S$ into their projections $s' = h(s)$. If the alphabet $\Sigma = \{A, B, C, \dots, X, Y, Z\}$ is reduced to a binary alphabet $\mathfrak{s} = \{0,1\}$ using the character mapping $\forall s_i \in s$, $h(s_i) = 0$ if $s_i \in [a, m]$ and $h(s_i) = 1$ if $s_i \in [n, z]$. Then for $s' = h(s)$, where $s =$ "JAMES", $s' =$ "00001". The string "JAMES" and all other strings with the code "00001" would be assigned to the same bucket. All binary strings that differ by k or less bits are calculated and all k-related strings should be in these buckets. The character mapping is The reduced 1-neighborhood for "JAMES" would be:

The string "00001"

1 single transposition

5 single character deletions

6 x 2 single character insertions

5 x 2 single substitutions

This substantially reduces the size of the neighborhood. This is reduced set *C* has 29 elements and is used to find string candidates $t \in T$ to be processed with a more elaborate edit-distance to choose string pairs, where edit-distance $DL(s, t) \leq k$.

Ed-Join: An Efficient Algorithm for Similarity Joins with Edit Distance Constraints

In [Xiao, Wang and Lin 2008], the authors offer a new method for similarity joins based on studying mismatching *q*-grams by analyzing their locations and contents. Their method exploits mismatch-based filtering algorithms, uses positional q-grams and a probing window over q-gram signatures to decrease the size of candidate sets. The

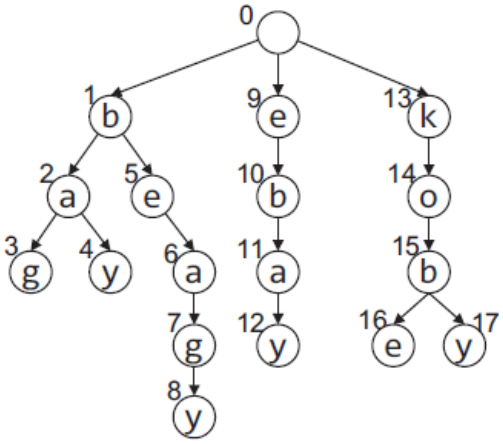
method has very impressive performance results for both efficiency and effectiveness. However, as with previous methodologies, this method joins or concatenates all fields into one string, which substantially decreases the amount of information contained within compared records. Also, it would not be practical to use this on relatively short fields, such as first name and last name, because single edit errors can increase the potential for false negatives as was shown in the discussion about blocking.

Trie-Join: Efficient Trie-based String Similarity Joins with Edit Distance Constraints

Special purpose approximate string matching tree structures, referred to as "tries", have been used to facilitate DL type searching. In [Wang, Feng and Li 2010], 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 is based. An interesting algorithm is their Prefix Pruning Edit Distance algorithm. It decreases the run-time of edit distance by restricting computation of elements a distance k from the diagonal and introducing an early termination protocol. We had developed a similar method but Prefix Pruning outperformed it. We modified this algorithm, which is described in Prefix Pruning, above.

The Trie-Join method combined the concept of the aforementioned prefix pruning and a tree structure to index strings that approximately match from a dictionary, list or data store. Figure 17 below shows a trie and the data used to create the tree structure. If a node in the trie refers to a string in the list, it will contain a pointer to it. These methods typically perform a depth first search to find all approximate matches in the trie. The

downside to these methods is that there can be up to |Σ| pointers from each node at each level to cover the whole alphabet and pointers back to the string list. These structures typically require at least nine times the memory that it takes to store the list. Consider that the string "bag" requires three bytes and the trie entry requires thirty-nine bytes. These methods are fast but will require too much memory for very large data.



A sample data set

SID	String
S_1	bag
S_2	ebay
S_3	bay
S_4	kobe
S_5	koby
S_6	beagy

Figure 17: Prefix trie

2.4.4 Overview of String Comparison and Optimization

Figure 18 below shows the taxonomy of the described phonetic matching algorithms based on their applications to names from various ethnic groups. Most of these algorithms are designed to cover names from a specific ethnicity or group of ethnicities. The Russell Soundex was designed for Anglo-Saxon names because most names in the U.S. were within this ethnicity in the late 1800s. Presently, the U.S. has a much more ethnically diverse population. Only the second and third generations of the Metaphone claim to be designed for a large group of ethnicities.

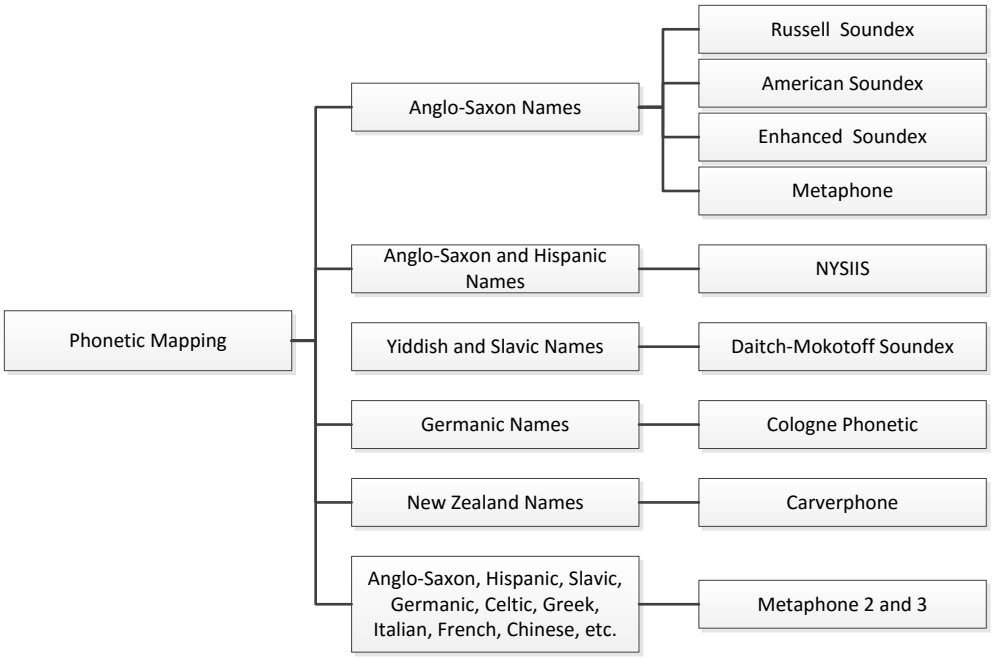


Figure 18: Taxonomy of phonetic algorithms

Figure 19 below shows the taxonomy of string distance measures and optimizations based on the primary matching methodology. Edit distance is the number of characters that differ between compared strings. Proximal similarity checks for matching characters in compared strings that are near each other. Set operations consider strings as sets of characters and perform various set comparisons to determine likeness or difference. Exact match methods generate a neighborhood of candidate strings and us an exact match to find matching entries in a list or dictionary. Substring methods cut strings into groups of characters and match the substrings. Filtering uses a computationally efficient method to generate a subset of candidates that may match. Indexing use a fast match to avoid comparing all strings in a list or dictionary.

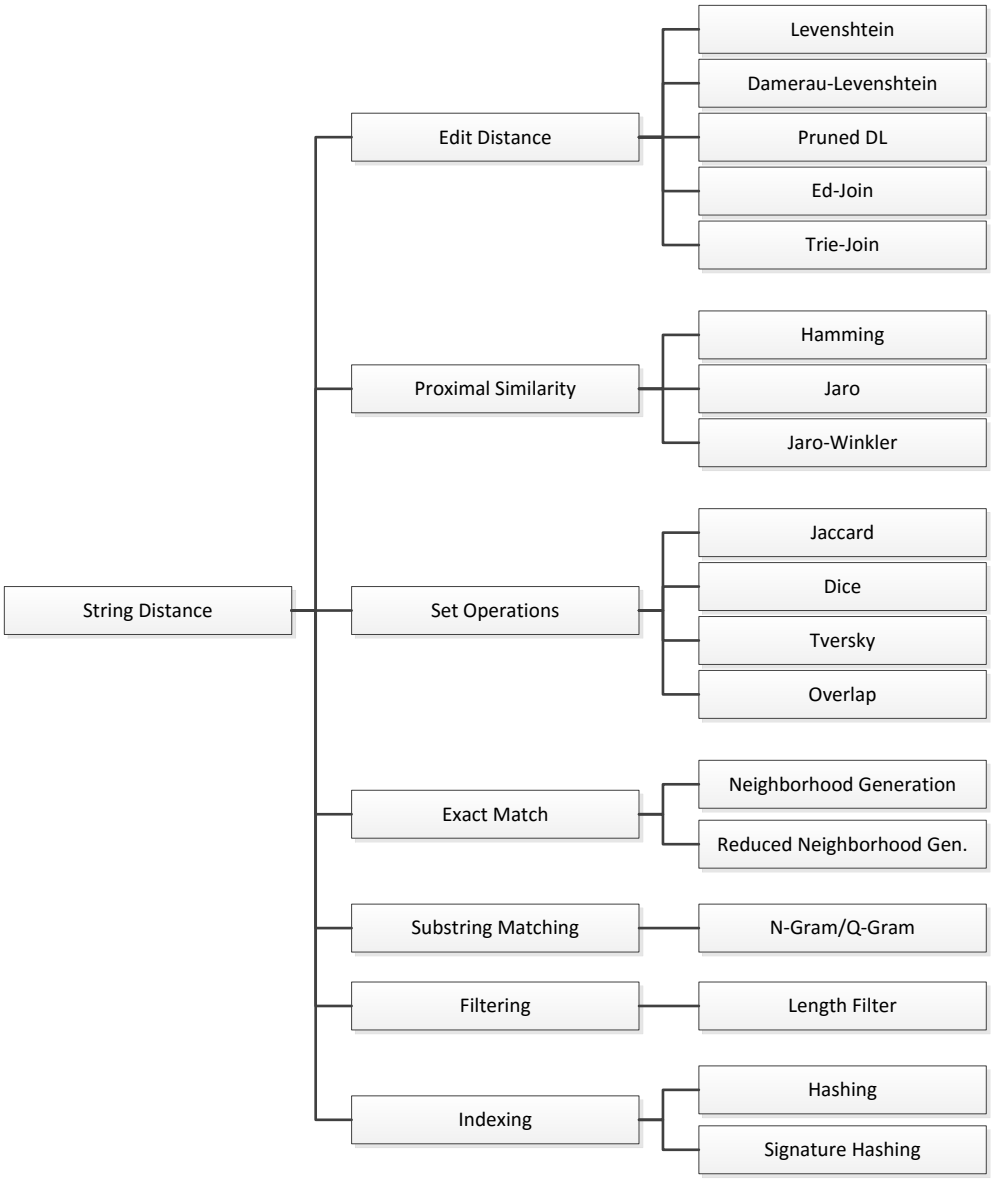


Figure 19: Taxonomy of string distance metrics and optimizations

2.5 Survey of Existing RL Methods

We consider eight examples of RL applications that were developed by academia and government health agencies.

2.5.1 FRIL (2008)

FRIL is an acronym for fine-grained record integration and linkage [Jurczyk, Lu, Xiong, Cragan and Correa 2008a] [Jurczyk, Lu, Xiong, Cragan and Correa 2008b]. The FRIL project was a cooperative effort between the Mathematics and Computer Science Dept. at Emory University and the National Center on Birth Defects and Developmental Disabilities at the Centers for Disease Control and Prevention. The FRIL system uses a set of tools described in toolboxes such as TAILOR [Elfeky, Verykios, and Elmagarmid 2002] but unlike other offerings, some described below, FRIL allows a fine tuning of the match system by allowing the user to choose various search methods, distance functions for record similarity and decision models. Figure 20 below shows the FRIL architecture. The search methods are algorithms that determine which records to compare. The two search methods in FRIL are nested loop join (NLJ), an exhaustive search that compares all possible pairs, and the sorted neighborhood method (SNM), which sorts the records in databases Q and R on some relevant attributes and only compares records that fall within some fixed-size windows, ω_Q for database Q and ω_R for database R that are advanced down the records in each database. This is a blocking method as mentioned above. It avoids the exhaustive search and should substantially decrease pairwise comparisons. FRIL also facilitates schema mapping to normalize discrepancies between databases by allowing a user to identify attributes that map from one source to another. This is referred to as attribute selection and mapping (ASM) and the set of attributes is denoted as Φ and each element is $a \in \Phi$

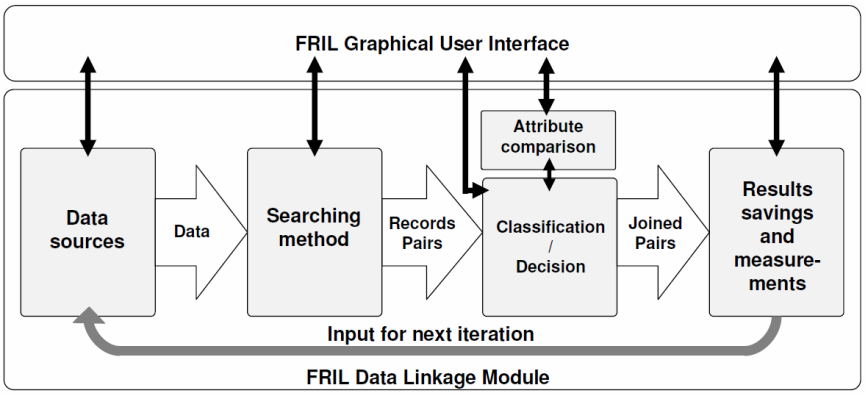


Figure 20: FRIL Model

FRIL has four distance functions that a user can choose, edit distance, Soundex, Q-gram and equality, called the distance function selection (DFS). The user can indicate a threshold for acceptance or rejection of a string pair using a simple probabilistic fuzzy logic, referred to as attribute scoring (AS). Equation 1 below shows a function m_{fa} for calculating a string match from a distance function f_a. The tunable parameters min_{fa}, the threshold for a full match, and max_{fa}, the threshold for a full unmatch, are entered by the user on [0, 1]. If a string pair is not a full match or unmatch, a probability value on [min_{fa}, max_{fa}] is calculated.

Equation 21: FRIL String Matching

$$\begin{aligned} m_{fa} &= 1 \; if \; f_a(s_1, s_2) \leq min_{fa} \\ &= 0 \; if \; f_a(s_1, s_2) > max_{fa} \\ &= \frac{max_{fa} - f_a(s_1, s_2)}{max_{fa} - min_{fa}} \; otherwise \end{aligned}$$

If the distance function were edit distance, say between strings “AARON” and “ARON”, the edit distance is 1, *fa* is 0.2 (calculated as the edit distance divided by the longest string length 1/5 = 0.2), and if min_{fa} is 0.5, m_{fa} = 1.0. FRIL also allows a weight to be assigned to each attribute based on its expected value for record matching within its decision

model, referred to as an attribute weighting (AW) the weight is denoted as $\alpha_a \in [0,1]$. The last step in the decision model is record scoring (RS). The score function is:

Equation 22: FRIL Record Scoring Function

$$score(q_i, r_j) = \frac{\sum_{a \in \Phi} \alpha_a m_{fa}\left(\pi_a(q_i), \pi_a(r_j)\right)}{\sum_{a \in \Phi} \alpha_a}$$

Note that π is tha standard projection operator. The user can select tunable parameters max_t, the minimum score for a match, and min_t, the upper limit of an unmatch condition. The match condition M is evaluated as:

Equation 23: FRIL Match Condition

$$\begin{aligned} M(q_i, r_j) &= 1 \; if \; score(q_i, r_j) \geq max_t \\ &= 0 \; if \; score(q_i, r_j) < min_t \\ &= \frac{score(q_i, r_j) - min_t}{max_t - min_t} \; otherwise \end{aligned}$$

The FRIL system was specifically designed to match birth certificates with a database recording birth defects. Specifically, there were 1.25 million birth certificates of children born in the time frame of 1997 to 2006 for over 12,700 cases of birth defects for the MACDP program. The selected attributes for their linkage experiments included those listed in Table 4. The two columns contain the percentage of non-null fields in the attribute columns.

Table 4: MACDP Selected Attributes

	MACDP	Birth cert. data
Birth date baby	100.0%	100.0%
Name baby	100.0%	100.0%
Birth date mother	100.0%	100.0%
Name mother	100.0%	100.0%
Birth date father	82.0%	83.0%
Name father	82.0%	83.0%
Hospital #	100.0%	99.5%
City	100.0%	97.5%
Zip code	100.0%	99.8%
Gender baby	100.0%	100.0%

The team used precision and recall as their measure of accuracy. Table 5 below shows the results and tuned parameters for 4 experiments. During the experiments, the user tuned the FRIL's parameters to increase accuracy. The results show that varying the tunable parameters can substantially increase precision and recall.

Table 5: MACDP Results

Experiment	Precision	Recall
1	0.95	0.86
2	0.98	0.85
3	0.99	0.85
4	0.99	0.95

2.5.2 Febrl (2008)

Freely Extensible Biomedical Record Linkage (Febrl) [Christen 2008a] [Christen 2008b] [Christen and Churches 2003] is an open source RL system developed by the ANU Data Mining Group at the Australian National University. This system has a GUI interface and recently developed advanced techniques for data cleaning and standardization, blocking, field comparison and record pair classification. The system also has some useful data exploration tools. The authors state that the system is appropriate for data sets that contain up to several hundred thousand records. The system has an impressive

collection of phonetic and distance-based field comparison functions, including DL, Jaro-Winkler, NYSIIS, Double Metaphone and Soundex. It has record matching functions based on the traditional Fellegi and Sunter model [Fellegi and Sunter 1969], a supervised Optimal Match method, K-Means and Farthest First [Goiser and Christen 2006] classifiers, a Support Vector Machine classifier [Chang and Lin 2001] and the Two Step classifier [Christen 2007]. The system also includes RL evaluation tools, such as accuracy, precision, recall, F-measure AOC and ROC.

2.5.3 Link Plus (2007)

Link Plus was developed by the Cancer Division at the Centers for Disease Control and Prevention [Thoburn, Gu and Rawson 2007]. It was developed as a linkage tool for cancer registries to support the CDC's National Program of Cancer Registries. The system was designed after a careful review of the literature going back to 1969 and performs probabilistic RL based on the theoretical work of Fellegi and Sunter [Fellegi and Sunter 1969] and uses the EM [Dempster, Laird and Rubin 1977] algorithm to estimate parameters from Fellegi and Sunter's model. The system allows the user to select blocking variables to partition the set of all enumerable record pairs and decrease the search space. Link Plus uses the Soundex and NYSIIS for phonetic matching. The system has a GUI to facilitate the configuration of the matching system and to import datasets. It has a collection of matching methods [Link Plus Guide 2007]:

- Exact—character-for-character match
- Last and First Name—incorporates value specific and NYSIIS and treats hyphenated names as two individual strings
- Jaro-Winkler Metric

- Middle Initial
- SSN—Uses a partial match and transposed character match. Also has a 9-to-4-digit conversion
- Date—Incorporates partial matching and a weighted method to provide higher weights to better matches
- Value-Specific (Frequency-Based)—Adjusts weights for matched based on frequencies of values. For first names, "James" is more frequent than "Alonzo" and would pay a weight penalty. Also done with race value.
- Generic String—Uses the Levenshtein edit distance function for typographical errors.
- Zip Code—Compares 9 and 5-digit Zip Codes.

Link Plus also allows the user to define special strings used to mark missing values. The system also has useful reporting tools to analyze matching results. There was no mention of experiments in the documentation to measure the quality of RL.

2.5.4 The Link King (2005)

The Link King [Campbell 2005] is an SAS/AF application for record linkage and deduplication that was developed at the Washington State Division of Alcohol and Substance Abuse. The documentation immediately states that administrative datasets have inherent discrepancies due to inconsistencies in the data. Some of the problems linking databases that may lead to incomplete or inaccurate linking from different sources are:

- Use of nicknames

- Hyphenated names
- Misspelled names
- Transposed SSN digits
- Transposed date fields

The Link King incorporates both probabilistic and deterministic record linkage methods to increase the linkage power of the application. In order to match records, the Link King requires a first name, last name, and either a SSN or birthdate. This system also performs gender field imputation for fields with a missing gender value using a lookup table and the first name of the entity to be matched. The system uses SAS's Spedis function for approximate spelling match, and the Soundex and the New York State Intelligent Information System (NYSIIS) for phonetic matching. It also has a nickname identification method that uses a lookup table to identify names that are synonymous. The system also has a user interface that provides access to facilitate importing data and matching. The problem with this application is that it is no longer available for download and the Website, www.the-link-king.com, is no longer active. Some of the features in this system, especially the ability to process aliases and the requirement that either SSN or birthdate need be present for matching are very important considerations for RL.

2.5.5 LinkWiz (2001)

This system was developed by LinkageWiz Inc. and offered as a trial version commercial record linkage and deduplication analysis application [LinkageWiz 2009]. There is no documentation for the system's methodology or how the system generates matches. The Website for the system, www.linkagewiz.com is no longer active. We only include a reference to this application because it has been mentioned in the literature.

2.5.6 Record Linkage as DNA Sequence Alignment Problem

In [Hong, Yang, Kang and Lee 2008], the authors use the popular BLAST (Basic Local Alignment Search Tool) gene sequence alignment algorithm to perform RL. This method assumes that all the data for the records are contained in single fields, i.e. "John Doe, 110 E. Foster Ave. State College, PA, USA". While their method is well designed and has very promising results, it may not be appropriate for RL on administrative datasets that contain demographic data. Combining all the fields into one will decrease the information in the record by removing the context or meaning of the individual fields. This method seems very appropriate for problems such as comparing references and keyword lists.

2.5.7 EARL: an Evolutionary Algorithm for Record Linkage

In [Cusmai, Aiello, Scannapieco, and Catarci 2008], the authors formulate RL as a multi-objective optimization problem and introduce an evolutionary programming based solution. They comment that probabilistic models have problems working with large datasets and knowledge-based techniques rely on the presence of training data. They state that evolutionary algorithms are appropriate because they are stochastic, adaptive and intrinsically parallel in their search space. The process is comprised of two steps: formulation of the initial population and the application of the Non-dominated Sorting Genetic Algorithm-II (NSGA-II). The strengths of this algorithm are it is a one-shot algorithm with a linear complexity runtime, and that it does not depend on prior knowledge like probabilistic or training sets as in knowledge based models. One problem is that for any given record $q \in Q$ it only allows for one match $r \in R$ where $q = r$, allowing only one match between two datasets. For many administrative datasets, this may not be the case. This problem will be revisited in the Future Directions section.

2.5.8 Swoosh: a generic approach to entity resolution

In their paper [Benjelloun, Garcia-Molina, Menestrina, Su, Whang and Widom 2008], the authors discuss new methods that they have developed for RL but more importantly they discuss some important mathematical properties:

- For a domain of records $\mathcal{R}$,
- an instance $I = \{r_1, \dots, r_n\}$ is a finite set of records from $\mathcal{R}$ or a set of records to perform match and merge functions on,
- a Boolean match function *M* over $\mathcal{R} \times \mathcal{R}$ to determine if records match the same real-world entity,
- a partial function merge μ that maps $\mathcal{R} \times \mathcal{R}$ into $\mathcal{R}$ to aggregate matched records.

They define a merge closure of I denoted $\bar{I}$ as the smallest set of records S such that: $I \subseteq S$ and for any records $r_1, r_2 \in S$, if $r_1 \approx r_2$ then $\langle r_1, r_2 \rangle \in S$. They determine that $\bar{I}$ exists and is unique. They extend a partial preorder (because it is reflexive and transitive) or domination on instances by definition: Given two instances I_1, I_2, we say that I_1 is dominated by I_2, denoted $I_1 \preccurlyeq I_2$ if $\forall r_1 \in I_1, \exists r_2 \in I_2$ such that $r_1 \preccurlyeq r_2$. An entity resolution of I is a set of records I', in which $I' \subseteq \bar{I}$, $\bar{I} \preccurlyeq I'$ and no strict subset of I' satisfies conditions either of these conditions. They determine that an entity resolution I' exists and is unique.

Their new methods are modeled by mathematical properties that they refer to as ICAR and merge dominance. The basis for ICAR is:

1. *Idempotence*: $\forall r, r \approx r$ and $\langle r, r \rangle = r$. A record always matches itself, and merging it with itself still yields the same record.

2. *Commutativity*: $\forall r_1, r_2, r_1 \approx r_2$ Iff $r_2 \approx r_1$, and if $r_1 \approx r_2$, then $\langle r_1, r_2 \rangle = \langle r_2, r_1 \rangle$.
3. *Associativity*: $\forall r_1, r_2, r_3$ such that $\langle r_1, \langle r_2, r_3 \rangle \rangle$ and $\langle \langle r_1, r_2 \rangle, r_3 \rangle$ exist, $\langle r_1, \langle r_2, r_3 \rangle \rangle = \langle \langle r_1, r_2 \rangle, r_3 \rangle$.
4. *Representativity*: If $r_3 = \langle r_1, r_2 \rangle$, then for any r_4 such that $r_1 \approx r_4$, we also have $r_3 \approx r_4$.

Merge dominance $\leq$ is and order on records in which the following monotonicity conditions hold:

(A) For any records r_1, r_2 such that $r_1 \approx r_2$, it holds that $r_1 \leq \langle r_1, r_2 \rangle$ and $r_2 \leq \langle r_1, r_2 \rangle$, i.e. a merge record always dominates the records it was derived from;

(B) if $r_1 \leq r_2$ and $r_1 \approx r$,.then $r_2 \approx r$, i.e. the match function is monotonic;

(C) if $r_1 \leq r_2$ and $r_1 \approx r$,.then $\langle r_1, r \rangle \leq \langle r_2, r \rangle$, i.e. the merge function is monotonic;

(D) if $r_1 \leq s$, $r_2 \leq s$ and $r_1 \approx r_2$, then $\langle r_1, r_2 \rangle \leq s$.

The authors propose that if merge dominance, which is a canonical domination order, exists and that *M* and μ meet the monotonicity requirements above, than $\leq$ can be substituted with $\preccurlyeq$. If *M* and μ are reflexive and commutative, then the ICAR properties are also satisfied. The matching function *M* may not be transitive in that for $r_1 \approx r_2$ and $r_2 \approx r_4$ it is possible that $r_1 \not\approx r_4$. The important finding is that matching records $r_1 \approx r_2$ can be merged $\langle r_1, r_2 \rangle$ and that by aggregating records, bridges can be made between record matches that may have been missed because it will be the case that $r_3 \approx \langle r_1, r_2 \rangle$ and $r_3 \approx r_4$. This means that it is possible to force a transitive closure on the RL process, which is very desirable according to [Benjelloun, Garcia-Molina, Menestrina, Su, Whang and Widom 2008] and [Dong, Halevy and Madhavan 2005]. The authors acknowledge

that redundancy in matching is a problem in RL and their new methods show that it can be eliminated using bookkeeping to keep track of previous comparisons that did not match, which is also desirable, and makes the process much more efficient.

2.6 Similar Work in Other Domains or with Differing Methodologies

The work discussed in this section is within the Record Linkage field but differs from our research. It is discussed solely to show that RL is applicable to other problems and has other modalities. Some of the work below postdates our work (2009 for first version of ALIM) and is being mentioned as support and also to differentiate our work from other work that may seem superficially similar.

Temporal Data Handling

In [Li, Dong, Maurino and Srivastava], the authors discuss the existence of temporal data in terms of DBLP authors. The data contains the name of the author, co-authors, affiliation and the year of publishing. The year is used as a timestamp and the authors apply time decay and clustering. Time decay basically decreases the indication that two records match as the amount of time between them increases. Their temporal clustering compares the records to match authors. Our temporal data does not include the timestamp. However, it includes all data that has changed over time (i.e. all addresses, phone numbers, etc.). Our solution aims to leverage this temporal data to increase match efficiency and effectiveness by aggregating all known data about an entity and matching the aggregations between entities.

Internet Data Sources

In [Bleiholder, Khuller, Naumann, Raschid and Wu 2006], the authors discuss the topic of querying life science sources on the Web. They state that the data is an "interconnected…large and complex graph, and there is an overlap of objects in the

sources." The WWW is basically a graph of data. It makes sense to treat this as a graph traversal problem. Our data is not stored on the Web and is not able to be traversed in tuple form. Our proposed solution uses graph to model the data and create aggregate records to handle the temporal data problem. Traversing in ALIM involves the relationships between data and records—it's not just a path from source to source. In [Bleiholder, Naumann 2008], the authors discuss data fusion for Internet sources. They mention that challenges of data fusion are "uncertain and conflicting data values" and its goals are "complete, concise and consistent data". Our data has significant missing values, conflicting values from temporal changes, data entry errors and multiple entries for many entities. Complete, concise and consistent are unreachable goals with this data. In [Rezig, Dragut, Ouzzani and Elmagarmid], the authors use an iterative caching approach to match online records. It uses the Bed tree index to approximate match string data.

Use of Graphs

[Ravikumar and Cohen 2004] and [Kardes, Konidena, Agrawal, Micah Huff, Sun 2013] also use graphs to facilitate record linkage. However, these methods use graphs in the decision model—not in the data model. The ALIM insert method converts data tuples into a data graph. This allows for temporal multi-data approximate record matching. In [Wenfei, Fan, Tian, Dong 2015], the authors use graphs with vertices (data) and keys (edges) to find pairs of vertices that match. Vertices might be "Led Zeppelin", "Beatles", "Robert Plant" and "Hey Jude", and keys might be recorded_by and member_of. While this establishes data relations, it's not temporal. This scheme is slightly similar to our record relation methodology that forms relations as parent_of and sibling_of. In [Lunen

2012], the author uses graph mining to link sparse data for travel records by using places as vertices and people and dates as edges.

Summary of New Developments

The table below shows features of the methods in this comparison with respect to similarities and differences. The table basically shows considerable differences between ALIM and all other methods. These methods were designed for disparate purposes and share little in their structure or methodology. The primary purpose for ALIM is to leverage temporal data to increase the confidence of record linkage by linking data that is already known to match within constituent databases. The primary contribution from ALIM is the proposition that temporal data should be organized into graph-based multi-data records and matched in an iterative manner to increase the effectiveness of record linkage. The following choices were made for ALIM after a in depth study of real data from 11 databases for a local health department.

- We use a graph data model to represent and process temporal semi-structured evolving data

- We choose a blocking free approach because for every match decision we need a discrete step-by-step reason for the match

- Internet data sources were not considered because our data was structured database data but evolved into semi-structured data over time. Internet data is unstructured. Our work can be extended to unstructured data but this was not our original intention.

Table 5: Methods vs Features

Method/Features	Temporal Data	Graph-based	Database Data	Deduplication	Online Data	Edit Distance Optimization	Alias Matching	Eliminates redundant data and processing
ALIM (2009)	Processes temporal data but does not require time stamp	Uses graphs to relate data and records in the data model. Data may be alias or approximate match. Records may have familial or affinity relations.	Aggregates compound records in databases and matches the aggregated records to increase match confidence	Links records between databases and deduplicates at the same time	Not considered for this study. Problem involved linking enterprise databases.	Uses Probabilistic Signature Hashing to match 80% of minor spelling errors based Damerau-Levenshtein	Stores alias relations in graph loaded from alias list	Data graphs only record the first instance of a datum and other records refer to it
Bleiholder, Khuller (2006)	NA	Treats online data as graph and traverses from instance to instance. Focus is on paths between objects.	NA	NA	For use on online data	NA	NA	NA
Kardes, Konidena (2013)	NA	Use graphs to decide which records match using MapReduce	Yes but had to run on a 50 node hadup cluster	Can be used for linkage and deduplication	NA	NA	NA	MapReduce
Li, Dong (2011)	Processes temporal data but requires date stamp to relate matches. Temporal clustering.	NA	DBLP Authors dataset used. Structured data.	NA	NA	NA	NA	NA
Lunen (2012)	Uses dates associated with traveling to geographic locations in Britain, which was primarly used as a way to process data in time chunks	Uses a decision tree logic to match and uses a data graph with lodations as vertices and people moving from one location to another as edges	Steam Engine Makers' Society data containing sparse data about the travel of entities from one location to another	NA	NA	Uses Soundex and non-optimized Levenshtein edit distance	NA	NA
Ravikumar and Cohen (2004	NA	Uses graphs in the decision model	Used structured datasets and used a blocking	NA	NA	Use an unspecified distance metric	NA	Uses Blocking
Wenfei, Fan (2015)	NA	Use graphs with data as vertices and data relations as edges.	Data from various domains and knowledge bases	NA	NA	NA	NA	MapReduce

PART II

FINDINGS

CHAPTER 3

INTRODUCTION TO FINDINGS

We evaluated samples of a real health and social sciences database and RL system, currently used by a local health department and found that the data had some interesting properties. In our study, we identified two primary tasks that were required by the RL system: Real-time searches to provide reports about client histories, usually for a caseworker or investigator, and bulk list (of clients) matching, usually for administrative research or cohort studies. We have also found that many social scientists would like these systems to perform not only person-based but family based Record Linkage and social network analysis.

3.1 Evaluation of the CARES System

The City of Philadelphia Department of Health and Opportunities currently uses a system called CARES to perform RL operations across 11 social sciences related databases. The CARES matching algorithm is used to determine if two or more records retrieved from different data sources actually correspond to the same real world entity. This is critical in several important applications. The general method, within the domain, for determining whether two records r_1 and r_2 can be matched includes two steps. First, values in corresponding attributes/fields from r_1 and r_2 are matched. Typically, for each attribute α,

a similarity (or distance) between $r_1\alpha$ and $r_2\alpha$ is computed. Second, the similarities between value pairs under all attributes are aggregated to determine whether r_1 and r_2 are matched. The first step is accomplished by employing various value matching functions, most of which are approximate string matching functions such as edit-distance, n-gram, Jaro-Winkler and some phonic based functions [Elmagarmid, Ipeirotis and Verykios 2007]. The second step is carried out by employing record matching rules that can be derived manually, statistically, or by machine learning and/or classification techniques based on sample matched and unmatched record pairs. There have been many papers published about the entity identification problem (e.g., [Thor and Rahm 2007] [Wang and Madnick 1989] [Shu and Meng 2009] [Shen, DeRose, Vu, Doan and Rama-Krishnan 2007] [Shu, Meng and Yu 2007]) and a general survey of these methods can be found in [Elmagarmid, Ipeirotis and Verykios 2007].

The current CARES matching system consists of following two steps:

1. Field comparisons:
    a. Exact string and date match.
    b. Approximate phonic based attribute matching (Soundex) for names, and sum of digits for social security number matching given 5 matching digits.
2. Field-by-field scoring system for record matching similar to the previously described system.

The CARES system uses the following demographic data to match records:

1.	First Name
2.	Last Name
3.	Address

4.	Phone Number
5.	Gender
6.	Birthdate
7.	Social Security Number

Figure 21: CARES Demographic Fields

Table 6 below shows the non-missing data for one of the databases that are processed by CARES. Notice that 46% of SSNs, 47% of addresses and 58% of phone numbers are missing.

Table 6: CARES Missing Data

	# of occurrences	Percentage in total
All occurrences (demographics records)	5869873	
Non-blank first name	4936300	84%
Non-blank last name	4965268	85%
Non-blank ssn	3185454	54%
Non-blank dob	4441929	76%
Non-blank gender	4771010	81%
Non-blank phone	2436911	42%
Non-blank address	3092366	53%

Note:
1. Only active records (including old) are considered
2. The count for each demographic attribte is derived separately, no correlation is considered or implied.

In our preliminary experiments, we found that the Soundex had inherent problems with errors and inconsistencies in the data. If spelling errors or inconsistencies affected critical characters, characters that affect the generation of the Soundex code, this method would completely fail to identify approximate matched. If "JAMES" is misspelled as "KAMES" ('J' and 'K' are adjacent to each other on the keyboard), their Soundex codes are "J520" and "K520", respectively. Because Soundex matching only considers exact codes to match, a single character error can foil it for approximate spelling matching. There is a significant difference between the two problems: Identifying spelling errors and identifying phonetic matches. Both may be present in the data. Aliases are also a problem for the Soundex. Consider that "JAMES" and "JIM" are considered

synonymous. Their Soundex codes are “J520” and “J500”, which also do not match. We also found that the sum of digits was not a good approximate match for Social Security Numbers. This method is sometimes used because transposition errors are common with SSNs. We performed a histogram analysis of the sums of SSNs and found a high potential for false positive matches, even for exact matches. Consider that “170-44-9887” and “288-79-7007” have the same sums (48) and are completely different SSNs. We discuss our findings from the CARES evaluation and literature review below. For a more complete description of the preliminary CARES findings, please see [Jupin Prelim 1 2011].

3.2 Temporal Data and Compound Records

A very important property of the RL problem is that each client may not have only one record containing their demographic data but many clients may have many records that with differing demographic data throughout their lives. A synthetic example is shown below in Table 7. Every time a client received services and the client’s demographic information had changed, a new tuple may have been created. Notice that the first SSN differs from the rest. This is because SSNs are not issued at birth and the agency providing the service issued its own temporary ID for the SSN because SSN is used as the client ID. Every time a client moves to a new address, changes a phone number, gets married or divorced, or uses an alias (i.e. “Jim” for “James”), the client’s demographic data changes in the system.

Table 7: Data Records for Mary

Num	FirstName	LastName	Address	Zip	Phone	Birth	SSN	Gender
1	Mary	Smith	123 REED ST	19147	215-123-4567	1/1/1970	999-34-5678	F
2	Mary	Smith	123 REED ST	19147	215-123-4567	1/1/1970	123-45-6789	F
3	Mary	Smith	1326 S HANCOCK ST	19147	267-222-9876	1/1/1970	123-45-6789	F
4	Mary	Johnson	114 MORRIS ST	19148	267-222-9876	1/1/1970	123-45-6789	F
5	Mary	Williams	114 MORRIS ST	19148	215-543-6789	1/1/1970	123-45-6789	F
6	Mary	Williams	3604 POWELTON AVE	19104	215-543-6789	1/1/1970	123-45-6789	F
7	Mary	Smith	15146 ENDICOTT ST	19116	215-543-6789	1/1/1970	123-45-6789	F
8	Mary	Smith	15146 ENDICOTT ST	19116	267-987-6543	1/1/1970	123-45-6789	F
9	Mary	Brown	1033 E CHELTEN AVE	19138	267-987-6543	1/1/1970	123-45-6789	F

In the real data 46% of SSNs and 24% of birthdates were missing. If the Birth and SSN columns are removed from Table 7, the last record (9^{th}) matches the 8^{th} on only 3 fields (FirstName, Phone and Gender). It only matches the rest by 2 fields (FirstName and Phone). Neither condition gives much confidence to a match even though these records belong to the same entity. This is a disaster for blocking records. If the blocking keys are "The first 3 characters of the last name and the Zip code", the example above is scattered across 6 blocks for the same entity record set. It should also be considered that data entry errors in blocking keys will decrease their effectiveness. [Ravikumar and Cohen 2004] mentions that databases frequently contain multiple records that refer to the same entity. In [Li, Dong, Maurino and Srivastava 2011], the authors also note that data for entities can change over time. Their method uses time decay and temporal clustering techniques to link temporal records. This condition is found very frequently in our real data. However, we do not compare our proposed solution to this system because our dataset does not have timestamp data.

When the term "Record Linkage" was first introduced by Halbert L. Dunn in 1946 [Dunn 1946], he described it in terms of taking the pages or records of a person's life and assembling it into a single volume or "Book of Life". It may be the case that a social worker is attempting to use an RL system to create a summary of a client's whole life.

Table 8 shows the aggregation of data in Table 7 with redundant data removed. This compound record will match any individual record from Table 7 if all fields are used in the match. An effective RL model would exploit this property because it makes best use of the data to increase match quality. If each record is a page, the compound record is a chapter of someone's life experiences within a member agency within the enterprise and given the situation, it makes sense to assemble the chapters from the pages before assembling the volume.

Table 8: Aggregated Data Record for Mary

FirstName	LastName	Address	Zip	Phone	Birth	SSN	Gender
Mary	Brown	1033 E CHELTEN AVE	19138	215-123-4567	1/1/1970	123-45-6789	F
	Johnson	114 MORRIS ST	19148	215-543-6789		999-34-5678	
	Smith	123 REED ST	19147	267-222-9876			
	Williams	1326 S HANCOCK ST	19147	267-987-6543			
		15146 ENDICOTT ST	19116				
		3604 POWELTON AVE	19104				

3.3 Data Quality

Considering the demographic fields in Table 8, it should be noted that there are both mutable and immutable data in records. Fields that contain data related to SSN, birth date and gender should not change over time but may contain multiple distinct values due to data entry errors and inconsistencies. Another property that can be used to increase performance is to pre-disqualify records that may have too much missing data. Consider: a record is unmatchable if it cannot match itself under a given match rule. If the record can't match itself, it will not match any other record.

The RL system should be able to match commonly known aliases [Campbell 2005] because they are commonly used. The names “James”, “Jim”, “Jimmy”, and “Jamie” should be considered synonymous. In the explored real dataset, even the gender field has errors, inconsistencies—actually contradictions. In some cases, entities are recorded as both male and female. This condition should be considered the same as if the gender was missing. It is probable that some people use nicknames instead of proper names. Someone named Robert may use Rob, Bob or Bobby. The characteristics of nicknames differ based on ethnicity or culture. Western European nicknames are likely derived from the proper name (i.e. Bob for Robert and Pat for Patricia). Hispanic nicknames (called apodos) are more likely to be based on profession, affection or physical characteristics [Mexican Nicknames]. The relation between proper names and nicknames is not one-to-one and onto. The nickname Ron is shared by proper names Aaron, Cameron, Ronald and Veronica. Other nicknames for Aaron are Erin and Ronnie. Even the use of a nickname look-up table does not ensure perfect accuracy. It can potentially increase matching accuracy if properly incorporated into a record matching system. Simple string comparators will likely miss the relationships that exist between proper and nicknames. A partial list of 1230 proper-nickname pairs was constructed from various sources on the Internet for testing. This can be used as a look-up table to connect nicknames with proper names and as a method to substitute nicknames for proper names.

It has been found that approximately 2% of data are obsolete within 1 month of entry [Eckerson 2002].

3.4 SSNs as Universal Identifiers

As mentioned, SSNs may be missing from a significant number of records. Even if all SSNs are present, there are critical weaknesses in its design and use as a Universal Identifier (UID). In [Daechsel 1979a], the author outlines 5 necessary characteristics of a UID:

1. Uniqueness
2. Permanence
3. Universality
4. Availability
5. Economy

The author of [Kuch 1972] adds 2 more characteristics:

6. Self-checking
7. Memorability

Each UID must be unique to each individual and each individual must have only one UID. Though the SSN is intended to be unique, there have been more than 1 million instances of a person having multiple SSNs and the same SSN has been issued to more than one individual in a few instances [Daechsel 1979b]. A UID must be permanent throughout an individual's life. A UID must be universally recognized. The SSN is permanent and universally recognized. A UID must be available, meaning easy to access and communicate. The SSN is mostly available but is not if an individual loses their card or intentionally reports a false SSN. A UID must have little cost in issue and in use. The

SSN is fairly economical at 9 digits. The SSN is absolutely not self-checking. There is no way to validate an SSN without accessing an external database. Consider that approximately 70% of randomly generated 9-digit numbers are valid SSNs [Kuch 1972] by the Social Security Administrations rules to issue them. A UID should be easy to remember. SSNs are not necessarily memorable but most individuals can memorize their own SSN given a little time. Given the weaknesses of the SSN, it is reasonable to assume that there may be errors in client reported SSNs–also that data entry errors and missing data further complicate its use. Even though there is a high rate of missing SSNs in health and social sciences databases, the National Center for Health Statistics (NCHS) does consider it to be an important identifier. NCHS considers records to be linked if they have the same SSN, birthdate and gender [NCHS 2010].

3.5 Transitive closure

The transitive property suggests that may be the case that records may act as bridges to find more matches for a given entity. This idea is found in [Bhattacharya and Getoor 2004] and [Dong, Halevy and Madhavan 2005]. The authors suggest that a transitive closure can be imposed on data records that should match using an iterative scheme that merges matching records and re-matches to increase the likelihood of finding additional matched records that would have otherwise been ignored. This property has also been acknowledged in [Campbell 2005] but is handled by joining the match results. The method suggests that if record r_i matches record r_j and record r_j matches record r_k, then assign record r_i matches record r_k. The iterative scheme is more powerful because it can

use the multi-record based compound data to find matches that would otherwise be ignored. If record r_i matches record r_j but neither r_i or r_j matches r_k, the composite record $r_i r_j$ (r_i and r_j's data combined into a compound record, $r_i r_j = r_i \cup r_j$) may match r_k. This is important because the data in these demographic records change over time—it's actually a temporal data problem.

3.6 Family-Based Social Networks and Matching Context

[Bhattacharya and Getoor 2004] state that by examining the context of a record or relationships that a record has with other records, the accuracy of the linkage can be increased. Consider a match for any two of the records shown for Mary in Table 7. If in both records, Mary has a husband, John, then we can have a higher confidence that the two records match. If one record has a husband, John, and the other does not, if any of the husband's data matches from the one with the relationship to the other record for Mary without the relationship, we can also have a higher confidence. We can use wives, mothers, fathers, sons, daughters, sisters, brothers, cousins and grandparents to increase RL accuracy. [Ananthakrishna Chaudhuri and Ganti 2002] [Bilenko and Mooney 2003] have also taken this approach to RL.

3.7 The Soundex as an Phonetic Match

The major problems with the Soundex are sensitivity to single key data entry errors and high false positive and false negative rates as is shown in the following two genealogy papers. In [Stanier 1990], the Soundex missed 25% of true positives and had a 66%false negative rate. [Lait and Randell 1996] reports 60% false negative rate and only 37% of true positives were found. Many database management systems include the Soundex as an approximate string matching algorithm. The benefit is that it is computationally economical—with a low computational complexity. The cost is that it is not very accurate. Our research found that the Soundex is extremely susceptible to single key data entry errors, when they occur on a critical character—one that is used to generate the Soundex 4-digit code. Even though the Soundex has some real issues, it is still used as a phonetic match algorithm. As previously mentioned, many database management systems, such as Oracle, SQL Server, MySQL, etc., use the Soundex as a phonetic match algorithm.

3.8 Edit-Distance as a String Comparator

As mentioned, edit distance has an advantage for effective string similarity measurement over other metrics because it considers the ordering of characters and allows nontrivial alignment [Xiao, Wang and Lin 2008]. Because of the ability to detect transposition errors, the Damerau-Levenshtein edit distance algorithm has the ability to detect approximately 80 percent of manual data entry errors [Damerau 1964]. Combining

Prefix Pruning [Wang, Feng and Li 2010] with the Damerau-Levenshtein algorithm can increase performance for string similarity measurement by restricting the problem search space [Gusfield 1997] and by using a threshold value *k* to force an early termination of a comparison if the edit-distance can no longer meet the threshold but will still produce the same results of Damerau-Levenshtein [Sakoe and Chiba 1990].

3.9 CARES Final Report

The CARES final report [Jupin and Shi 2011] implemented a prototype graph-based RL system, originally called Aggregate-Link-Match (ALM), renamed to Aggregate Link and Iterative Match (ALIM) Prototype 1. ALIM was found to find more true positive matches and run approximately 12.5 times as fast as the original CARES method, currently in use by the City of Philadelphia Department of Health and Opportunities. The old prototypes are briefly mentioned in the Current Progress section and the latest prototype of ALIM will be discussed in the Proposed Methodology section.

PART III

METHODOLGIES, EXPERIMENTS AND RESULTS

CHAPTER 4

INTRODUCTION TO METHODOLGIES, EXPERIMENTS AND RESULTS

We have developed improvements in two interrelated domains: String Comparison and Record Linkage. Our latest development in string comparison is a method called the Probabilistic Signature Hash (PSH). This method combines the power of a collection of methods to find strings that approximately match and is mathematically proven to produce the same exact results as the Damerau-Levenshtein (DL) Edit distance. PSH is also experimentally proven to be up to 6000 times faster than DL. Our latest development in RL is Aggregate, Link and Iterative Match (ALIM) prototype 3. This in-memory graph based RL system uses PSH for fast approximate matching and produces better match results faster than a system currently used by a local health department. The sequence of developments for each system is described below.

We also discuss the first two ALIM prototypes and experiments. Our third optimization uses a probabilistic approach to create a more efficient hash function.

CHAPTER 5

STRING DISTANCE OPTIMIZATION

We have performed research and development on DL edit-distance optimization. We describe a sequence of 3 methods to optimize performance. Our first string comparison optimization is the Fast Bitwise Filter (FBF), which is based on the filter and verify approach, which achieved a 130 times speedup over DL when combined with a length filter (LFPDL described below). The second method is the Hybrid Bitwise Signature Hashing with Bitwise Neighborhood Generation, Length Filter, FBF and *PDL* composite (GLFPDL described below), which combines hashing filtering and pruning to obtain computational speedups up to 3000 times over DL. The third and latest method is the Probabilistic Signature Hash that combines GLFPDL with a probability optimized hash function, which decreases the number of entries in hash buckets, decreasing the number of candidate strings, increasing speedup 6000 times that of DL.

5.1 First Optimization Technique: Fast Bitwise Filter

Our first development for optimizing DL edit-distance called the *Fast Bitwise Filter* (FBF) [Jupin, Shi and Obradovic 2012]. The FBF is a filter and verify method that substantially increases 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

a length filter) than Damerau-Levenshtein edit distance and preserves all approximate matches (meaning no false negatives). 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 provides substantial performance gains to approximate string comparison in database, record linkage and deduplication data processing systems.

The Venn diagrams in Figure 22 below show string comparison using the set concept for some sample last names. The left diagram shows a deletion error for "SMITH" as "SMIT", which results in one character outside the intersection. The center diagram shows the substitution error for "SMITH" as "SNITH", a single edit difference that results in two characters outside of the intersection. The right diagram shows that the strings "SMITH" and "JONES" have eight characters outside the intersection and that these strings are significantly different.

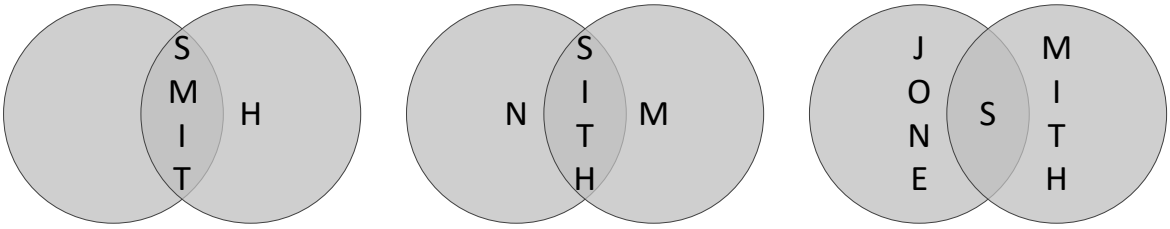


Figure 22: Venn diagrams on names

Our 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 [Microsoft MSDN]. The idea is

that the string's 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 Figure 23 and Figure 24.

The signature x for string s is actually a checklist of a subset of characters in s, where bit $x_0 = 1$ iff $'A' \in s$ and bit $x_1 = 1$ 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 or all digits (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?")

```
------ZY XWVUTSRQ PONMLKJI
       HGFEDCBA
00000000 00001100 00010001
       10000000
```

Figure 23: 32-bit alphabetic FBF bit signature for "SMITH"

```
--999888 77766655 54443332
       22111000
00000001 00000011 10000000
       11011011
```

Figure 24: 32-bit numeric FBF bit signature for "8005551212"

Consider that for a function to generate binary signatures $f()$ that $x = f(s)$ and $y = f(t)$ and a function $F(x, y)$ to compare signatures. The statement shown below demonstrates the use of FBF. We prove below that all strings x, y that differ by k edits are contained in the set whose signatures differ by $2k$ bits.

$$if\big(F(x,y) \leq\ 2k\ AND\ k \leq PDL(s,t,k)\big), M \leftarrow (s,t)$$

Equation 24: Composition of FBF and PDL

where M is a set of matched string pairs. Pruning furthur decreases the computation for *DL* by comparing prefixes or suffixes. Prefix pruning eliminates string pairs that differ by more than k characters when compared from the beginning to the end of the respective strings and suffix pruning from the last to the first. The comparison stops using an early termination when the k difference threshold is no longer possible. We developed our own flavor of the Prefix Pruning method that is combined with *DL* (*PDL* shown in Algorithm 4) and a more limited search space to increase edit distance performance.

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.

5.1.1 Generating FBF Signatures

Algorithm 6, $\text{SetAlphaBits}(s)$, 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 $\Sigma_c(...\Sigma_c)^j \in s$, where $\Sigma = \{A, B, C, ..., X, Y, Z\}$. Upon completion of SetAlphaBits(s), the following condition is true: $x_{j\ bit\ c} \leftarrow 1 \Leftrightarrow \Sigma_c(...\Sigma_c)^j \in s, c \in \mathbb{Z}|0 \leq c < |\Sigma|,\ j \in \mathbb{Z}|0 \leq j < l$. Note that $\ll$ below represents the bit shift operator.

Algorithm 6: SetAlphaBits(s, l)

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

Algorithm 7, 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. 6 above. For $\Sigma = \{0,1,2, ...,7,8,9\}$, the following condition is true upon completion of SetNumBits(s): $x_{bit\ 3c+j} \leftarrow 1 \Leftrightarrow \Sigma_c(...\Sigma_c)^j \in s, c \in \mathbb{Z}|0 \leq c < |\Sigma|, j \in \mathbb{Z}|0 \leq j < 3$.

Algorithm 7: SetNumBits(s)

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

5.1.2 Filtering FBF Signatures

The different characters between strings s and t can be found by using the exclusive disjunction on their signatures x and y, respectively. Algorithm 8, FindDiffBits(x, y, l), uses a fast bit counting method to count the ones in the exclusive disjunction result of x and y. The loop only executes as many times as there are ones in the string [Wenger 1960]. 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 FindDiffBits(x, y, l) algorithm, described below. The loop will only execute n times for each of the n ones in the integers' exclusive

disjunction's bits, which represents n members in a set, and is denoted as $n = |x \oplus y|$. The algorithm for FindDiffBits(x, y, l) is:

Algorithm 8: FindDiffBits(x, y, l)

Input: x: 32-bit int vector signature for string s
y: 32-bit int vector signature for string t
l: 32-bit integer length of x and y vectors
Output: n: 32-bit integer count of different bits
d: 32-bit unsigned integer
begin
  $i \leftarrow 0$
  $n \leftarrow 0$
  while $i < l$
    $d \leftarrow x_i \oplus y_i$
    while $d > 0$
      $n \leftarrow n + 1$
      $d \leftarrow d \wedge (d - 1)$
    end-while
    $i \leftarrow i + 1$
  end-while
return n
end

The length variable l is set to 1 for numeric signatures as defined in Alg. 7. 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.

5.1.3 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 x, y as: $x = \mathrm{SetNumBits}(s)$ and $y = \mathrm{SetNumBits}(t)$. There

is a relation between *PDL* and FBF signature comparison, $\text{FindDiffBits}(x, y)$, that the set of all string pairs $\langle s, t\rangle \in S \times T$ with a $\text{FindDiffBits}(x, y, l) \leq 2k$, 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 $G_{\leq 2k} = \forall(s, t) \in S \times T$, $\text{FindDiffBits}(x, y, l) \leq 2k$, where x is the signature of s and y 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 $\text{FindDiffBits}(x, y, l) = |x \oplus y|$.

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 $|x \oplus y| = 0$ 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, $|x \oplus y| = 0$.

If a single edit operation is a delete, the worst case is $|x \oplus y| = 1$ 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**, $|x \oplus y| = 1$.

If the single edit is an insertion, the worst case is $|x \oplus y| = 1$ 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**, $|x \oplus y| = 1$.

If the single edit is a substitution, the worst case is $|x \oplus y| = 2$ 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 $|x \oplus y| = 2$.

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 examples above show examples and proof for k = 1. By inductive reasoning, the worst case for a *PDL* of k is $|x \oplus y| = 2k$ if all k edits are substitutions. We claim that

$\forall h(s,t,k) \in H_{\leq k}, \exists g(x,y,l) \in G_{\leq 2k}$

and

$\text{PDL}(s,t,k) = TRUE \implies \text{FindDiffBits}(x,y,l) \leq 2k.$

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. $\text{FindDiffBits}(x,y,l)$ uses signatures x and y for strings s and t, respectively, to decide which pairs to perform the computationally more expensive $PDL(s,t,k)$ using the aforementioned threshold relationship. To process numeric strings, create FBF signature arrays *X*, *Y* as: $\forall s \in S, \exists x \in X, x_i = \text{SetNumBits}(s_i)$ and $\forall t \in T, \exists y \in Y, y_j = \text{SetNumBits}(t_j)$ and follow the process in $\text{MatchStrings}(S,T,X,Y,k,l)$:

Algorithm 9: $\text{MatchStrings}(S,T,X,Y,k,l)$

Input: S, T: Strings lists to be matched
X, Y: 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$(x_i, y_j, l) \leq 2k$
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 $G_{>2k} = \forall(s,t) \in S \times T, |x \oplus y| > 2k$, then $H_{\leq k} \cap G_{>2k} = \emptyset$, $\forall h(s,t,k) \in H_{\leq k}, \nexists g(x,y,l) \in G_{>2k}$ and FindDiffBits$(x,y,l) > 2k \Longrightarrow PDL(s,t,k) \neq TRUE$. 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. 25 below shows a typical relation between the four sets: $G_{\leq 2k}$, $G_{>2k}$, $H_{\leq k}$ and $H_{>k} = \forall(s,t) \in S \times T, PDL(s,t,k) \neq TRUE$. 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 FindDiffBits$(x,y,l) \leq 2k$ but $PDL(s,t,k) \neq TRUE$. It may also be the case that $G_{\leq 2k} \cap H_{>k} = \emptyset$ if FindDiffBits(x,y,l) and $PDL(s,t,k)$ are in perfect agreement. 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. This proof is similar for alphabetic characters and addresses.

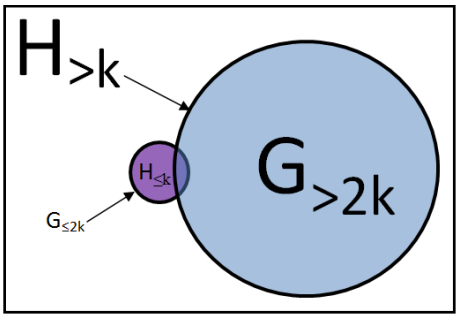


Figure 25: Venn diagram for FPDL relationship

5.1.4 Experiments for FBF

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 [U.S. Census 1990], 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]. 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 [SSA]. 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 [Gravano and Ipeirtis 2001] 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.

5.1.5 Results for FBF

The experimental results show significant performance gains using FBF. The results for SSN are shown below in Table 9 with the edit distance threshold $k = 1$ and the Jaro/Wink threshold set to 0.8 (0.75 for FN). Exactly 5000 of the 25,000,000 pairs are known to match. The *DL* algorithm is used as the baseline and ground truth 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 9, Gen, the $\text{SetNumBits}(s)$ function processes 10,000 SSNs in 0.6 milliseconds 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 24,876,682 unnecessary pair-wise comparisons before processing the strings with *DL* or *PDL*. This is a filter efficiency of more than 99%. *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 9: 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 10 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 10: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

Figure 26 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 given the search space, 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.

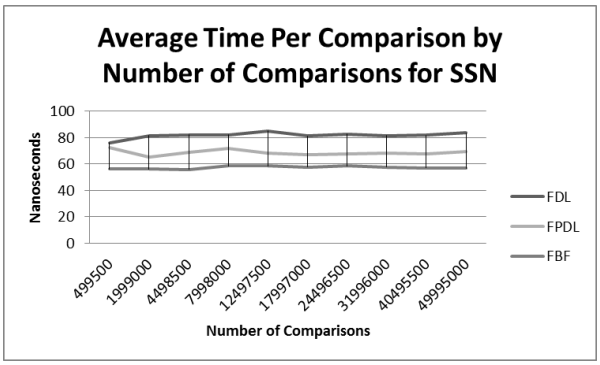


Figure 26: Plot of average pairwise comparison by number of comparisons

Table 11 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, which is a linear runtime function. *FPDL* is 27.29 times faster than *DL*.

Table 11: 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. Table 12 below shows the experimental results for a sample of 5,000 of 547,771 Philadelphia street addresses. Addresses are usually the longest strings in demographic data and typically have the largest alphabet. The experiment contained 2 files, each containing 5,000 strings. The first file contained clean street addresses and the second contained the same street addresses with a single edit error randomly injected in each address. All of the strings in the error file were matched with all of the strings in the clean file and the

accuracy and performance were recorded. The rows are the methods and the columns are type 1 and type 2 errors (false positives and false negatives, respectively), time and computational speedup over the baseline *DL* method. It should be noted that the ground truth for these experiments are the *DL* results. We refer to an optimization as *k*-safe if it produces the same exact string matching results as *DL*. The experimental data was aligned by index, meaning that the error string index 1 was clean string index 1 with a single edit error injected. If a method matched string pairs between sets whose index matched, it was recorded as a true positive match. Any other match was recorded as a false positive. All of the methods with 120 type 1 errors actually have no errors because the results exactly match *DL*. The *FDL* method achieved a 78.18 times speedup. The result for the *FPDL* method achieved a 79.6 times speedup and was more than 3 times faster than Hamming distance, which has an $O(n)$ complexity (where n is the length of the shortest string). These results clearly show the cumulative results of combining *k*-safe optimizations and that the combination does not harm results.

Table 12: 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 13 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 *DL*s $O(mn)$ complexity. Table 13 shows the performance speedups from the smallest average length, FN on the left, to the longest, Ad on the right.

Table 13: 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 14 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 14: 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 15. 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 15: 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 16 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 16: 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

Figure 27 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.

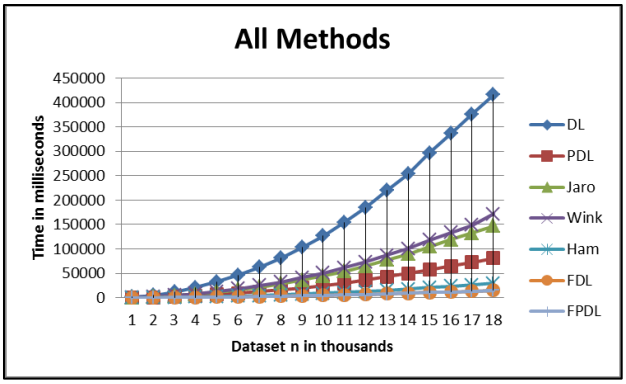


Figure 27: Runtime curves for all methods

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 17 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 17: 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 18 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 18: 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

Figure 28 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 Figure 28, are the middle two curves.

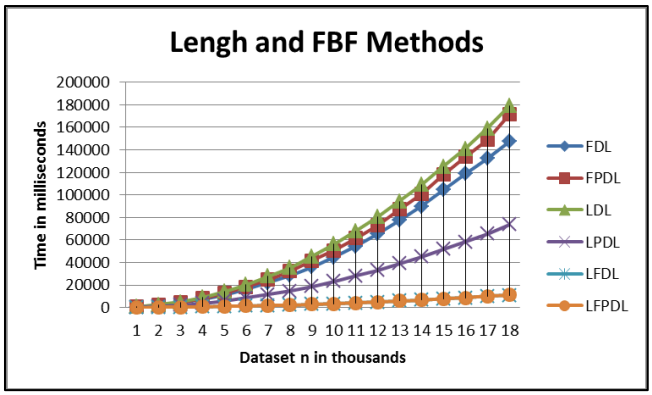


Figure 28: Runtime curves for length filter and FBF methods for last name

The results of polyfit on the runtime curves from Figure 28 are shown in Table 19. Notice that the coefficient *a* for *LFPDL*, 3.41E-05, is about 27% smaller than the *a* coefficient for *FPDL* from the results in Table 17.

Table 19: 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 20 shows the last name results for *DL* and *FPDL* from Table 11 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 25,000,000 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 20 or any of the other experiments that use these filtering methods have any type 2 errors.

Table 20: 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 $\text{FindDiffBits}(m, n, l)$ function for every possible string pair. The FBF row in Table 11 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 21 shows the counts for string lengths for the Census last name data used in our experiments.

Table 21: 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 22 shows the experimental results for comparing 5,000 randomly selected Philadelphia street addresses, one clean dataset and one with single edit errors injected, for *DL* (baseline), standard Damerau-Levenshtein edit-distance, and *FPDL*, Fast Filtered Prefix Pruned Damerau-Levenshtein edit-distance, and results including the addition of the length filter. The combined method increased the speedup of *FPDL* from 79.6 times faster than DL to 130.83 times faster shown in *LFPDL*, Length and Fast Filtered Prefix Pruned Damerau-Levenshtein edit-distance. The filter efficiency describes how well a filter removes string pairs that do not match before applying *DL*. The filter efficiency for the length filter was 61.5% and the combination of length and FBF was 99.987%. Addresses are the longest strings, have the largest alphabet and are the most computationally complex to compare under *DL*.

Table 22: Accuracy and performance results for Ad (Phila. Street Addresses) string experiment using length filter, FBF and PDL

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

5.1.6 Conclusion for FBF

As the number and size of databases grow and research and business problems require merging data without a reliable unique identifier, identity resolution has become a commonly required function for health and human, government services and business 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 the compared 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 up to 25 characters or bytes and a phone number is 10 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. Adding the length filter (LFPDL) to FPDL, further increases performance without loss of accuracy. 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 across heterogeneous databases.

Our methodology substantially differs from SSJoin [Chaudhuri, Ganti and Kaushik 2006] 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 very quickly.

5.2 Second Optimization Technique: Hybrid Bitwise Signature Hashing with Bitwise Neighborhood Generation

Our second method uses a new signature for alphabetic, numeric and alphanumeric that only requires one 32-bit integer for each type. We also developed a signature hashing method that substantially increases performance without harming accuracy.

5.2.1 New Signatures for FBF

It is evident that smaller FBF signatures will process faster and require less memory to store. The original numeric FBF signature only contains one unsigned integer counting up to three occurrences of each digit. The original alphabetic FBF signature contains two and the alphanumeric contains three. As of this report, all three are only contained within one unsigned integer. Other improvements include using the unused bits to record more

information about the strings that they represent. The old numeric signature had two unused bits shown as dashes in Figure 29 below.

```
--999888 77766655 54443332 22111000
00000001 00000011 10000000 11011011
```

Figure 29: Old FBF signature for numeric strings

The new numeric signature uses these bits to record digits four times (%) and digits have occurred five times (#), which still ensures the relation between k edits and a $2k$ bit signature difference is maintained. Figure 30 below shows the new FBF signature for the phone number "215-555-2345".

```
#%999888 77766655 54443332 22111000
11000000 00000011 10010010 11001000
```

Figure 30: New FBF signature for numeric strings

The new algorithm $\text{SetNumBits2}(s)$ is shown below in Algorithm 10. There are two additional tests added to the if-else chain to record the occurrences of 4 or 5 matching digits in the string s.

Algorithm 10: $\text{SetNumBits2}(s)$

Input: s: string of characters
Output: x: 32-bit unsigned integer
$\Sigma = \{0,1,2,3,4,5,6,7,8,9\}$
j: $|\Sigma|$ integer vector of zeros
c: integer to set signature bit
x: integer to contain signature
begin
  for each $s_i \in s$
    if $s_i \in \Sigma$
      $c = c|\Sigma_c = s_i$
      if $j_c = 4$
        $x_{31} = x_{31} \vee (1 \ll 31)$
      else if $j_c = 3$
        $x_{30} = x_{30} \vee (1 \ll 30)$
      else if $j_c = 2$
        $x = x \vee \big(4 \ll (3 * c)\big)$
      else-if $j_c = 1$
        $x = x \vee (2 \ll (3 * c))$
      else-if $j_c = 0$

```
            x = x ∨ (1 ≪ (3 ∗ c))
        end-if
        j_c += 1
    end-if
  end-for
  return x
end
```

The old signature for alphabetic strings used two unsigned integers to count up to two occurrences of each character but left six bits per unsigned integer unused. The new signature uses one unsigned integer, counts the first occurrence of a character in the first 26 bits and uses the six remaining bits to count characters that occur multiple times in the string. Figure 31 below shows that the last six bits (bits to the left) records that up to three characters occurred twice (2222), up to two characters occurred three times (33). The figure below shows the signature for "MISSISSIPPI". This string only has four unique characters but the signature contains nine 1's for a string containing eleven characters. This enables the signature to record some of the characteristics of string length for longer strings with multiples of matching characters.

```
332222ZY XWVUTSRQ PONMLKJI HGFEDCBA
11011100 00000100 10010001 00000000
```

Figure 31: New FBF signature for alphabetic string "MISSISSIPPI"

The new algorithm SetAlphaBits2(s) is shown below in Algorithm 11. There are two additional tests added to the if-else chain to record the occurrences of 4 or 5 matching digits in the string s. The C implementation of this algorithm and the others discussed use registers and bitwise operators to achieve high performance. The algorithm shows that the code is calculated bit-by-bit. The ∧ symbol represents the logical AND, the ∨ symbol represents the bitwise OR and the ≪ symbol is the bit-shift to the next higher order bit in the integer.

Algorithm 11: SetAlphaBits2(s)

Input: s: string of characters
Output: x: vector of 32-bit unsigned integers
$\Sigma = \{A, B, C, \ldots, X, Y, Z\}$
j: $|\Sigma|$ integer vector of zeros to count first occurrence
c: integer to set signature bit
c_2: integer set to zero to count second occurrence
c_3: integer set to zero to count third occurrence
x: integer to contain signature

```
begin
   x = 0
   for each s_i ∈ s
      if s_i ∈ Σ
         c = c|Σ_c = s_i
         if j_c = 0
            x = x ∨ (1 ≪ c)
         else if j_c = 1 ∧ c_2 < 3
            x = x ∨ (1 ≪ (c_2 + 26))
            c_2 += 1
         else if j_c = 2 ∧ c_3 < 2
            x = x ∨ (1 ≪ (c_3 + 29))
            c_3 += 1
         else if j_c = 3
            x = x ∨ (1 ≪ 31)
         end-if
      j_c += 1
      end-if
   end-for
   return x
end
```

There is also a signature designed specifically for Philadelphia street addresses that only requires one unsigned integer for all $|\Sigma| = 36$ characters of $\Sigma = \{0,1,2,3,4,5,6,7,8,9, A, B, C, \ldots, X, Y, Z\}$. We performed a character frequency analysis on all 547,771 taxable street addresses within the city of Philadelphia, shown in Figure 32. The histogram for the count of each address that contains one or more of each alphabet character in Σ. Notice that nearly 89% of all addresses contain at least one 'S' or 'T'. Also note that the characters 'J', 'Q', 'X', and 'Z' are very infrequent in the addresses.

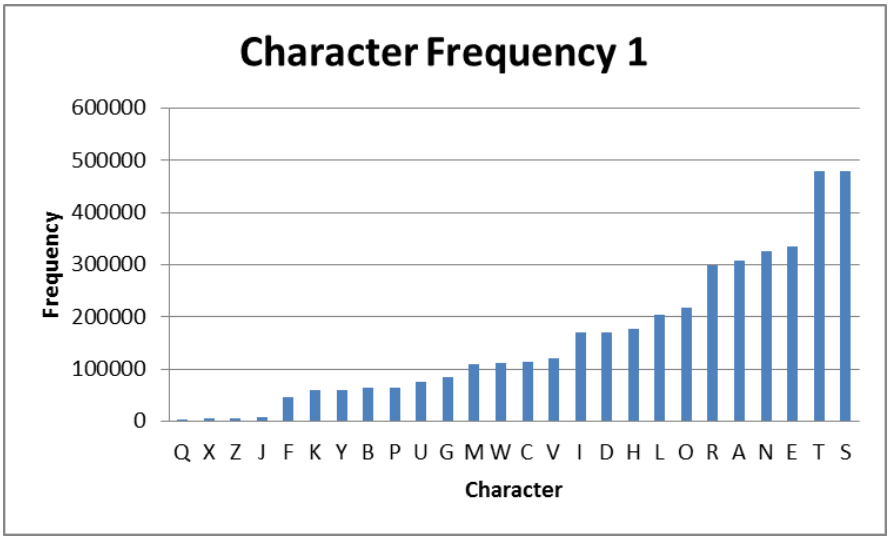


Figure 32: Histogram of character occurrence in Philadelphia addresses

The new signature has the 'S' or 'T' removed since they are not a very good measure of variance because the majority of addresses that contain them. The 'J' and 'Q' were combined into one bit as well as the 'X', and 'Z' characters because they are very infrequent. All ten digits were retained. The figure below shows the address signature for "1801 N BROAD ST".

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

Figure 33: New FBF Address signature

Algorithm 12 shows how to implement the $SetAddrBits(s)$ function to generate a signature for strings that may only contain alphanumeric characters for street addresses.

Algorithm 12: SetAddrBits(s)

Input: s: string of characters
Output: x: 32-bit unsigned integer
$\Sigma = \{A, B, C, D, E, F, G, H, I, JQ, K, L, M, N, O, P, R, U, V, W, XZ, Y, 0,1,2,3,4,5,6,7,8,9\}$
(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 ∈ s
    if s_i ∈ Σ
      c = c|Σ_c = s_i
      x = x ∨ (1 ≪ c)
    end-if
  end-for
  return x
end
```

5.2.2 Bitwise Signature Hashing with Bitwise Neighborhood Generation

One of the problems with Reduced Alphabet Neighborhood Generation is that the codes generated are variable length making it hard to use the code as the index for a hash table bucket. The code for "JAMES" is "00001" and the code for "TAM" is "001". These names are somewhat different and should not map to the same bucket with an integer value of 1. Signature Hashing has the ability to better differentiate disparate strings due to a larger alphabet than Reduced Alphabet Neighborhood Generation and it also has the benefit of using a numeric signature value that directly accesses the hash buckets. In the Signature Hashing scheme discussed below, the string "JAMES" maps to the binary string "0001001010101", which is the decimal 597. The pointer to "JAMES" and all other strings with the same hash code will be in this bucket. We suggest that it is possible to combine both of these methods using bitwise operators to make Signature Hashing faster and Reduced Alphabet Neighborhood Generation more robust. Consider that the 26 character alphabet is condensed to 13 characters as described for signature hashing. Signature Hashing would have to compare all the numeric values in all 2^{13} buckets with the pattern string p to find candidates that differ by $2k$ bits (see Proof of Correctness section for FBF), which are used as candidates for an edit-distance of k or less edits. Instead of comparing all 2^{13} bucket indices, a function can be used to calculate a neighborhood of buckets containing candidates more quickly than comparing all buckets

and the buckets can be inspected quickly by only processing buckets that contain references to strings $s \in S$ (i.e. nonempty buckets). The same logic proves that the FBF is lossless with respect to false negatives also proves that this method is lossless because the hash codes are constructed from disjunctions of bits in signatures, which expands the hash candidate set to a dilation of the signature candidates. Any string signature that will pass the signature filter is guaranteed to be in the hash candidate set. Table 23 below shows the hash buckets local to "JAMES" from the U.S. Census list of first names for 5,163 males and females [U.S. Census 1990].

Table 23: Sample Hash Buckets for 13 Bits

Bucket	Count	Length	Contents																	
585	15	50	HANS	HASSAN	MATHA	NATASHA	NATHAN	SAMANTHA	SAMATHA	SANG	SHAN	SHANA	SHANNA	SHANNAN	SHANTA	TAMATHA	THANH			
586	0	0																		
587	2	50	NATACHA	SHANDA																
588	0	0																		
589	15	50	AGNES	ATHENA	BETHANN	ETHAN	SHANAE	SHANE	SHANTAE	SHANTE	SHEENA	SHENA	SHENNA	TAMESHA	TANESHA	TEGAN	TENESHA			
590	0	0																		
591	1	50	CHANTE																	
592	3	50	MISS	MISTI	TIM															
593	24	50	ANASTASIA	ANISA	ANISSA	ANITA	ANNIS	ANNITA	JANIS	JANITA	JASMIN	MINTA	NITA	SABINA	SANTINA	SIMA	SINA	TAINA	TAMI	TAMMI
				TANIA	TANJA	TATIANA	TIANA	TIANNA	TINA											
594	0	0																		
595	7	50	ANASTACIA	CANDIS	CATINA	DANITA	JACINTA	SANDI	TAMICA											
596	11	50	EMMITT	INES	JENETTE	JENISE	JENNETTE	JESTINE	MISTIE	MITTIE	NETTIE	TENNIE	TIEN							
597	35	50	ANJANETTE	BENITA	BETTINA	FATIMA	JAMES	JANESSA	JANET	JANETT	JANETTA	JANETTE	JANISE	JANNET	JANNETTE	JASMINE	JEANETT	JEANETTA	JEANETTE	JEANNETTA
				JEANNETT	JESENIA	JESSENIA	MAISIE	MATTIE	NENITA	SABINE	SAMMIE	SEBASTIA	STEFANI	STEFANIA	STEFANIE	STEFFANIE	TAMIE	TAMMIE	TIFFANI	TIFFANIE
598	5	50	DENIS	DENISE	DENISSE	DENNIS	DENNISE													
599	4	50	BENEDICT	DENITA	SANDIE	SENAIDA														
600	1	50	SHIN																	
601	15	50	ANISHA	ASHANTI	MAISHA	MISHA	NATASHIA	NATISHA	NISHA	SHAINA	SHANI	SHANITA	SHANTI	TAMISHA	TANISHA	TASHINA	TINISHA			
602	1	50	MITCH																	
603	2	50	CINTHIA	SHANDI																
604	3	50	GENESIS	GINETTE	SIGNE															
605	7	50	BETHANIE	FATIMAH	JANETH	MIESHA	NIESHA	SHENITA	TENISHA											
606	0	0																		
607	2	50	DENISHA	SHANICE																
608	0	0																		
609	1	50	TAMALA																	
610	0	0																		
611	0	0																		
612	2	50	KENT	SELENE																

Notice that bucket 597, containing "JAMES", has 35 strings. If the number of bits for the bucket signatures is increased to 16 bits, using the new alphabetic signature discussed above, the extra 3 bits can be used to record information about recurring characters. The binary signature for "JAMES" becomes "**000**0001001010101". From right to left the extra 3 bits record:

- First occurrence of at least 2 characters that occurred once were found to occur again
- Third occurrence of at least 2 characters that occurred once were found to occur again
- At least 3 characters that occurred twice were found to occur again

These rules ensure that some of the length and frequency of characters are represented in the signatures. Table 24 below shows that the lengths of the strings in the buckets are more similar than in the previous figure.

Table 24: Sample Hash Buckets for 16 Bits

Bucket	Count	Length	Contents							
585	3	50	HANS	SANG	SHAN					
586	0	0								
587	0	0								
588	0	0								
589	6	50	AGNES	ETHAN	SHANE	SHANTE	SHENA	TEGAN		
590	0	0								
591	1	50	CHANTE							
592	1	50	TIM							
593	8	50	JANIS	JASMIN	MINTA	NITA	SIMA	SINA	TAMI	TINA
594	0	0								
595	2	50	CANDIS	SANDI						
596	2	50	INES	TIEN						
597	8	50	BENITA	JAMES	JANET	JANISE	JASMINE	SABINE	STEFANI	TAMIE
598	1	50	DENIS							
599	2	50	DENITA	SANDIE						
600	1	50	SHIN							
601	4	50	MISHA	NISHA	SHANI	SHANTI				
602	1	50	MITCH							
603	1	50	SHANDI							
604	1	50	SIGNE							
605	5	50	JANETH	MIESHA	NIESHA	SHENITA	TENISHA			
606	0	0								
607	2	50	DENISHA	SHANICE						
608	0	0								
609	0	0								
610	0	0								
611	0	0								
612	1	50	KENT							

Notice that bucket 597 contains “JAMES” and seven other names that have the same hash code. The exact match for a search pattern s and a string $t \in T$ is guaranteed to be in the bucket that matches its hash code, where $h(s) = h(t)$. Only eight strings would

have to be compared for an exact match. There are 5,163 unique names in the list. These are names that occurred 100 or more times in the 1990 Census. A binary search could take as much as 12.33 comparisons ($log_2(5{,}163) = 12.33$). Figure 34 below shows the counts of strings in hash buckets by frequency. Notice that 1,387 buckets had only one string and only one bucket had 30. It should also be noted that 63,137 buckets are empty and that 4,354 of the first names are in buckets with 8 or less elements.

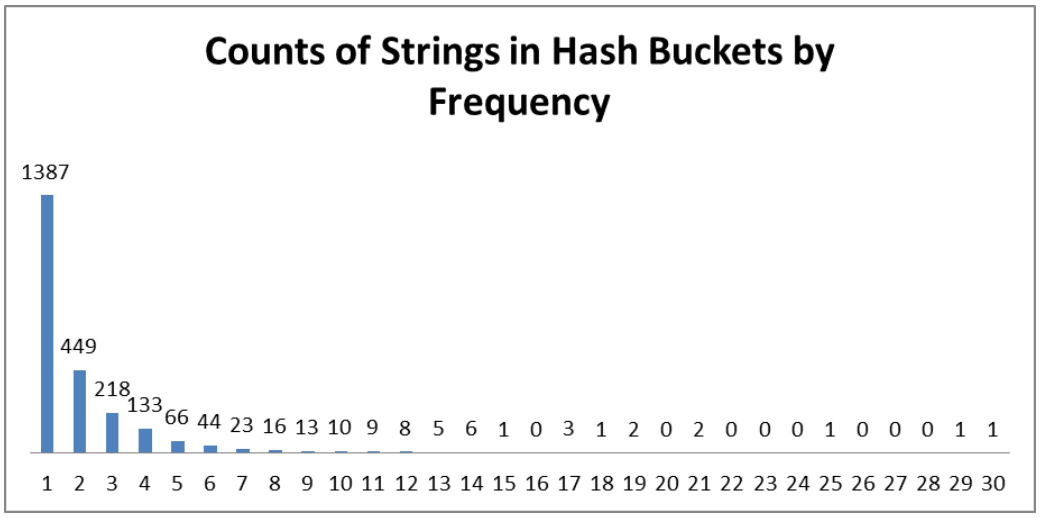


Figure 34: Counts of Strings in Hash Buckets by Frequency

The pointer array of integers for the buckets will use 64K x 4 bytes or 256K of RAM and will be constant as long as 16 bits are used for the hash code. Each of the integer indices in the buckets will use 4 bytes with all buckets using 5,163 x 4 or less than 20K of RAM, which will increase if more strings are added to the hash table. This will double for a 64-bit system, which will have to be used for large RL problems.

Signature Hashing and the concept of Full Neighborhood Generation have been combined with the reduced alphabet from the hashing to perform *Hybrid Bitwise Signature Hashing with Bitwise Neighborhood Generation*. The mapping in Figure 35 below is used to

generate a hash signature for the names “JAMES” (top) and “TAM” (bottom). These names now map to different buckets using this method as shown below.

```
322Y WUSQ OMKI GECA
322Z XVTR PNLJ HFDB
0000 0010 0101 0101

322Y WUSQ OMKI GECA
322Z XVTR PNLJ HFDB
0000 0010 0100 0001
```

Figure 35: Hash codes for JAMES (top) and TAM (bottom)

The neighborhood generation is performed by using bitwise operators. The XOR and bit-shift can be used to quickly compute all buckets that differ by $2k$ bit hash codes as shown in the code below.

```
442 // Generate b (1, 16) 1-bit and b(b-1)/2 2-bit related buckets
443 void get_alpha_hash_codes_2(unsigned int h, unsigned int *codes, unsigned int *n, unsigned int b)
444 {
445     register b1 = 1;     // For bit shift
446     register b2 = 2;     // For bit shift
447     register i = 0;      // Iterator
448     register j = 0;      // Iterator
449     register k = 1;      // Counter
450     // Calculate the size of the list of k-neighborhood
451     *n = 1 + b + b*(b-1)/2;
452     // Copy exact match code
453     codes[0] = h;
454     // Generate one and two bit related buckets
455     for(i=0; i<b; i++)
456     {
457         // Calculate one bit related
458         codes[k++] = h ^ b1;
459         // Calculate two bit related
460         for(j=i+1; j<b; j++)
461         {
462             codes[k++] = h ^ (b1 | b2);
463             b2 <<= 1;
464         }
465         // Shift to next starting bit
466         b1 <<= 1;
467         b2 = b1 << 1;
468     }
469     return;
470 }
```

Figure 36: Code to generate all buckets within a 2 bit difference

The number of buckets in the approximate neighborhood of a pattern’s hash code for a b bit code and k differing bits is:

$$1 + b + \sum_{i=0}^{k-2 \geq 0} \frac{(b-i)(b-i-1)}{2}$$

Equation 25: Number of buckets within 2k bits

Table 25 below shows the $2k$-neighborhood for "JAMES" with $2k = 2$. Notice that "TAM" is $2k$-related to "JAMES", with its hash code differing by 2 bits.

Table 25: 2k = 2 neighborhood buckets for JAMES

The $2k$-neighborhood for "JAMES" with $2k = 2$																		
Bucket	Count	Length	Contents															
8277	30	50	AIMEE	AMIEE	AMMIE	ANNIE	BENJAMIN	BENNIE	FABIAN	FANNIE	JAIMEE	JAIMIE	JAMEE	JAMMIE	JANAE	JANEAN	JANEE	
			JANEEN	JANENE	JANINE	JANNIE	JEANA	JEANE	JEANIE	JEANINE	JEANNA	JEANNE	JEANNIE	JENAE	JENNA	MAMIE	NINFA	
8789	21	50	BETTINA	FATIMA	JANESSA	JANETT	JANETTA	JANETTE	JANNET	JEANETT	JESENIA	JESSENIA	MAISIE	MATTIE	NENITA	SAMMIE	SEBASTIAN	
			STEFANIA	STEFANIE	STEFFANIE	TAMMIE	TIFFANI	TIFFANIE										
117	14	50	ALINE	BLAINE	ELINA	JAMEL	JANEL	KENIA	LAINE	LANIE	LIANE	MAILE	MELANI	MELIA	MELINA	NELIA		
8785	14	50	ANISA	ANISSA	ANITA	ANNIS	ANNITA	JANITA	SABINA	SANTINA	TAINA	TAMMI	TANIA	TANJA	TIANA	TIANNA		
81	12	50	AMI	IAN	IMA	INA	JAMI	JAN	JANI	JINA	MAI	MIA	MINA	NIA				
565	11	50	ALISE	BELKIS	ELIAS	ELISA	FELISA	ISABEL	ISELA	KASIE	KATIE	LEISA	LESIA					
87	10	50	CAMIE	CANDIE	DAINE	DAMIEN	DIANE	JANICE	MACIE	MADIE	MANDIE	NEIDA						
721	10	50	BONITA	JAMISON	JASON	JONAS	ONITA	SIMONA	SONIA	SONJA	TONIA	TONJA						
85	9	50	AMIE	JAIME	JAME	JAMIE	JANE	JANIE	JEAN	JENA	MANIE							
341	8	50	EFRAIN	FERMINA	IRENA	MAIRE	MARIE	MARINE	MARNIE	REINA								
593	8	50	JANIS	JASMIN	MINTA	NITA	SIMA	SINA	TAMI	TINA								
597	8	50	BENITA	***JAMES***	JANET	JANISE	JASMINE	SABINE	STEFANI	TAMIE								
4181	8	50	JAMEY	JANEY	JAYME	JAYMIE	JAYNE	JAZMINE	MAZIE	ZENIA								
9045	8	50	BRITTANIE	EMERITA	ERNESTINA	MARIETTA	MARIETTE	MERISSA	NERISSA	SERAFINA								
629	7	50	ISMAEL	KATELIN	LENITA	MELISA	MELITA	SELINA	TEMIKA									
853	7	50	BERNITA	MARTINE	RENITA	SEBRINA	SERINA	SIRENA	TERINA									
8788	7	50	EMMITT	JENISE	JESTINE	MISTIE	MITTIE	NETTIE	TENNIE									
8821	7	50	MELISSA	MELISSIA	MELLISA	MELLISSA	NATALIE	TAMEIKA	TAMEKIA					**Bucket**	**Count**	**Length**	**Contents**	
581	6	50	META	NETA	SEAN	SENA	STEFAN	TENA						21	1	50	JAE	
589	6	50	AGNES	ETHAN	SHANE	SHANTE	SHENA	TEGAN						517	1	50	ESTA	
789	6	50	ARTIE	BEATRIS	REITA	SERITA	TERISA	TIERA						583	1	50	DANTE	
8773	6	50	ANETTE	ANNETT	ESTEBAN	ESTEFANA	SEEMA	TEENA						592	1	50	TIM	
605	5	50	JANETH	MIESHA	NIESHA	SHENITA	TENISHA							598	1	50	DENIS	
849	5	50	MARIS	MARTI	MARTIN	MIRTA	TRINA							604	1	50	SIGNE	
69	4	50	BEN	EMA	ENA	MAE								625	1	50	NILSA	
213	4	50	EBONI	FIONA	JOANE	JOANIE								628	1	50	MILES	
529	4	50	ISA	TAI	TIA	TISA								631	1	50	MATILDE	
535	4	50	CASIE	JESICA	SADIE	STACIE								709	1	50	AFTON	
541	4	50	FAITH	IESHA	TEISHA	TIESHA								727	1	50	DESPINA	
601	4	50	MISHA	NISHA	SHANI	SHANTI		**Bucket**	**Count**	**Length**	**Contents**			855	1	50	FRANCIS	
637	4	50	KATHLINE	KENISHA	SHEMIKA	SHENIKA		93	2	50	ANGIE	GENIA		861	1	50	MARIBETH	
1617	4	50	AUSTIN	JUNITA	JUSTINA	MAVIS		595	2	50	CANDIS	SANDI		981	1	50	PETRINA	
8725	4	50	BESSIE	BETTIE	JESSIA	JETTA		596	2	50	INES	TIEN		1557	1	50	EVITA	
12885	4	50	TIFFANEY	TIFFANY	YESENIA	YESSENIA		599	2	50	DENITA	SANDIE		1621	1	50	VENITA	
25173	4	50	ANJANETTE	JANNETTE	JEANETTA	JEANNETTA		607	2	50	DENISHA	SHANICE		1623	1	50	VICENTA	
84	3	50	JEN	JENI	MEI			613	2	50	SELMA	TELMA		1749	1	50	FAUSTINO	
577	3	50	SAM	STAN	***TAM***			724	2	50	SIMONE	TONIE		1877	1	50	VERNITA	
661	3	50	JOSEFA	SOFIA	TOBIE			725	2	50	BENITO	JOSEFINA		4677	1	50	STEFANY	
733	3	50	JOSEPHINA	STEPHANI	THOMASINE			1620	2	50	JUSTINE	MITSUE		4689	1	50	YASMIN	
837	3	50	BRENT	TAREN	TRENA			4693	2	50	TIFANY	YASMINE		4949	1	50	BRITNEY	
2133	3	50	MAXIE	MAXINE	XENIA			8791	2	50	BENEDIC	SENAIDA		9813	1	50	FAUSTINA	
8917	3	50	ANTIONE	ANTOINE	STEPANIE			8797	2	50	BETHANI	FATIMAH		41557	1	50	JEANETTE	

The total number of neighboring buckets generated by the code above is 137. The table above shows all buckets that contained pointers to strings in the list of Census first names. Only 78 buckets contained pointers to 346 strings. This is a filter efficiency of

93.3% for the most common male first name in the U.S. The $DL("JAMES", t)$ algorithm would only be applied to these 346 strings.

A similar experiment was run on the U. S. Census last names list [U.S. Census 2000], containing 151,671 last names from the 2000 Census that appeared 100 or more times. A search for "JAMES" found a neighborhood containing 4,424 approximate string match candidates. This is a filter efficiency of 97%. The string "SMITH", which is the most common surname in the U. S., has a neighborhood containing 1,989 and this method has a filter efficiency of 98.7%.

The new method generates the candidate bucket list $C = H_k(u)$ from $u = h(s)$, where s is the search pattern, and performs the following hashed edit distance:

$$\forall C_i \in C, \forall v \in C_i,$$
$$if(DL(s, t_v) \leq k), M \leftarrow (s, t_v)$$

Equation 26: Composition of hashing and DL

where t_v is the string indexed by bucket element v in bucket c, y_v is its FBF signature, and M is a set of matched string pairs. The experimental results will be discussed in the description of the Hybrid model below.

5.2.3 Hybrid Bitwise Signature Hashing with Bitwise Neighborhood Generation, Length Filter, FBF and PDL

Prefix pruning, filters and hashing may be combined to substantially increase edit-distance performance. The key is to use hashing, filtering and pruning methods that are guaranteed not to remove pairs of strings (s, t) where $DL(s, t) \leq k$. We consider that a function is k-safe if this guarantee holds. If all hashing, filtering and pruning optimization methods are k-safe then the composite of these functions will be k-safe

because none of these methods will remove string pairs with a DL edit distance less than or equal to k. Hashing with Bitwise Neighborhood Generation can be used to quickly generate a candidate set C of buckets C_i containing references to strings $t \in T$, where $\forall C_i \in C, \forall t \in C_i$ we have $\sum|C_i| \ll |T|$. Lengths of strings can be used to filter out candidates that have no chance of matching a pattern for k or less edits if the lengths differ by more than k characters. The FBF can be used to make a fast pass over the signatures of string pairs that are passed from hashing and length filtering. The $PDL(s, t, k)$ algorithm is faster than $DL(s, t)$ and produces the same results.

Our solution combines hashing, neighborhood generation, filtering, and pruning to produce substantial performance gains over any single method or sub-combination of methods. When data is processed the FBF is generated from the input string $x = f(s)$ and the hash code $u = h(x)$ is generated from the FBF signature. This creates a direct relationship between the string, the FBF signature and the hash code that can be exploited to substantially increase the filter efficiency for approximate string comparison. Consider the pairwise comparison between the strings "JIM" and "TIM", which have a edit distance of 1. The filter signatures are shown below in Figure 37 (the first is JIM and the second is TIM). A bit is checked or flipped to 1 if a character is present in a string. The real method uses the six extra bits to record recurring characters. For $x = f(JIM)$ or $x = 00000000\ 00000000\ 00010011\ 00000000 = 4{,}864$ and $y = f(TIM)$ or $y = 00000000\ 00001000\ 00010001\ 00000000 = 528{,}640$.

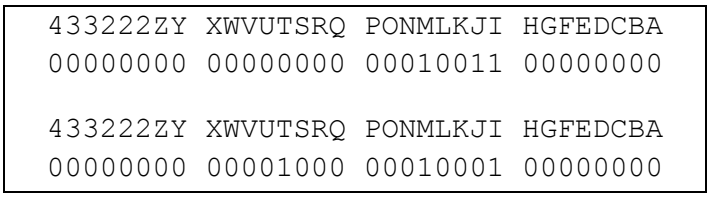


Figure 37: FBF signatures for JIM and TIM

The FBF signatures are compressed into 16 bit hash codes reducing the 2^{32} 32-bit possible strings into 2^{16} 16-bit hash buckets as shown below. For $u = h(4{,}864)$ or $u = 0000\ 0000\ 0101\ 0000 = 80$ and $v = h(528{,}640)$ or $v = 0000\ 0010\ 0101\ 0000 = 592$.

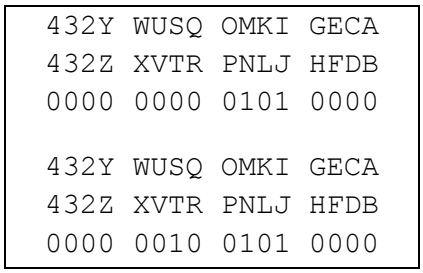


Figure 38: Hash codes for JIM and TIM

The bit string below in Figure 39 is the result of using the XOR operation on the first two signatures. Note that the 'J' and 'T' bits are 1. This is an example of a substitution difference and its filter difference is less than or equal to $2k$ for an edit distance k and will be passed to a more intensive comparator.

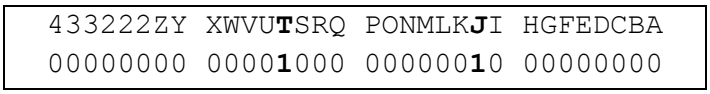


Figure 39: XOR result for JIM and TIM signatures

The bit string below in Figure 40 is the result of using the XOR operator on the hash codes. Notice that the difference between the hash code buckets for "JIM" and "TIM" now only differ by 1 bit due to the compression of characters into hash codes. The combination of 'I' and 'J' in the hash function reduces the differing bits in the hash

codes. This is the aforementioned dilation effect that ensures that the signature hash is *k*-safe.

```
 432Y WUSQ OMKI GECA
 432Z XVTR PNLJ HFDB
 0000 0010 0000 0000
```

Figure 40: XOR results for JIM and TIM hash codes

The example FBF signature and hash codes demonstrate the close relationship between the two optimizations and that they are also *k*-related. We use neighborhood generation to create a list of hash codes for candidate buckets. We do not generate alphabetic strings and code them. We use the hash code for the pattern string s to generate all 1 and 2-bit different hash bucket codes for u thereby skipping the character based generation described above and the subsequent signature and hash coding. The new method generates the candidate bucket list $C = H_k(u)$ from $u = h(x)$ and $x = f(s)$, where s is the search pattern, and performs the following filtering and pruning:

$$\forall C_i \in C, \forall v \in C_i,$$
$$if(L(s, t_v) \leq k \; AND \; F(x, y_v) \leq 2k \; AND \; PDL(s, t_v) \leq k), M \leftarrow (s, t_v)$$

Equation 27: Composition of hashing, length filter, FBF and PDL

where t_v is the string indexed by bucket element v in bucket C_i, y_v is its FBF signature, $L()$ is the length filter, $F()$ is FBF and M is a set of matched string pairs. The filter and pruning methods are able to process the strings from the buckets very quickly as is shown in the experiments below. These functions are arranged such that the computationally cheapest, the length filter, occurs first, the next cheapest, the FBF, occurs second and the most expensive, *PDL*, occurs last. Hashing and the two filters work together to decrease the size of the search space to a small number of candidates relative to the total search space.

5.2.4 Experiments for Hybrid Hash Method

The experiments are similar to those for the FBF using some of the comparators and the first name, last name and address datasets. The string comparators used in the experiments include:

1. Damerau-Levenshtein (DL)
2. Prefix Pruned Damerau-Levenshtein (PDL)
3. FBF Filtered PDL (FPDL)
4. Length then FBF Filtered PDL (LFPDL)
5. Hashed DL (HDL)
6. Hashed PDL (HPDL)
7. Bitwise Neighborhood Hashed DL (GDL)
8. Bitwise Neighborhood Hashed PDL (GPDL)
9. Hybrid Signature Neighborhood Hashed Length and Fast Filtered PDL (GLFPDL)

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)

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.

5.2.5 Results for Hybrid Hash Method

Table 26 below shows the experiments for the first name list from the 1990 U.S. Census [U.S. Census 1990], which contains 5,163 male and female first names. The clean data list had single edit errors injected for all experiments to create an error list. These two lists were compared as an $O(mn)$ but because the lists are the same size it is actually $O(n^2)$. There are two types of hashing shown in the results. *HDL* is straight up signature hashing, where the pattern's hash code is compared to all 64K hash codes in the hash table. *GDL* is the aforementioned neighborhood generation hashing method. *HDL* performed poorly in these because the number of buckets was much greater than the number of strings in the list and the strings are relatively short. The last row is the results for the new composite method. The Time ms column is the runtime to process the lists. On a relatively small set of strings, the hybrid method achieved a speedup of more than 300X, shown in the last row. The last 2 columns show the average time for a search and the average time for a pairwise comparison. The new method averaged 20 microseconds to find all approximate matches for a single string and the average pairwise comparison given the size of the search space was 3.9 nanoseconds.

Table 26: Experimental results for 5,163 Census first names

Experiments on 5,163 Census First Names	MatchPairs	Time ms	Time sec	Time hr	Speedup	Avg Search	Avg Comp
Damerau-Levenshtein (DL)	24,481	32870	32.870	0.009	1.000	6.366453612	0.001233092
Prefix Pruned DL (PDL)	24,481	12496	12.496	0.003	2.630	2.420298276	0.000468778
Fast Filtered PDL (FPDL)	24,481	933	0.933	0.000	35.230	0.18070889	3.50008E-05
Length Filtered FPDL (LFPDL)	24,481	819	0.819	0.000	40.134	0.158628704	3.07241E-05
Hashed DL (HDL)	24,481	31192	31.192	0.009	1.054	6.04144877	0.001170143
Hashed PDL (HPDL)	24,481	15823	15.823	0.004	2.077	3.064691071	0.000593587
Bitwise Neighborhood Hashed Hashed DL (GDL)	24,481	939	0.939	0.000	35.005	0.181871005	3.52258E-05
Bitwise Neighborhood Hashed PDL (GPDL)	24,481	457	0.457	0.000	71.926	0.08851443	1.7144E-05
Hybrid Sig/Neigh Hashed Length and Fast Filtered (GLFPDL)	24,481	104	0.104	0.000	***316.058***	***0.020143328***	***3.90148E-06***

Table 27 below shows the experiments for the last name list from the 2000 U.S. Census [U.S. Census 2000], which contains 151,671 last names. The last row is the results for

the new method. On a larger set the results are much better. The experiments show over a 1000X speedup and still produce the same matched pairs as Damerau-Levenshtein. The new method averaged 224 microseconds to find all approximate matches for a single string and the average pairwise comparison given the size of the search space was 1.5 nanoseconds.

Table 27 Experimental results for 151,671 Census last names

Experiments on 151,671 Census Last Names	MatchPairs	Time ms	Time sec	Time hr	Speedup	Avg Search	Avg Comp
Damerau-Levenshtein (DL)	1,254,103	34106563	34106.563	9.474	1.000	224.8720124	0.00148263
Prefix Pruned DL (PDL)	1,254,103	14174836	14174.836	3.937	2.406	93.45778692	0.000616188
Fast Filtered PDL (FPDL)	1,254,103	1030502	1030.502	0.286	33.097	6.794324558	4.47965E-05
Length Filtered FPDL (LFPDL)	1,254,103	721989	721.989	0.201	47.240	4.760231026	3.13852E-05
Hashed DL (HDL)	1,254,103	36398984	36398.984	10.111	0.937	239.9864443	0.001582283
Hashed PDL (HPDL)	1,254,103	16626852	16626.852	4.619	2.051	109.6244635	0.000722778
Bitwise Neighborhood Hashed DL (GDL)	1,254,103	611113	611.113	0.170	*55.811*	4.029201363	2.65654E-05
Bitwise Neighborhood Hashed PDL (GPDL)	1,254,103	274686	274.686	0.076	124.166	1.811064739	1.19407E-05
Hybrid Sig/Neigh Hashed Length and Fast Filtered (GLFPDL)	1,254,103	33987	33.987	0.009	***1003.518***	***0.224083707***	***1.47743E-06***

Table 28 below shows the results for street address experiments with lists containing 10,000 strings each. *HDL* fared much better because the address strings are much longer than names. The new method had a speedup of over 2300 times and the average pairwise comparison was approximately 2.5 nanoseconds.

Table 28: Experimental results for 10,000 Philadelphia street addresses

Experiments on 10,000 Philadelphia Street Addresses	MatchPairs	Time ms	Time sec	Time hr	Speedup	Avg Search	Avg Comp
Damerau-Levenshtein (DL)	10,467	596831	596.831	0.166	1.000	59.683	5.97E-03
Prefix Pruned DL (PDL)	10,467	42924	42.924	0.012	13.904	4.292	4.29E-04
Fast Filtered DL (FDL)	10,467	177596	177.596	0.049	3.361	17.760	1.78E-03
Length Filtered DL (LDL)	10,467	223683	223.683	0.062	2.668	22.368	2.24E-03
Length Filtered FPDL (LFPDL)	10,467	7581	7.581	0.002	78.727	0.758	7.58E-05
Hashed DL (HDL)	10,467	14131	14.131	0.004	42.235	1.413	1.41E-04
Bitwise Neighborhood Hashed Hashed DL (GDL)	10,467	9394	9.394	0.003	63.533	0.939	9.39E-05
Hybrid Sig/Neigh Hashed Length and Fast Filtered (GLFPDL)	10,467	252	0.252	0.000	***2366.499***	***0.025***	***2.52E-06***

The timing model in Figure 41 below shows that the average runtime for 50,000 addresses to all 547,771 street addresses in the City of Philadelphia. The x-axis is the number of addresses in each of two lists and the y-axis is the runtime in milliseconds. The formula of the curve is $y = 2.08E\text{-}06x^2 + 0.107x - 6628.47$. The GLFPDL method

was 3,000 times faster than DL for the experiment with all addresses. Using this method would take approximately 36 minutes to compare 2 lists, each containing 1,000,000 addresses each. The average pairwise comparison given the size of the search space was two nanoseconds.

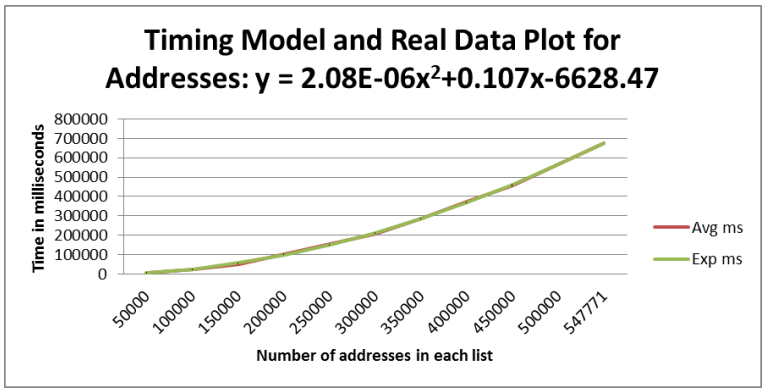


Figure 41: Timing model for 50,000 to all Philadelphia street addresses

5.2.6 Conclusion for Hybrid Hash Method

The combination of the new FBF signature when combined with the hybrid hashing, filtering and pruning distance metric, combined with Bitwise Neighborhood Generation, provides significant performance advantage over the previous LFPDL method, LFPDL achieved a maximum of 130 times speedup and the new method achieved a 3,000 times speedup.

5.3 Third Optimization Technique: Probabilistic Signature Hashing (PSH)

Our latest Damerau-Levenshtein string distance optimization is described below. The new method doubles the performance of GLFPDL by making use of character co-

occurrence probabilities revealed within the data, which is used to generate a more efficient hash function. We have developed a methodology, the Probabilistic Signature Hash (PSH) that combines the capabilities of hashing, filtering and pruning in conjunction with co-occurrence probabilities revealed by the data to substantially increase the speed of approximate string matching in datasets.

5.3.1 Hash Filtering Revisited

After some research, a method referred to as Signature Hashing was found in a methodology comparison report [Boytsov 2011]. This concept, albeit a different methodology, was modified and integrated into our composite method. A signature hashing method was developed based on our pruning and filtering methods. FBF is basically a 32-bit binary checklist of characters that appear in strings. These signatures can be compared very quickly using binary machine instructions. The new signature based hashing reduces the signature to a 16-bit code, which is used as the hash bucket code. The method is deterministic—all exact match strings will be in the same bucket and all strings that match by *k* edits will be in buckets whose codes differ by 2*k* or less bits. Our first attempt to hash signatures is shown in the example below. Figure 42 shows the signature for the address "1801 N BROAD ST" as was described in Signature filtering. Figure 43 shows the reduction of the 32-bit signature to a 16-bit hash code by combining the bits from the signature, where bits '0' and '1' are combined for the 0 bit, bits '2' and '3' for the 1 bit, …, '30' and '31' for the 15 bit. This reduces the upper limit of the search index space from 2^{32} to 2^{16} , which is a reasonable number of buckets for a hash table. This example shows an address string. This methodology was also applied to

numeric and alphabetical strings. The experiments and results will be discussed in subsequent sections.

```
          J           X
ABCDEFGH IQKLMNOP RUVWZY01 23456789
11010000 00000110 10000011 00000010
```

Figure 42: Signature for "1801 N BROAD ST"

```
    I      X
ACEGJKMORVY02468
BDFHQLNPUWZ13579
1100001110010001
```

Figure 43: Signature hash code for "1801 N BROAD ST"

Algorithm 13 shows the calculation of a simple hash code from a string signature. The 'A' and "B' bits or bits 0 and 1 from the signature shown above are the lowest order bits (the signature is shown inverted with the lowest order bit first) and are combined to generate the lowest order bit in the hash code shown above. The algorithm basically combines each two bits, in order into the hash code. The line with $m \wedge 3$ compares the bits under consideration with the binary string "011", which is true if either considered bits from the signature is '1'.

Algorithm 13: GetHashCode(m)

Input: m: 32-bit vector signature for string s
Output: h: 16-bit integer hash code
begin
  $i = 0$: 32-bit integer iterator
  $b = 0$32-bit integer for bit shift
  $h = 0$
  while $i < 16$
    if $m \wedge 3$
      $h = h \vee b$
    end-if
    $b = b \ll 1$
    $m = m \gg 2$
    $i = i + 1$
  end-while
  return h

end

We have also found that scanning all bucket codes for candidates is not necessary. A function can be used to more efficiently generate all approximate match buckets that contain all strings guaranteed to match using FBF. This is because the hash coding is based on a disjunction of bits in the FBF signature. Only 137 buckets need be compared for an edit-distance $k = 1$, which, like signatures, relates to a $2k$ difference in hash codes for approximate matching. This is much faster than scanning all 65,536 hash codes. It should be noted that increasing k will decrease performance. For $k = 2$, there will be 2,517 buckets. The number of buckets searched grows by $\sum_{r=0}^{2k}\binom{16}{r}$. The 137 buckets for $k = 1$ are:

- The exact match bucket code $\binom{16}{0}$
- 16 1-bit differing codes $\binom{16}{1}$
- 120 2-bit differing codes $\binom{16}{2}$

Algorithm 14 below shows the calculation of hash candidates.

Algorithm 14: GetHashCandidates(h)

Input: h: 16-bit hash code
Output: C_*: Vector of 137 candidate hash codes
begin
  $i = 0$: 32-bit integer iterator
  $j = 0$: 32-bit integer iterator
  $k = 1$: 32-bit integer counter
  $b_1 = 1$: 32-bit integer for bit shift
  $b_2 = 2$: 32-bit integer for bit shift
  $C_0 = h$
  while $i < 16$
    $C_k = h \oplus b_1$
    $k = k + 1$
    $j = i + 1$
    while $j < 16$

$C_k = h \oplus (b_1 \vee b_2)$
$k = k + 1$
$b_2 = b_2 \ll 1$
$j = j + 1$
end-while
$b_1 = b_1 \ll 1$
$b_2 = b_1 \ll 1$
$i = i + 1$
end-while
return C
end

To make the hash function algorithms more readable, we define a few data structures. We define a DATA structure for a datum d as shown in Data Structure 1 below. The structure contains the string value, 32-bit signature, hash code and the length of the string value.

Data Structure 1: DATA

```
*val;   // String value pointer
sig;    // String signature
bkt;    // Hash bucket code
len;    // Length of string value
```

We define the DATA_ARRAY structure as shown in Data Structure 2 below. The structure contains an array of DATA elements, the count of DATA elements inserted, the pre-allocated length of the array and the type of data contained in the array.

Data Structure 2: DATA_ARRAY

```
DATA *array; // Array of DATA
count;       // Count of entries in array
length;      // Pre-allocated length of array
type;        // Type of DATA (alphabetic, numeric or address)
```

We define the HASH_ARRAY structure as shown in Data Structure 3 below. It contains an array of pointers to hash bucket arrays, an array recording the count of insertions into

each bucket, pre-allocated length of each bucket and the configuration of the dynamic hash function.

Data Structure 3: HASH_ARRAY

```
*array[65536];// Array of hash buckets
count[65536]; // Count of entries in each bucket
length[65536];// Allocated length of each bucket
pr[16][2];       // Prob Pair Config (Used to store dynamic hash function)
```

Algorithm 15 shows the hash match algorithm.

Algorithm 15: HashMatch(s, st, l, k, H, D)

Input: s: string s
st: String type (alphabetic, numeric or address)
l: 32-bit integer length of s
k: 32-bit integer max edit-distance
D: DATA_ARRAY
H: HASH_ARRAY
Output: A_*:d length vector of matching string indices
d: 32-bit unsigned integer count of matches
Begin
$m = 0$: 32-bit vector signature for string s
$h = 0$: 16-bit hash code for signature m
C_*: Hash bucket candidate array of length 137
$i = 0$: Iterator
$j = 0$: Iterator
$d = 0$: Count of matches
(Set signature and hash code)
if st is alphabetic
$m =$ SetAlphaBits2(s)
else if st is numeric
$m =$ SetNumBits2(s)
else if st is address
m = SetAddrBits(s)
end-if
h = GetHashCode(m)
C = GetHashCandidates(h)
(Process all candidates in C)
$\forall c \in C$
$\forall c_i \in c$
if $abs(l - D[c_i].len) \leq (k + 1)$
if FindDiffBits$(m, D[c_i].bkt) \leq 2$
if $PDL(s, D[c_i].val, 1)$

```
                $A_d = c_i$
                $d = d + 1$
            end-if
        end-if
    end-if
  return A, d
end
```

There was a problem with the organization of the strings into hash buckets using the previously described naïve hashing method. The hash table for Philadelphia addresses had 46,129 of 65,536 empty buckets with a max bucket size of 6,513 strings (see Table 30 below in Experiments). The Probabilistic Pair Hash technique was developed to smooth the contents of the buckets without violating the *k*-safe rule. Smoother hash tables with less empty buckets and a lower number of strings in non-empty buckets may increase performance. The new method performs a frequency analysis of co-occurring bits that are set to 1 in the signatures. For addresses the following characters are used for signature generation (note that Q is JQ and Z is XZ):

```
ABCDEFGHIQKLMNOPRUVWZY0123456789
```

Figure 44: Characters for address signature

The comparison makes a single pass over all of the addresses in the data set and counts the frequency for all possible enumerated pairs of signature bit indices (the example has characters instead of bits to make it more readable):

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

Figure 45: Frequencies of co-occurring pairs

Sort the enumerated pairs by frequency:

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

Figure 46: Frequency sorted co-occurring pairs

Start at the frequency closest to the mean and select next best pairs where neither character has been picked in a previous pair. The characters below were selected from the signatures for the street address data. Notice that the set frequencies are relatively close.

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 47: Pairs and frequencies selected for addresses

Algorithm 16 shows the creation of the probabilistic hash function. Our new hashing method generates a dynamic hash function based on the statistical properties of the data signatures. This is accomplished by recording the counts of the co-occurrence of bits for all string signatures in the dataset and selecting combinations that will reduce the maximum bucket size in the hash table. Less entries in a hash bucket locally reduces the number of string comparisons. Our method uses chaining to handle collisions. So many buckets may have many entries. The method counts the co-occurrence of character pairs in a string by examining the '1's in the binary signatures.

The signatures for "SMITH" and "ABLE" are shown below. For example, bit 0 for an alphabetic signature records that at least one 'A' is in a string and bit 1 records a 'B'. The count for of "AB" in "SMITH" is 0 but the count in "ABLE" is 1 because 'A' and/or 'B' is contained within the string. It should be noted that all pairs that contain 'A' $\{AB, AC, AD, \dots, A3_0, A3_1, A4\}$ or 'B' $\{BC, BD, BE, \dots, B3_0, B3_1, B4\}$ will record the

presence of either character. All counts of co-occurring characters in $\{AB, AC, AD, \ldots, 3_0 3_1, 3_0 4, 3_1 4\}$ are counted from the dataset. The character representation of signature elements is to explain the method using an alphabetic string as an example. The system actually uses the indices of the bit positions $\{00\ 01, 00\ 02, 00\ 03, \ldots, 29\ 30, 29\ 31, 30\ 31\}$, which enables the same methods to be used on all data types that use the aforementioned 32-bit signatures.

```
ABCDEFGH IJKLMNOP QRSTUVWX YZ222334
00000001 10001000 00110000 00000000
11001000 00010000 00000000 00000000
```

Figure 48: Signatures for "SMITH" and "ABLE"

The mean of the counts vector is calculated and the counts are sorted, ascending. The count closest to the mean is selected as the first best choice and recorded as the first bit in the dynamic hash function. The next best count closest to the mean, where neither bit position has already been selected, is selected until all bit positions are assigned in the hash function.

Algorithm 16: GetHashFunction(D)

Input: D: DATA_ARRAY
Output: F_{**}: probabilistic hash function 16 by 2 array of integers
begin
$n_* = 0$: 32 length array of integers
$idx_{**} = 0$: 496 by 2 array of integers
$cnt_* = 0$: 496 length array of integers
$sum = 0$: Integer accumulator
$avg = 0$: Average
$bestIdx = 0$: Record best idx so far
$bestVal = 0$: Record best value so far
$i = 0$: Iterator
$j = 0$: Iterator
$k = 0$: Iterator
$m = 0$: Signature
(Process all data string values in D)
$\forall d \in D$
$m = d.sig$

while $i < 32$
if $m \wedge 1$
$n_i = n_i + 1$
end-if
$m = m \ll 1$
$i = i + 1$
end-while
$i = 0$
while $i < 31$
$j = i + 1$
while $j < 32$
if $(n_i > 0) \vee (n_j > 0)$
$cnt_k = cnt_k + 1$
$sum = sum + 1$
end-if
$idx_{k,0} = i$
$idx_{k,1} = j$
$k = k + 1$
end-while
end-while
(Calculate the hash function F)
$avg = sum/496$
$sort(cnt, idx)$
$n_* = 1$
$bestIdx = 0$
$bestVal = abs(avg - cnt_0)$
$i = 1$
while $i < 496$
if $abs(avg - cnt_i) < bestVal$
$bestIdx = i$
$bestVal = abs(avg - cnt_i)$
end-if
end-while
$F_{0,0} = idx_{bestIdx,0}$
$F_{0,1} = idx_{bestIdx,1}$
$cnt_{idx_{bestIdx,0}} = 0$
$cnt_{idx_{bestIdx,1}} = 0$
$j = bestIdx - 1$
$k = bestIdx + 1$
$i = 1$
while $i < 16$
while $j \geq 0$
if $\left(cnt_{idx_{j,0}} > 0\right) < \left(cnt_{idx_{j,1}} > 0\right)$
$break$
else
$j = j - 1$
end-if

```
    end-while
    while k < 496
      if (cnt_{idx_{k,0}} > 0) < (cnt_{idx_{k,1}} > 0)
        break
      else
        k = k + 1
      end-if
    end-while
    if j ≥ 0 ∧ k ≤ 496
      if abs(avg − cnt_k) ≥ abs(avg − cnt_j)
        F_{i,0} = idx_{j,0}
        F_{i,1} = idx_{j,1}
        cnt_{idx_{j,0}} = 0
        cnt_{idx_{j,1}} = 0
        j = j − 1
      else
        F_{i,0} = idx_{k,0}
        F_{i,1} = idx_{k,1}
        cnt_{idx_{k,0}} = 0
        cnt_{idx_{k,1}} = 0
        k = k + 1
      end-if
    else-if j ≥ 0
        F_{i,0} = idx_{j,0}
        F_{i,1} = idx_{j,1}
        cnt_{idx_{j,0}} = 0
        cnt_{idx_{j,1}} = 0
        j = j − 1
    else-if k ≤ 496
        F_{i,0} = idx_{k,0}
        F_{i,1} = idx_{k,1}
        cnt_{idx_{k,0}} = 0
        cnt_{idx_{k,1}} = 0
        k = k + 1
    end-if
  end-while
  return F
end
```

Algorithm 17 shows the calculation of the probabilistic hash bucket code from the hash function $H.pr$. Note that the reference to $H.pr$ is the array F returned from Algorithm 16. A pass over the data converts the signatures into their respective dynamic hash

codes, which are stored in the DATA elements. We call this a dynamic hash code because it can be updated to best represent the current statistical properties of the data, ensuring near optimal performance as is shown in our experiments.

Algorithm 17: GetHashCodeF(m, H)

Input: m: 32-bit vector signature for string s
H: HASH_ARRAY
Output: h: 16-bit integer hash code
begin
$i = 0$: 32-bit integer iterator
$b = 0$: 32-bit integer for bit shift
$h = 0$
while $i < 16$
if $\left((m \wedge 2^{H.pr_{i,0}}) > 0\right) \vee \left((m \wedge 2^{H.pr_{i,1}}) > 0\right)$
$h = h \vee 2^i$
$i = i + 1$
end-if
end-while
return h
end

The improved hash matching function is shown below in Algorithm 18. The subtle difference is the call to the GetHashCodeF(m, H) function to get the probabilistic hash code based on the hash function stored in the HASH_ARRAY.

Algorithm 18: HashMatchF(s, st, l, k, H, D)

Input: s: string s
st: String type (alphabetic, numeric or address)
l: 32-bit integer length of s
k: 32-bit integer max edit-distance
D: DATA_ARRAY
H: HASH_ARRAY
Output: A_*:d length vector of matching string indices
d: 32-bit unsigned integer count of matches
Begin
$m = 0$: 32-bit vector signature for string s
$h = 0$: 16-bit hash code for signature m
C_*: Hash bucket candidate array of length 137
$i = 0$: Iterator
$j = 0$: Iterator

$d = 0$: Count of matches
(Set signature and hash code)
if st is alphabetic
$m =$ SetAlphaBits2(s)
else if st is numeric
$m =$ SetNumBits2(s)
else if st is address
m = SetAddrBits(s)
end-if
h = GetHashCodeF(m, H)
C = GetHashCandidates(h)
(Process all candidates in C)
$\forall c \in C$
$\forall c_i \in c$
if $abs(l - D[c_i].len) \leq (k + 1)$
if FindDiffBits($m, D[c_i].bkt) \leq 2$
if $PDL(s, D[c_i].val, 1)$
$A_d = c_i$
$d = d + 1$
end-if
end-if
end-if
return A, d
end

5.3.2 Experiments for PSH

The data for the experiments was the 547,771 Philadelphia address data mentioned above. The string comparators used in the experiments include:

1. Damerau-Levenshtein (DL)
2. FBF Filtered PDL (FPDL)
3. Length then FBF Filtered PDL (LFPDL)
4. Hybrid Signature Neighborhood Hashed Length and Fast Filtered PDL (GLFPDL)
5. Probabilistic Hybrid Signature Neighborhood Hashed Length and Fast Filtered PDL (PGLFPDL)

The experiments were run 5 times and their average was recorded as the result. 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.

5.3.3 Results for PSH

Non-Probabilistic Hashed Method

The experimental results are shown in Table 29 below. The composite method with hashing is almost 1,800 times faster than the baseline *DL* method. The Pair Time column contains the average comparison time in milliseconds for a single string pair.

Table 29: Experiments on 5,000 street addresses with GLFPDL hashing

Ad	Type1	Type2	Time ms	Speedup	Pair Time
DL	120	0	135,099	1.00	5.40E-03
FPDL	120	0	1,697	79.60	6.79E-05
LFPDL	120	0	1,033	130.82	4.13E-05
GLFPDL	120	0	76	1,777.62	3.04E-06

Probabilistic Hashed Method

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 Probabilistic Pair method in the AdCleP column decreases the empty buckets 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 30: 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

The performance results for 5,000 randomly selected addresses are shown in Table 31 below. The new method is more than 2,800 times faster than the baseline *DL* method and produces the same results.

Table 31: Experiments on 5,000 addresses with PGLFPDL probabilistic hashing

Ad	Type1	Type2	Time ms	Speedup	Pair Time
DL	120	0	135,099	1.00	5.40E-03
FPDL	120	0	1,697	79.60	6.79E-05
LFPDL	120	0	1,033	130.82	4.13E-05
GLFPDL	120	0	76	1,777.62	3.04E-06
PGLFPDL	120	0	48	2,814.56	1.92E-06

The same experiments were performed with bigger data, containing 25,000 address strings in each dataset. The results are below and show that the new method is much more effective on larger datasets at more than 4,500 times faster than *DL*.

Table 32: Experiments on 25,000 addresses with PGLFPDL probabilistic hashing

Ad	Type1	Type2	Time ms	Speedup	Pair Time
DL	3,021	0	4,217,067	1.00	6.75E-03
FPDL	3,021	0	94,307	44.72	1.51E-04
LFPDL	3,021	0	46,985	89.75	7.52E-05
GLFPDL	3,021	0	1,543	2,733.03	2.47E-06
PGLFPDL	3,021	0	920	4,583.77	1.47E-06

The next set of experiments has 125,000 address strings per input file. *Note that the time for *DL* was calculated from the previous experiment because it would take too long to complete. *PGLFPDL* is now over 5,000 times faster than *DL*.

Table 33: Experiments on 125,000 addresses with PGLFPDL probabilistic hashing

Ad	Type1	Type2	Time ms	Speedup	Pair Time
DL*	67,392	0	105,426,675	1.00	6.75E-03
FPDL	67,392	0	2,421,357	43.54	1.55E-04
LFPDL	67,392	0	1,116,077	94.46	7.14E-05
GLFPDL	67,392	0	36,710	2,871.88	2.35E-06
PGLFPDL	67,392	0	20,928	5,037.59	1.34E-06

The next set of experiments has all 547,771 Philadelphia address strings per input file. *Note that the times for *DL*, *FDL* and *LFDL* were calculated from the previous experiment because they would take too long to complete. *PGLFPDL* is now nearly 6,000 times faster than *DL*.

Table 34: Experiments on all 547,771 addresses with PGLFPDL probabilistic hashing

Ad	Type1	Type2	Time ms	Speedup	Pair Time
DL*	955,992	0	2,024,550,229	1.00	6.75E-03
FPDL*	955,992	0	46,498,278	43.54	1.55E-04
LFPDL*	955,992	0	21,432,469	94.46	7.14E-05
GLFPDL	955,992	0	606,182	3,339.84	2.02E-06
PGLFPDL	955,992	0	341,693	5,925.06	1.14E-06

Comparison to Trie-Join

The experiments in Table 35 below compare the new method to a recently published novel string comparison optimization called Trie Join and a variant Bi Trie Join [Wang, Feng and Li 2010]. 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 35: 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

5.3.4 Conclusion for PSH

The composite probabilistic hashing methodology is primarily designed for the Record Linkage (*RL*) problem for very large databases. The new method and Trie Join methods are all thousands of times faster than *DL*. However, due to the size of the data problem and the limited information available for matching demographic data records, we suggest that our new method is more appropriate for comparing record fields that may contain minor spelling errors. The *DL* algorithm has been found to match approximately 80% of data entry errors. Our new method is both more compact and *k*-safe, providing the same match results as *DL* but nearly 6,000 times faster. Modern big databases have the potential for substantial missing and incorrect field entries. We found 40% of Social Security Numbers missing from a real health and social services dataset and cannot be sure how many contain entry errors. Faster edit-distance comparison of database fields is crucial to the record linkage process. Record linkage is necessary to link a client's data

across multiple heterogeneous databases to aid health and social service workers to efficiently and effectively serve their communities.

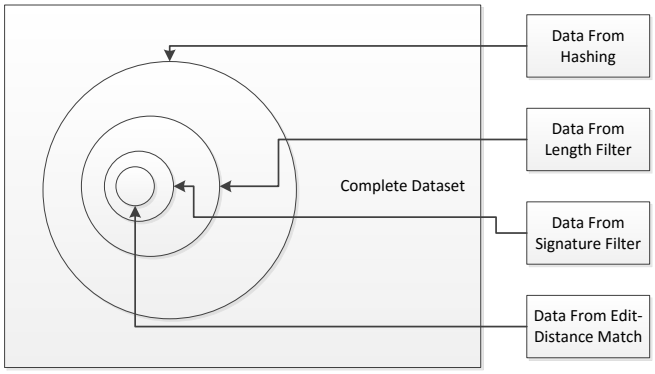


Figure 49: k-Safe Sets

The hashing process is *k*-safe because the hash codes are generated by a disjunction that combines bits from the signature into the hash code, which dilates the signature candidate set and will contain all candidates within 2*k* differing bits. Length filtering is *k*-safe because strings that differ by more than *k* characters cannot differ by *k* or less edits. As previously discussed, the signature filter is *k*-safe. Prefix pruning is *k*-safe because the path in the DL matrix that contains all matches within k edits will all be contained within the 2*k*+1 stripe on the diagonal. Because all of these methods are *k*-safe, meaning that they do not filter out true positive matches, their composite is *k*-safe. These four techniques progressively reduce computing complexity without harming comparison accuracy (Figure 49). The Prefix Pruning produces exactly the same results as DL. The Signature Filter is mathematically proven not to filter out strings that would pass DL for *k* or less edits and represents a dilation of the set of strings that pass DL. The Length Filter is guaranteed to pass all strings that will pass DL. It is also mathematically proven that the results returned from the Bitwise Signature Hash function will contain all strings that will pass the Bitwise Signature Filter because the hash function is based on the

signature and the hash set represents a dilation of the set of strings that will pass the signature-based filter. Since all of these optimizations pass all strings in a dataset for k or less edits for a given search pattern, their composite will also pass all strings in a dataset for k or less edits. We refer to this quality of optimization as being k-safe.

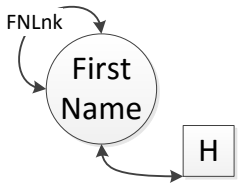
PSH, like most edit-distance optimizations, is most efficient for $k = 1$. Increasing the value of k will substantially increase the time to find approximate matches and the number of approximate matches. The ALIM system uses string relationship graphs that contain edges for exactly how strings are related. The FNLnk edge would record that "JIM" is an alias of "JAMES", "KIM" is one edit different that "JIM", and "JEM" has the same Soundex code as "JIM". The edge would link all alias, approximate ($k = 1$) and phonetic matches for all observed strings. For RL, we do not suggest increasing to $k > 1$, because of the potential for false positives. Consider that "JIM", "JOE" and "JEN" are all 2 edits different from each other but are distinctively different names. PSH was designed for use in ALIM to handle minor spelling errors. Most demographic data used in RL are relatively short strings.

CHAPTER 6

STRING AND RECORD RELATIONSHIP GRAPHS

RL systems need to organize and store data efficiently and effectively to ensure that the system produces correct results in a timely manner. [Ravikumar and Cohen 2004] and [Kardes, Konidena, Agrawal, Huff and Sun 2013] use graphs in the decision models to find records that approximately match. We use graphs in the data model to accommodate changing temporal data. To store data we introduce the concept of a String Relationship Graph. Strings can be related in different ways. "TIM" and "JIM" are spelled approximately alike because they differ by one edit. "SMITH" and "SMYTHE" are phonetic matches because they sound alike. "JAMES" and "JIM" are aliases because they are synonymous terms that may refer to the same person. These relationships are very important for RL systems. Figure 50 on the left below shows a String Relationship Graph for first names. The First Name vertex represents the names and the FNLink edge represents how any two given names in the graph may be related. These are represented as edge and vertex lists in our model. The small square containing the 'H' is a hash table that enables fast hash searches when inserting new data or attempting to find strings in the graph. Figure 50 in the center is the C code to define a datum in the graph. Notice that the FBF signature (sig), the hash bucket code (bkt), the Soundex code (sdx) and the length of the string (len) are stored along with the string value (val). These vertices are stored in an array. Storing this data uses a little more memory but saves time over

computing it on-the-fly for small data. As the data grows, this method of storage will compress the data by eliminating redundancy. Figure 50 on the right is an edge list the array1 and array2 arrays contain the indices of the two related strings from the data array. The index in array1 is always numerically less than the index in array2. The sorted1 and sorted2 arrays comprise a bidirectional sorted index that enables fast binary searches on related links between strings. The match variable is a binary checklist that record how two strings are related. The count and length variables record the number of links inserted and the pre-allocated space for future inserts. The mode variable records whether the link array is for a data-to-data link or a record-to-data link, which is discussed below. There are two scenarios to search the graph: one where the search term is in the graph and one where it is not. If the search term is in the graph, all related data is extracted and returned from the FNLnk edge—no string comparison is necessary because these are stored and retrieved. If the search term is not in the graph, *PGLFPDL* is used to quickly find all approximate matches and the Soundex codes are compared to find all phonetic matches. *PGLFPDL* is also used during the insert operations into the data graph.




```
// Define DATA struct
typedef struct
{
    char *val;
    unsigned int sig;
    unsigned int bkt;
    char sdx[5];
    unsigned int len;
} DATA;
```

```
// Define LINK_ARRAY struct
typedef struct
{
    int *array1;
    int *array2;
    int *sorted1;
    int *sorted2;
    unsigned char *match;
    int count;
    int length;
    int mode;
} LINK_ARRAY;
```

Figure 50: String relationship graph and structures

Figure 51 below is an example of a populated graph, which contains alias, approximate matches for $k \leq 1$ and phonetic matches as found by the American Soundex. Notice that some string pairs are related in more than one way. The idea is that each unique string

pair only needs to be compared once. If a second occurrence of a string is inserted, no comparison is necessary. This model facilitates the removal of all inter record redundancy given a field in the table.

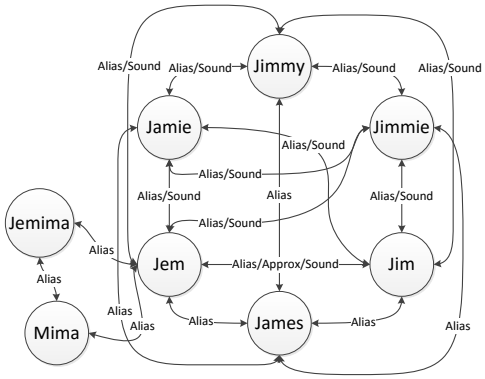


Figure 51: String relationship example

To make this system useful for RL a Record graph is linked to the data graph as shown in Figure 52 below on the left. The Record vertex mostly contains data that refers back to the database and record(s) in which the data strings in the First Name vertices are stored to allow the linkage system to generate groups database/records that match under the RL rules. This facilitates query searches and the generation of reports that use the linked data. Figure 52 on the right shows the definition of a Record vertex. The agency represents the agency database that contains the person's data and the agency_id is the unique identifier within the database for that person. The person's gender is stored with the record because a person should only have one and the only acceptable values are 'F' and 'M'. The RecLnk edge stores data about how records are related. It is the same data structure as is used with the data graph except in record mode, the array1 refers to the record and array2 refers to the data. Person *A* may be a sibling, child or parent of person *B*. This is useful for family linkage, which basically combines RL with social networks. The RecFN edge connects each Record with its corresponding Data. This structure

allows multiple Records to be associated with a single datum or multiple data values for a field to be associated with each Record. A date can be added to RecFN to allow temporal sorting of data. If a woman has been married more than once, every last name can be sorted as it has changed over time. The same can be said for addresses and phone numbers, which are also likely to change over time.

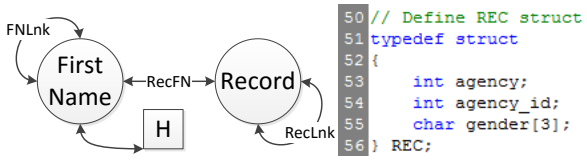


Figure 52: Record relationship graph and structure

Figure 53 below shows that record index 100 is associated with "James" and that "James" is related to other data within the graph.

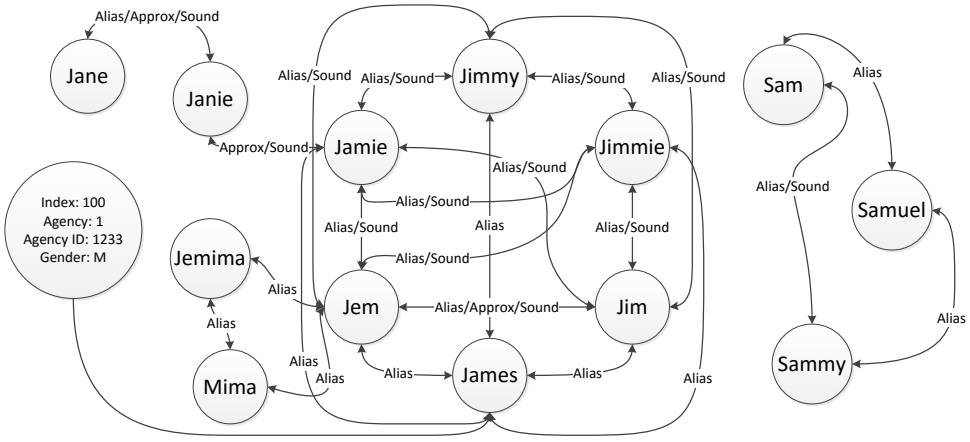


Figure 53: Record and relationship graph example

Figure 54 below shows a set of record vertices that are associated with compound records from agencies 1 and 2. Notice that the RecLnk edges between the record vertices act as a social network graph embedded into the Record graph. This can facilitate family linkage and the use of family members' data as a context to increase confidence and accuracy for the RL process. The "Keep together" and "Keep apart" edges are human entered

corrections that keep records known to be the same person together and keep apart records that are known to be from different people. This allows the domain and other experts to share their knowledge and experience about the data with the RL system.

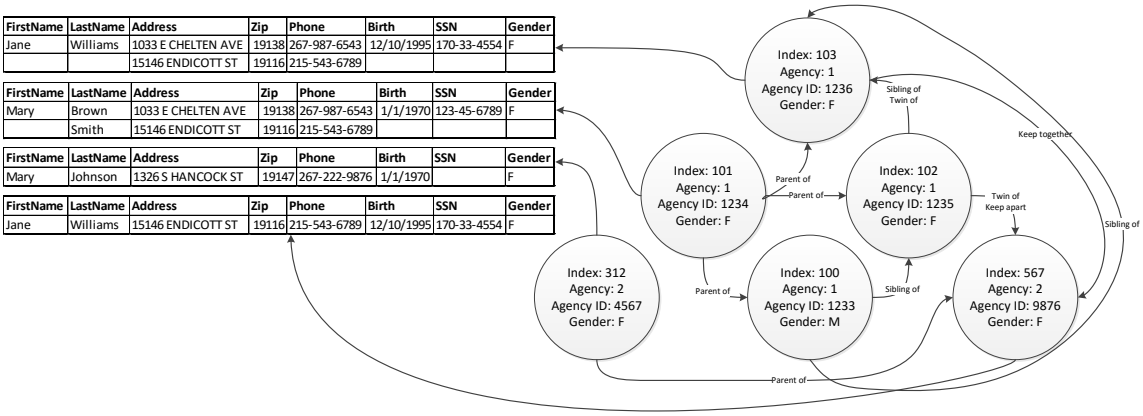


Figure 54: Record graph and associated compound records

CHAPTER 7

AGGREGATE, LINK AND ITERATIVE MATCH (ALIM)

Aggregate Link and Iterative Match (ALIM) is an in-memory, temporal data graph-based RL system designed to efficiently and effectively process temporal demographic (and possibly other) data that is generally observed in health and social sciences databases.

7.1 ALIM Prototype 1

The first prototype was used during the final experiments run for the CARES assessment. There were some memory management issues and a slight false positive RL matching problem that were fixed in the 2nd prototype. Figure 55 shows the basic schema for ALIM Prototype 1. The model shows 6 vertices (large outer circles—contain the data string value and information used for approximate and phonetic matching) that contain demographic data, one node that defines the record (large center circle—contains the origin agency, the agency's UID and the clients gender). The edges (small circles with leaders) record how data strings are related and link the data with the records.

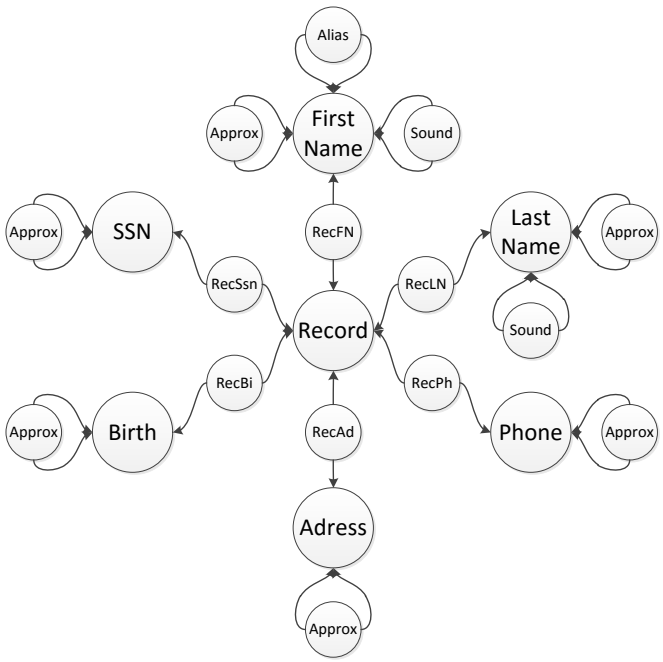


Figure 55: ALIM Prototype 1

7.1.1 Experiments for ALIM Prototype 1

We created synthetic records for our RL experiments. There were 11 datasets containing 10,000 records each. The records match by index, meaning that record *n* in one dataset matches record *n* in all datasets. These were constructed with clean records and errors, inconsistencies and missing data were injected to mimic the condition of the health departments real data. The synthetic records were constructed randomly using the data used in the string comparison experiments. The data are:

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)

There were 5,000 male and 5,000 female records. First names were chosen to match gender. The records were constructed with the following fields:

1. First Name
2. Last Name
3. Address
4. Phone
5. Gender
6. Birth Date
7. Social Security Number

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.

7.1.2 Results for ALIM Prototype 1

The experimental results for this model are shown in Table 37 below. The best TP accuracy was obtained in the middle column A8S4I3Al for the synthetic CARES data. The original CARES method had 61,705 true positives and 0 false positives, resulting in a very low 12.637% sensitivity. The best ALIM result had 120,132 true positives, 967 false positives, and a 27.884% sensitivity. The column titles are described in the Table 36 below. OCM is the method used by the local health department. The A8 as the first 2 characters means that 8 points were added for an approximate match for the first name, last name, address and phone fields. The S4 means that 4 points were added for a Soundex match for the first name and last name fields. I2 means that 2 match iterations

were performed and I3 means that 3 were performed. An iteration is basically a re-matching of a record with the results of the previous match incorporated into the query exemplar. The Al on the end of the column title means that the alias match function was active.

Table 36: Experimental configurations

OCM	Original CARES Method
A8S4I2Al	2 iterations and alias search active
A8S4I3Al	3 iterations and alias search active
A8S4I2	2 iterations and alias search not active
A8S4I3	3 iterations and alias search not active

Table 37: Previous experiments

	OCM	A8S4I2Al	A8S4I3Al	A8S4I2	A8S4I3
True positives	61,705	117,266	120,132	117,011	119,896
False negatives	488,295	432,734	429,868	432,989	430,104
False positives	0	246	967	286	1,041
True negatives	6,049,395,000	6,049,394,754	6,049,394,033	6,049,394,714	6,049,393,959
Accuracy	99.992%	99.993%	99.993%	99.993%	99.993%
Sensitivity	12.637%	27.083%	27.884%	27.006%	27.809%
Specificity	100.000%	100.000%	100.000%	100.000%	100.000%
S_Accuracy	56.318%	63.542%	63.942%	63.503%	63.904%
Read time	0:00:01	0:08:11	0:08:08	0:07:26	0:07:58
Match time	0:50:28	0:16:33	0:20:46	0:04:21	0:06:04

7.2 ALIM Prototype 2

The String Relationship Graph as described above is a crucial part of ALIM version 2. The second prototype's data model had been updated to be more compact, accurate and useful. The new model differs from the old model:

1. New standardized data and link nodes

2. Only one relationship link per data node for alias, approximate and phonetic matches
3. Relationship link for records allow inclusion of family relationships and social networks
4. New length filter, FBF and *PDL* optimization (*LFPDL*) for edit distance
5. New unmatchable record optimization eliminates unnecessary RL comparison
6. New index optimization for existing records (binary searching)
7. Memory leak fixed
8. Link index problem fixed that harmed previous ALM results
9. SSN and Birthdate both missing optimization (required fields for RL matching)

Figure 56 below shows a schema for the new model. The indexing system was also made more efficient. The *LFPDL* string comparison substantially decreases data insertion into data graphs. Data relationship links were aggregated to one link and a link was added to the Record node to store Record relationships. The *Lnk edges contain all alias, approximate and sound match data in the bits of an unsigned char. The RecLnk can contain data to link records with other records that are related (i.e. parents, siblings, spouses, records known to match or not match etc.). This will allow family matching and the use of family data for linking entities. We forgo an elaborate description of these prototypes and will describe all the features for the newest prototype in the Proposed Methodology section.

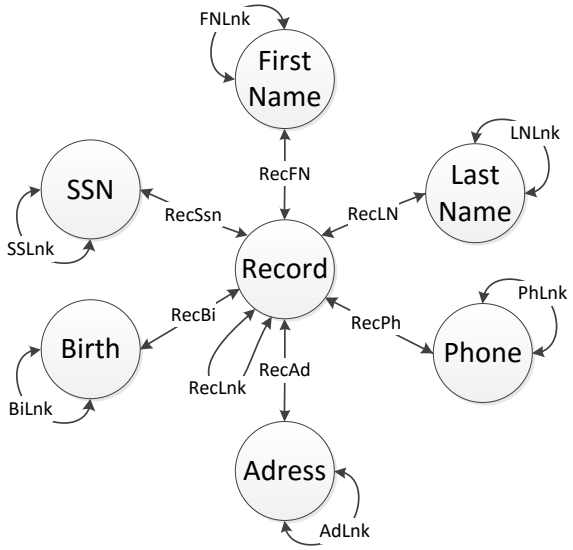


Figure 56: ALIM Prototype 2

7.2.1 Experiments for ALIM Prototype 2

The data for these experiments were the same as used in ALIM Prototype 1. 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.

7.2.2 Results for ALIM Prototype 2

The new models results are below. True positives increased to 166,187 (+46055)and false positives dropped to 0 when compared to the results for ALIM prototype 1. The sensitivity increased to 30.216% (+2.332%) The new model match time increased from 20min 46sec to 45min 19sec but the read/build time is the same. The unmatchable record optimization effects are shown in the second column. A record is considered unmatchable if it cannot match itself under a given match rule. There are 29,683 unmatchable records in the 11 datasets, which resulted in 256,805 false negatives due to lost unmatchable record pairs. If unmatchable record pairs are considered true negatives,

the sensitivity increases to 56.681%. ALIM records unmatchable pairs in the hope that more data will arrive in the future to change its state to matchable.

Table 38: New experimental with unmatchable records

	ALMv2	With UM
True positives	166,187	166,187
False negatives	383,813	127,008
False positives	0	0
True negatives	6,049,395,000	6,049,651,805
Accuracy	99.994%	99.998%
Sensitivity	30.216%	56.681%
Specificity	100.000%	100.000%
S_Accuracy	65.105%	78.340%
Read Time	0:08:11	0:08:11
Match Time	0:45:19	0:45:19
Unmatchable Recs	29,683	29,683
Unmatchable Pairs	256,805	256,805

The 56% sensitivity may seem low but this dataset was made to model the missing data in the CARES system. This data does not have aggregated records—each data set has only one record per entity. Aggregated records will make ALIM results much more accurate when compared to methods that ignore the dependence between some records. The new results show significant improvement over the previous results. The results below are the old results from the ALIM report to the Dept. of Health. The experiment compared 220,000 synthetic records for 110,000 Agency/Entities. There were two records for each entity in each agency. ALM really shows its efficiency and effectiveness in these results. The table on the left is the results for the existing CARES method (OCM) and the right is the original ALIM results. Since OCM treats records as independent, there are 4 times as many pairwise comparisons.

Table 39: Experimental results for OCM and ALIM

OCM			A4S0	
Possible pairs	24,199,890,000		Possible pairs	6,049,945,000
Possible matches	2,310,000		Possible matches	550,000
Possible nonmatches	24,197,580,000		Possible nonmatches	6,049,395,000
True positives	260,967		True positives	392,130
False negatives	2,049,033		False negatives	157,870
False positives	0		False positives	53
True negatives	24,197,580,000		True negatives	6,049,394,947
Accuracy	99.992%		Accuracy	99.997%
Sensitivity	11.297%		Sensitivity	71.296%
Specificity	100.000%		Specificity	100.000%
S_Accuracy	55.649%		S_Accuracy	85.648%
Read time	0:00:02		Read time	0:40:52
Match time	3:09:15		Match time	0:31:41

The new ALIM 2 results are shown below and are a significant improvement. The A4S0I2Al is the same basic configuration as A4S0 above. There are 84,008 more true positive matches and 48 less false positives. Also note that the increased accuracy with respect to OCM has no runtime cost—it's actually faster even with multiple match iterations. This is due to the elimination of redundant and unnecessary comparisons of data fields and records. It appears that the experiments with 2 iterations yield the most true positives.

Table 40: ALIM v2 results

	Less restrictive points			More restrictive points		
	A8S4I3Al	A8S4I2Al	A8S4I1Al	A4S0I3Al	A4S0I2Al	A4S0I1Al
Possible pairs	6,049,945,000	6,049,945,000	6,049,945,000	6,049,945,000	6,049,945,000	6,049,945,000
Possible matches	550,000	550,000	550,000	550,000	550,000	550,000
Possible nonmatches	6,049,395,000	6,049,395,000	6,049,395,000	6,049,395,000	6,049,395,000	6,049,395,000
True positives	467,862	476,138	431,594	467,862	476,138	431,593
False negatives	82,138	73,862	118,406	82,138	73,862	118,407
False positives	34	5	0	34	5	0
True negatives	6,049,394,966	6,049,394,995	6,049,395,000	6,049,394,966	6,049,394,995	6,049,395,000
Accuracy	99.999%	99.999%	99.998%	99.999%	99.999%	99.998%
Sensitivity	85.066%	86.571%	78.472%	85.066%	86.571%	78.471%
Specificity	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
S_Accuracy	92.533%	93.285%	89.236%	92.533%	93.285%	89.236%
Read time	42:47	44:18	44:02	43:42	43:12	44:11
Match time	1:22:27	1:05:48	0:25:22	1:22:36	1:05:10	0:25:47
Total time	2:05:14	1:50:06	1:09:24	2:06:18	1:48:22	1:05:58

The table below shows the results from the previous table but considers unmatchable record pairs as false positives. Note the increase in sensitivity. The unmatchability of a record is decided based on missing data from a given record.

Table 41: ALIM v2 results with unmatchable records

	Less restrictive points			More restrictive points		
	A8S4I3AI	A8S4I2AI	A8S4I1AI	A4S0I3AI	A4S0I2AI	A4S0I1AI
Possible pairs	6,049,945,000	6,049,945,000	6,049,945,000	6,049,945,000	6,049,945,000	6,049,945,000
Possible matches	550,000	550,000	550,000	550,000	550,000	550,000
Possible nonmatches	6,049,395,000	6,049,395,000	6,049,395,000	6,049,395,000	6,049,395,000	6,049,395,000
True positives	467,862	476,138	431,594	467,862	476,138	431,593
False negatives	48,226	39,950	84,494	48,226	39,950	84,495
False positives	34	5	0	34	5	0
True negatives	6,049,428,878	6,049,428,907	6,049,428,912	6,049,428,878	6,049,428,907	6,049,428,912
Accuracy	99.999%	99.999%	99.999%	99.999%	99.999%	99.999%
Sensitivity	90.655%	92.259%	83.628%	90.655%	92.259%	83.628%
Specificity	100.000%	100.000%	100.000%	100.000%	100.000%	100.000%
S_Accuracy	95.328%	96.130%	91.814%	95.328%	96.130%	91.814%
Read time	42:47	44:18	44:02	43:42	43:12	44:11
Match time	1:22:27	1:05:48	0:25:22	1:22:36	1:05:10	0:25:47
Total time	2:05:14	1:50:06	1:09:24	2:06:18	1:48:22	1:05:58
Unmatchable Recs	3,448	3,448	3,448	3,448	3,448	3,448
Unmatchable Pairs	33,912	33,912	33,912	33,912	33,912	33,912

7.3 ALIM Prototype 3

This section discusses our latest Record Linkage model: Aggregate Link and Iterative Match (ALIM) Prototype 3. Our new data model is proposed as a heterogeneous in-memory data graph structure with search operations similar to database queries but with a focus on approximate string and record matching as opposed to exact or partial string or record matching. Figure 57 below shows an overall schema based on the demographic fields shown in Figure 21. The results of RL searches on the data graph structure are

converted into comparison vectors, which can be processed using deterministic, probabilistic or data mining record matching algorithms.

Since vast majority data entry errors are typographical, character-by-character, edit-distance algorithms are the most effective typographic error detectors. A fast short-string comparison algorithm PSH has been developed and is used in this prototype. Secondly, unlike traditional databases, approximate matching is a non-trivial compute intensive and data intensive function, RL marks a departing point from the single truth model traditional databases. Figuratively speaking, ALIM is a dedicated memory resident "co-processor" with its own sustainable processing/storage layer in addition to the multiple database sources.

One significant difference is that these methods were tested on a real big dataset from a system currently in use by a local health department.

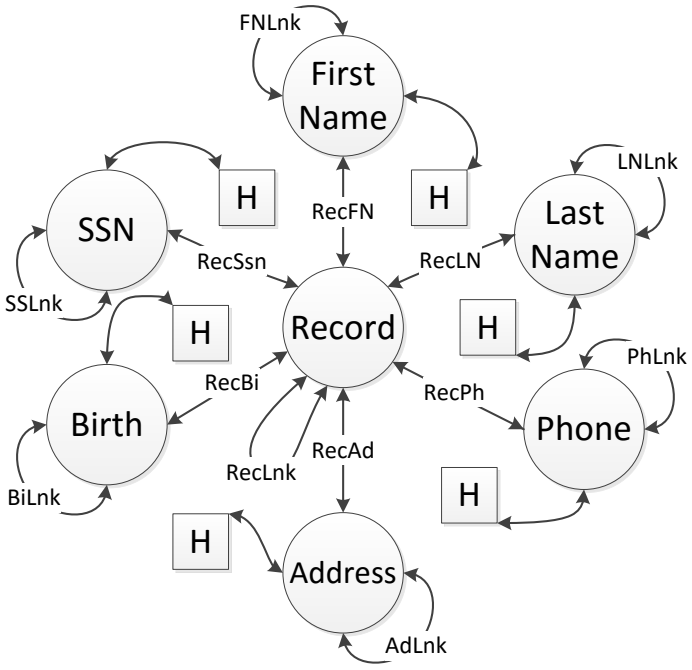


Figure 57: ALIM Prototype 3 Model

7.3.1 ALIM Graph Components and Initialization

The model is a system of interconnected graphs. The circular nodes or vertices in Figure 57 contain the RL systems data. The box-shaped vertices are hash tables that are used to optimize approximate string searching in the vertices of the data subgraphs. All of the bidirectional leaders or edges are the relationships between data. If the leader is a loop, it records how the data in a vertex is related to itself. If the edge connects a data vertex with a record vertex, it links records with their data. If the edge connects a data vertex to a hash table, it is used for approximate string searching. With the exception of the gender variable, all inter-record redundancy is eliminated, compressing the graph to the least possible number of vertices—similar to a 5^{th} or 6^{th} normal form database [Ling, Tompa and Kameda 1981] [Date, Darwen and Lorentzos 2003]. Gender is stored in the Record because of the limited number of valid values: "" (missing data), "F", "M" and "FM" (FM meaning that for some reason both M and F have been observed in the data for an individual—a contradiction that is treated as missing data). This compression makes it much less expensive to record the relationships between unique data vertices within each subgraph.

Figure 58 shows the FirstName and Record subgraphs. The FirstName subgraph, $G_{FN} = (FirstName, FNLnk)$, is a string relationship graph that represents all the data for first names, where FirstName is the list of vertices, each element of which contains a value, such the name "James" and FNLnk represents the edges, which relates one datum to another, such as "James" is an alias of "Jim" or "Jim" is approximately spelled like "Kim" because they differ by 1 edit (all edges in Figure 57 ending in Lnk are similar to this). These edges are basically similar to a database table with a many-to-many relation

from itself onto itself. The RecFN is a many-to-many relationship between the data in FirstName and Record that connects the two subgraphs (all edges in Figure 57 beginning with Rec are similar to this). The Record subgraph, $G_{Rec} = (Record, RecLnk)$, represents all the data input from a single database about a single entity from each agency. The Record contains data about the data source and its unique identifier and the gender. The data source is the agency that contains the record combined with the unique identifier within the agency, which comprise an agency/entity. Agency/entities from each agency are compared to produce enterprise/entities during the RL process. RecLnk represents how each Record is related to connected records (i.e. parent/child, siblings, spouses, etc.). The edge, vertex and relation lists are implemented as arrays of structures but can also be stored in a database. However, initializing and updating the model may be difficult in a database due to a lack of byte and bit level access to data. Our method requires this access to speed up the initialize and update processes (see FBF below and Hybrid Signature Hashing with Bitwise Neighborhood Generation, Length Filter, FBF and *PDL* above).

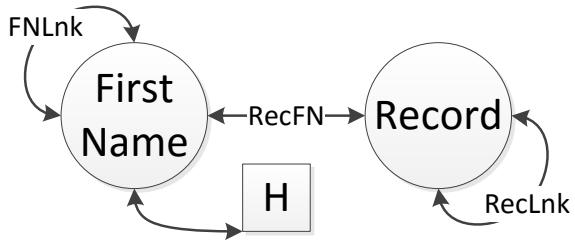


Figure 58: FirstName and Record Graphs

The hash table is precomputed with a quick single pass over the data. The relationships are all calculated and stored as each record and the record's data are inserted into the graph.

There are three phases in ALIM: a) building a demographic data graph model, b) querying the model, and c) updating the model. For each query, the system should be able to quickly traverse the entire graph in all dimensions before concluding a potential match. Figure 57 shows a typical ALIM demographic data model. Unlike relational databases, the ALIM data model stores all data evidence for each field in the client's record. For example, if a person changes their last name twice, all names will be included in the compound record with multiple entries in the cycle "RecLN" edge list. The graph also stores relations between data values defined by multiple approximation matching algorithms. These include approximate spelling, phonetic and alias matches, which are pre-computed when records are inserted into the graph data model. This allows fast query performance during RL. For example, the "FNlnk" (first name link) edge links "James", "Jim", "Jimmy" and "Jamie" as synonymous terms, which will be considered as exact matches. The graph also stores record and family relationships between entities in the RecLnk (record source link) edge list. This allows using family data to increase the confidence of ER. If data is missing from an entity's record that makes it unmatchable, family data can be substituted to use in RL processing. This also allows ALIM to leverage social network data. The "RecLnk" edge also allows users to incorporate their knowledge of the data within the matching system. Records that are known to be from the same entity can be forced to link and records from entities that are not from the same entity can be forced apart using a "Keep-together/Keep-apart rule".

7.3.2 ALIM Matching

Because ALIM stores relationships between data strings, string comparison need not performed while matching records unless some or all of the data in the query has not been

inserted into the system of graphs. A match table with one row for each entity/agency in the system is used to tabulate the field-by-field agreement between the query record and every entity/agency in the system. Immediately after the ALIM graph system is initialized with all of the enterprise's records, pre-matching of all entity/agency records in the system can be performed to speed up queries for agency/entities in the system. Most queries are likely to already be in the system.

7.3.3 ALIM: Architecture: The next-generation Entity Resolution (ER)

Table 42 shows that each entity has a compound record constructed from all data relevant to the entity within a given dataset.

Table 42: Multiple record entries for the same entity

ID	SSN	Birth	Gender	Phone	Address	Last Name	First Name
12	123456789	01012000	M	2151234567	1234 N BROAD ST	SMITH	JAMES
12	123457689	01012000	M	2677654321	3456 PINE ST	SMIHT	JIM

Figure 59 shows a compound record (left) and its data (right). These inconsistencies are very common in human service systems. There is also a temporal dimension in the proposed ER system that includes dates to each data point (not shown in Figure 59). This allows exploitation of temporal changes to enhance matching confidence.

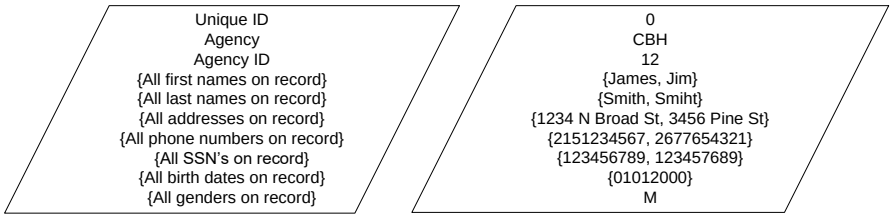


Figure 59: Compound Record

In the query phase, all related fields are traversed based on multiple matching criteria. Figure 60 shows approximate spelling, phonetic and alias matches on First Names.

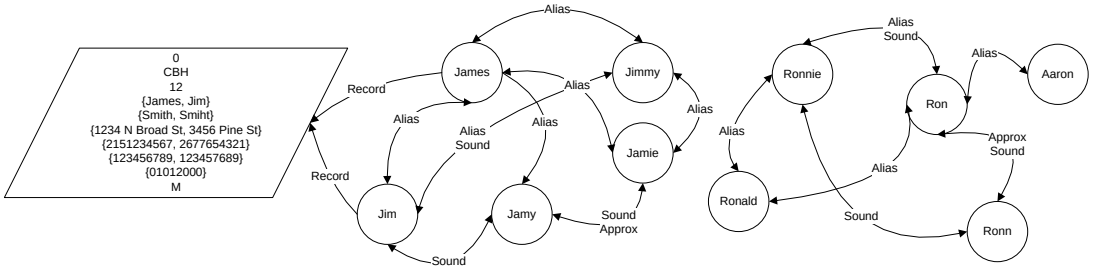


Figure 60: ALIM Query Process

Table 43 shows an example of a point system for the data graph from Figure 57. Note that there are penalties for immutable fields that generally should not change, for example birth date and gender. The penalty decreases the possibility of false positive matches. Two records are considered a match if the maximum match point values exceeds a user-defined threshold. Note that an alias match is technically identical to an exact match. Table 44 shows an example of comparison vectors (Vt) and their sum in the Total column, where Record 12 is considered a match if the threshold is less than or equal to 48. ALIM is also capable of iterative matching to force a transitive closure on RL (i.e. if records $r \sim q$ and $q \sim p$ then $r \sim p$). Unlike generic machine learning systems, an ALIM match can be readily interpreted by a human user by examining the comparison vectors. ALIM's in-memory data graph model compresses the data by removing redundancy and increases efficiency by pre-computing approximate matches between data.

Table 43: Summed comparison vectors

	FirstName	LastName	Address	Phone	Social	Birth	Gender
Exact	8	8	8	8	22	11	2
Approx	8	8	8	8	11	6	
Sound	4	4					
Penalty					11	6	5

Table 44: Weighted Point System

	FN	LN	Ad	Ph	So	Bi	Ge	Total
1	8			8				16
2		4						4
3						11	2	13
5	4		8					12
6					22		2	24
7		4					2	6
9				8				8
11		8				11		19
12	8	8		8	22		2	48
13			8					8
15	8							8
16			8				2	10

7.3.4 Experiments for ALIM Prototype 3

The data in these experiments are real data from the CARES RL system. The first experiments data includes two 10,000 record datasets that hand selected from CARES data known to match by index. The second data set is real data from the health department's three largest datasets containing a total of 8,376,754 tuples. The experiments were processed on a dual quad-core Xeon server with 48GB of RAM.

7.3.5 Results for ALIM Prototype 3

Table 45 below shows the results from the real data experiments. The first result column, OCM, is a deterministic algorithm that uses the Soundex for approximate matching and no blocking, currently in use by a local health department, the second is ALIM with a threshold of 25 points (liberal analysis), and the third is ALIM with a threshold of 35 points (more restrictive analysis). ALIM has higher true positives but with higher false positives. However, we discovered after further investigation that these datasets contained more than one record per person—it also has a deduplication problem as do most of the datasets. This was discovered by desk checking the data for false positives and revealed no actual false positives. The FPs were actually TPs. The S_Accuracy is calculated as $S\_Accuracy = (Sensitivity + Specificity)/2$.

Table 45: ALIM results

	OCM	ALM25	ALM35
True positives	8,336	8,907	8,646
False negatives	1,664	1,093	1,354
False positives	302	406	355
True negatives	199,979,698	199,979,594	199,979,645
Accuracy	99.999%	99.999%	99.999%
Sensitivity	83.360%	89.070%	86.460%
Specificity	100.000%	100.000%	100.000%
S_Accuracy	91.680%	94.535%	93.230%

Our current experiment has processed a real dataset containing 8,376,754 records. OCM treats the individual tuples in compound records as completely independent and had to process 35,084,999,599,881 record pairs (i.e. a matrix diagonalization $\big(n \times (n-1)\big)/2$ for n records). It would have taken over 50 weeks to complete this experiment so it was terminated. We added a blocking scheme that required at least one field to be an exact match to be considered a candidate for the OCM algorithm to decrease the runtime. It took the same deterministic algorithm used in the previous experiment with blocking added over 710 hours to complete. ALIM's 5^{th} normal form data graph reduced the dataset to 1,733,921 compound records and further compressed the data by eliminating all data redundancy. Table 46 shows the number of records inserted into each algorithm and the runtimes. ALIM also has an optimization that considers a record unmatchable if it cannot mat itself given the match rule. The Unmatchable column shows that ALIM eliminated 68,608 records that did not contain enough nonempty data fields to be matchable. It took ALIM less than 42 hours to build the data graph, including all approximate string matching, and less than 300 hours to complete the record matching process. ALIM is twice as fast without the use of blocking.

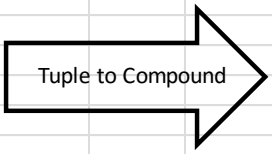
Table 46: Inserts and runtimes by method

Method	Hours	Minutes	Inserts	Unmatchable
OCM	710	27	8,376,754	
ALIM	341	28	1,733,921	68,608

The results from the current RL experiment between OCM and ALIM cannot be directly compared because of the structural differences between these methods. The ALIM matches had to be expanded to individual tuple-based matches to be comparable to the OCM matches. To expand the ALIM results, the records in the original data files were numbered by their respective compound record. OCM performs a tuple-based search, comparing all individual pairs of records. ALIM uses a 5th normal form graph-based approach that immediately combines tuples that originate from the same agency unique identifier into a compound record arrangement, which substantially reduces the number of records to be matched. Table 47 shows the modifications that converted the raw OCM input data (OCM_{Tuples}) to a tuple-based representation for ALIM expanded ($ALIM_{Tuples}$) results. Notice that the Insert number column from OCM was converted to the Compound Record insert number (CompRec) for ALIM to create the expanded ALIM file, which is a relation between each method's results by index.

Table 47: Tuple to compound records

Insert	Agency	AgencyID
0	1	1
1	1	1
2	1	1
3	1	1
4	1	10
5	1	100
6	1	1000
7	1	1000
8	1	1000



CompRec	Agency	AgencyID
0	1	1
0	1	1
0	1	1
0	1	1
1	1	10
2	1	100
3	1	1000
3	1	1000
3	1	1000

There are two types of matches in $ALIM_{Tuples}$ results. Matches by definition ($ALIM_{Defined}$) are the multiple records for an entity that are known to be true positive matches because the agency's unique identifier matches, mentioned above. Search matches ($ALIM_{Search}$) are matches that were produced by the matching algorithm and expanded using $ALIM_{Tuples}$. The $ALIM_{Tuples}$ were converted directly into $ALIM_{Defined}$. Notice that the 4 tuples for CompRec 0 from indices 0 to 3 produced 6 defined matches from $(0,1)$ to $(2,3)$. We used 99 points as a sentinel value for $ALIM_{Defined}$ matches. This operation produced a file containing only true positive matches because they are known to be from the same entity within a given agency, which can be used as a limited ground truth. It should be noted that the data were sorted lexicographically by unique ID and not numerically.

Table 48: ALIM tuples to ALIM defined matches

Index	CompRec	Agency	AgencyID
0	0	1	1
1	0	1	1
2	0	1	1
3	0	1	1
4	1	1	10
5	2	1	100
6	3	1	1000
7	3	1	1000
8	3	1	1000

Compound to Tuple

Index 1	Index 2	Points
0	1	99
0	2	99
0	3	99
1	2	99
1	3	99
2	3	99
6	7	99
6	8	99
6	9	99

There were 170,446,671 OCM matches with a 35 point threshold. ALIM had 872,115 compound record matches with a 35 point threshold. The $ALIM_{Search}$ matches were extracted by using the $ALIM_{Tuples}$ tuples to convert the ALIM compound matches ($ALIM_{Compound}$) to $ALIM_{Search}$ matches. Table 49 shows that Index 33, CompRec 6, which is Agency 1, AgencyUID 1000000, has 6 matches in the $ALIM_{Search}$ list on the

right. Also note that Indices 33 to 41 are all the same entity because they share the same CompRec.

Table 49: Convert ALIM tuples to ALIM search matches

Index	CompRec	Agency	AgencyID
33	6	1	1000000
34	6	1	1000000
35	6	1	1000000
36	6	1	1000000
37	6	1	1000000
38	6	1	1000000
39	6	1	1000000
40	6	1	1000000
41	6	1	1000000

Compound to Tuple

Index 1	Index 2	Points
33	7994240	59
33	7994241	59
33	7994242	59
33	7994243	59
33	7994244	59
33	7994245	59
34	7994240	59
34	7994241	59
34	7994242	59

In order to compare $ALIM_{Defined}$ and $ALIM_{Search}$ to OCM, the OCM results had to be converted into similar datasets. To extract $OCM_{Defined}$ from OCM_{Raw}(all matches found with the OCM algorithm on the data), set intersection, as $OCM_{Defined} = OCM_{Raw} \cap ALIM_{Defined}$ was performed. To extract OCM_{Search} from OCM_{Raw}, set subtraction, as $OCM_{Search} = OCM_{Raw} \ominus OCM_{Defined}$, was performed. There were 167,740,521 expanded defined matches for ALIM and 43,861,648 expanded search matches for a total of 211,602,169. ALIM produced 41,155,498 or 24.15% more total tuple-based matched pairs than OCM. OCM missed 3,107,897 defined matches. ALIM produced 38,047,601 or 7.54 times the search matches than OCM. Table 50 contains the resulting counts.

Table 50: Counts of matches

Method/Type	Count
OCM_{Raw}	170,446,671
$OCM_{Defined}$	164,632,624
OCM_{Search}	5,814,047
$ALIM_{Compound}$	872,115
$ALIM_{Defined}$	167,740,521
$ALIM_{Search}$	43,861,648
$ALIM_{All}$	211,602,169

The results in Table 51 below show counts for records matched by one method and missed by the other (i.e. $|ALIM_{Search} \ominus OCM_{Search}| = 38{,}368{,}945$).

Table 51: Comparison of match types

Set operation	Count
$\lvert ALIM_{Search} \ominus OCM_{Search} \rvert$	38,368,945
$\lvert OCM_{Search} \ominus ALIM_{Search} \rvert$	321,345
$\lvert ALIM_{Defined} \ominus OCM_{Defined} \rvert$	3,107,897
$\lvert OCM_{Defined} \ominus ALIM_{Defined} \rvert$	0

The following analysis shows the counts of the search match results for ALIM and OCM by point value. Table 52 shows three rows of results with point scores from 35 to 75 points for each methods match scores. The maximum score for OCM is 75 points and the maximum score for ALIM is 67 points. ALIM's point system is more conservative than OCM's. The Difference rows show the point differences between methods with positive values showing more matches for ALIM and negative values showing more matches for OCM. Notice that 32.94 million of the 43.86 million search matches are at the 35 point threshold. This suggests that many of the records are missing some data or contain some inconsistent data.

Table 52: Match points and difference

Points	35	36	37	38	39	40	41
ALIMsearch	32,943,476	770,731	2,115,060	18,102	0	20,372	55,347
OCMsearch	467,540	21,397	98,689	30,394	55,409	223,597	189,117
Difference	32,475,936	749,334	2,016,371	-12,292	-55,409	-203,225	-133,770
Points	42	43	44	45	46	47	48
ALIMsearch	189,626	1,468,548	43,710	422,488	1,712	0	10,008
OCMsearch	615,714	38,546	12,577	178,314	10,030	372,911	5,885
Difference	-426,088	1,430,002	31,133	244,174	-8,318	-372,911	4,123
Points	49	50	51	52	53	54	55
ALIMsearch	42,038	26,499	1,010,458	190,835	0	313	0
OCMsearch	13,423	538,038	38,238	68,595	6,036	1,010	2,031,943
Difference	28,615	-511,539	972,220	122,240	-6,036	-697	-2,031,943
Points	56	57	58	59	60	61	62
ALIMsearch	5,891	159,280	0	3,034,579	68,160	0	0
OCMsearch	1,672	79,816	593	1,804	158,443	3,023	5,111
Difference	4,219	79,464	-593	3,032,775	-90,283	-3,023	-5,111
Points	63	64	65	66	67	68	69
ALIMsearch	0	0	33,345	0	1,231,070	0	0
OCMsearch	360	91	477,384	109	5,171	0	0
Difference	-360	-91	-444,039	-109	1,225,899	0	0
Points	70	71	72	73	74	75	
ALIMsearch	0	0	0	0	0	0	
OCMsearch	26,362	0	0	0	0	36,705	
Difference	-26,362	0	0	0	0	-36,705	

The distribution in Table 53 below shows 3 bins of results for the previous point counts and totals. Notice that, again, the ALIM matches are pooled within the lower point values. The last column is the mean point value for all matches.

Table 53: Point counts in 3 bin distribution

Points	35-48	49-62	63-75	Total	Point Mean
ALIMsearch	38,059,180	4,538,053	1,264,415	43,861,648	38.70
OCMsearch	2,320,120	2,947,745	546,182	5,814,047	50.08
Difference	35,739,060	1,590,308	718,233	38,047,601	

Table 54 shows the ALIM threshold increased to 40 points and the OCM threshold at 35. Any false positive matches are most likely below this threshold because the Point Mean has increased from 38.7 to 54.62. ALIM produced a total of 2,200,232 more matches, even with a substantially increased threshold. The higher the threshold, the less likely we have false positive matches.

Table 54: Point counts with ALIM threshold at n=40 points, OCM at n=35 points

Points	n-48	49-62	63-75	Total	Point Mean
ALIMsearch	2,211,811	4,538,053	1,264,415	8,014,279	54.62
OCMsearch	2,320,120	2,947,745	546,182	5,814,047	50.08
Difference	-108,309	1,590,308	718,233	2,200,232	

The search match results for $ALIM_{Search}$ and OCM_{Search} were checked for possible duplicate entities (DE) and possible table matches (TM) by determining whether or not the matches are contained within the same agency's dataset. We call these possible DE and TM because some may be false positive matches. In large datasets, it's impossible to determine the exact number of true positive and false positive matches. If we could, the method to determine this would be a perfect solution to the problem, which is theoretically impossible. The tuples were inserted in order of occurrence from each of the agencies, which means that we can resolve an agency by tuple insert number. Table 55 shows the insert boundaries for the enterprise's respective agencies.

Table 55: Agency boundaries

Agency	First	Last
Agency1	0	7,623,994
Agency2	7,623,995	8,040,317
Agency3	8,040,318	8,376,753
ALL	0	8,376,753

The candidate TM and DE counts are shown in Table 56 below. Notice that the DE for $ALIM_{Search}$ is very high, which suggests the probability of a deduplication problem in the data (or false positives at lower threshold values).

Table 56: Candidate TM and DE counts

	TM	DE
ALIM	10,799,534	33,062,114
OCM	5,384,765	429,284

Table 57 shows the distribution for the candidate TMs and DEs. Both methods DEs and TMs decrease as the threshold increases, except for OCMtp, which peaks at the 49-62

threshold range. Notice that ALIM and OCM both have significant DEs in the 63-75 point range. Because these are matches from an agency onto itself, it suggests the probability of intra-agency duplicates. ALIM had nearly 200,000 matches with a point score of 67 and OCM had 4,041 with a point score of 75. These scores suggest exact matching duplicates within the same agency.

Table 57: Distribution for candidate TM and DE

Points	35-48	49-62	63-75	Total	Point Mean
ALIMtm	5,862,786	3,876,105	1,060,643	10,799,534	46.89
ALIMde	32,196,394	661,948	203,772	33,062,114	36.02
OCMtm	1,990,552	2,868,429	525,783	5,384,764	50.57
OCMde	329,568	79,316	20,399	429,283	43.93

Table 58 shows the ALIM threshold increased to 40 points and the OCM threshold at 35. This substantially lowered the DE for ALIM. With a Point Mean of 48.42, few of the ALIM DE are likely to be false positives.

Table 58: Distribution for candidate TM and DE with ALIM at 40 points

Points	n-48	49-62	63-75	Total	Point Mean
ALIMtm	636,566	3,876,105	1,060,643	5,573,314	57.33
ALIMde	1,575,245	661,948	203,772	2,440,965	48.42
OCMtm	1,990,552	2,868,429	525,783	5,384,764	50.57
OCMde	329,568	79,316	20,399	429,283	43.93

7.3.6 Conclusion for ALIM Prototype 3

The reason for the high number of matches is the treatment of tuples as independent in OCM and the expansion of ALIM's results to match this form. Some entities were observed to have more than 20 entries in a single database linked by a primary key. 20 entries for the same entity will result in 190 defined matches in ALIM. This condition is because some or all of the source databases are in 5^{th} or 6^{th} normal form to accommodate data that changes over time and the query that produced the independent tuples output the

Cartesian product of these records. Families or individuals move and the address changes; phone numbers change; first names have aliases; last names change due to marriage and divorce. Some databases have temporary entries in the Social Security Number for newborns that have not been issued an SSN. In rare cases, even gender changes in these data. There are also data entry errors in the data. Also consider that desk checking data found that there is a severe deduplication problem because entities have multiple unique IDs within the same agency.

If one field changes, say phone number, the query would output two independent tuples, one with each phone number. If two fields change, there will be four. Consider that if a woman marries and changes her last name, has three address on record due to moving, two phone numbers and is referred to as both Sue and Susan, she will have 24 independent tuples and there is no guarantee that all of these will match under a matching algorithm that treats data tuples independently. Using a graph-based compound record approach ensures that these data are stored and processed as they really are and does not assume that they will match within the RL system. It also increases the power of the RL process. Treating these data as independent is both less efficient and less effective.

The ALIM model can be extended to include Record relations. One such relation can allow human experts to include keep-together and keep-apart rules for records that known to match or not match for a given entity. Another is social network data that identify related entities within the system. This can increase accuracy by denying matches of family members. Twins are a problem with RL because most of their data matches. If two entities are identified as familial related, they cannot be the same. If a social network exists in one or more agencies, RL can be used to extend these to an enterprise-wide

social network, allowing Health and Social Service professionals to explore familial trends over time. ALIM can be extended to a 6th normal form data graph by including a timestamp on data entries, allowing temporal sorting and analysis of data. Figure 62 shows sample record relations.

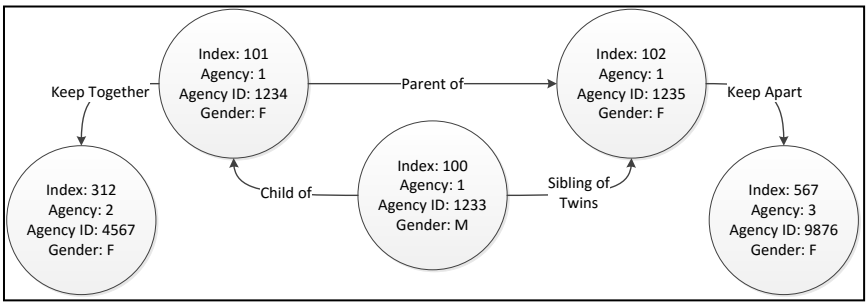


Figure 62: Record relations

RL is a useful but challenging topic for big data science theoretically and practically. Having a reasonable definition and practice grounded implementation helps to push the envelope of big data science towards trustworthy and dependable services. ALIM is capable of performing far more comprehensive record linkage matches than traditional deterministic static phonetic methods using relational or Non-SQL databases. In preliminary tests using a real dataset with a known ground-truth, we are able to outperform a system currently in use, producing more true positive matches. We are currently writing papers to present ALIM and PSH, where the methods are described in detail so that the experiments should be reproducible and the methods implementable by domain experts and researchers. One of the major problems in RL is the lack of real data to test methods. Real Health and Social Services data is protected in the U.S. by federal law and is extremely hard to obtain.

It has been suggested that processing big data in memory may not be possible, which is the case for some problems. The experiments mentioned above were processed on a dual

quad-core Xeon processor server with 48GB of RAM and consumed less than 12GB for approximately 8.5 million tuples. We checked the price of readily available hardware and found that a server with four 12-core Opteron processors and 256GB of RAM can be built for under $25,000 U.S., which may be capable of processing approximately 170 million records if the memory usage scales linearly (ignoring and compression that occurs by eliminating redundant data). Our experiment only included three of eleven datasets from the health department. All eleven datasets contain over 50 million records for approximately 1.5 million clients.

PART IV

CONCLUDING REMARKS

CHAPTER 8

CONCLUSION

The combination of ALIM and PSH methodologies create a powerful RL tool that handles many known issues (alias terms, use of DL and transitive closure of RL) and issues revealed by our research (temporal data, redundant data and reliable DL optimization). Our model also provides a facility for using domain expert guidance using "keep together" and "keep apart" rules, family member data to boost RL confidence and quality, and the potential for social network analysis. Though the storage mechanism is present in the ALIM model, these functions were not implemented because the data to populate the record relation links were not available.

We are confident that our current results are higher quality than OCM because our point and threshold system is stricter than OCM. We were concerned that data shared by parents and children with the same first names and twin siblings could be erroneously matched. This is a common problem with RL systems on big data sets. The upper limits for the point-and-threshold system for exact matches for ALIM and OCM are shown below in Table 59.

Table 59: Points for exact field matches for ALIM and OCM

	First Name	Last Name	Address	Phone	Gender	Birthdate	SSN
ALIM	8	8	8	8	2	11	22
OCM	10	10	10	10	2	10	20

Note that given the 35 point threshold, the sum of the first five fields is a match for OCM (42 points) and not a match for ALIM (34 points). A father and son with the same first name will match in OCM. In creating our match system, we really tried to use the same kind of logic that a human would use to match records. OCM also awards 3 bonus points for a Gender, Birthdate and SSN match. The max point value for ALIM is 67 points and the max point value for OCM is 75.

CHAPTER 9

CONTRIBUTIONS TO THE RL DOMAIN

We offer that our research has made the following contributions to the RL domain:

1. Handles the temporal data problem (aggregate records)
2. Removes data and processing redundancies
3. Ensures transitive closure on RL (Iterative RL)
4. Handles synonymous terms (aliases), phonetic terms and minor spelling errors
5. Guarantees that all optimizations are lossless with respect to accuracy
6. Provides a mechanism to allow system users to correct mistakes in RL
7. Provides a mechanism to make use of available contextual data (family members)
8. Provides a mechanism for the future implementation of family linkage (SNA)
9. Handles both RL and deduplication operations, concurrently
10. Performs approximate string matching faster with our new and novel hybrid hash method
11. Creates a better quality comparison vector than is possible with other methodologies

Our ALIM methodology handles all of the aforementioned principals of RL and will generate comparison vectors that better represent each entity's data. ALIM's in-memory data graph consolidates entities' temporal data (1) (may have multiple values for a single field because data can change over time), eliminates redundancy by using a 5^{th} normal

form graph to store data (2). ALIM uses iterative RL to make use of prior search results to ensure a transitive closure on RL (3) ALIM uses PSH for approximate string matching, Soundex for phonetic matching and Alias matching for synonymous first names in addition to exact field matching (4). The PSH optimization produces the same results as DL but much faster (5 and 10). The storage of "keep records together"/"keep records apart" rules at the record level allows users to use domain knowledge to correct mistakes (6), use family data as a context to boost matching confidence (7) record level familial and social relations allow family linkage and Social Network Analysis (SNA) (8). The system performs deduplication and matching, concurrently (9). ALIM has been experimentally proven to find more matches than a common system currently used by a local health department.

It should also be noted that the work in this document is not just a conjecture, hypothesis or theory of how to perform RL. It is experimentally proven to produce better results than a system currently in use by a local urban health department on big data from real databases that store actual clients' data.

CHAPTER 10

FUTURE WORK

ALIM Prototype 4 will enable the use of the family relation and keep together/keep apart rule record links. We will also obtain data and run experiments with family context, Family Linkage and Social Network Analysis features fully enabled. The following mechanisms are available but not implemented in the current model:

1. Provides a mechanism to allow system users to correct mistakes in RL
2. Provides a mechanism to make use of available contextual data (family members)
3. Provides a mechanism for the future implementation of family linkage (SNA)

ALIM Prototype 5 will be multi-threaded and distributed to take full advantage of multi-core computers and cluster computing to handle bigger data problems, faster. Since the system will use multiple computers with data redundancy, we will have to ensure reliable updates—meaning that updates are consistently applied across all computers. We will also have to ensure fault tolerance. If some part of the cluster fails, any job in progress should complete, correctly.

REFERENCES

[Ananthakrishna Chaudhuri and Ganti 2002] R. Ananthakrishna, Chaudhuri, R., and Ganti, V.: Eliminating fuzzy duplicates in data warehouses. In Proceedings of the 28th International Conference on Very Large Databases (VLDB-2002), Hong Kong, China, 2002.

[Apache Commons 2102] Apache Commons Codec Software Webite. http://commons.apache.org/codec/. Retrieved 2012-11-03

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

[Arasu, Ganti and Kaushik 2006] Arasu, A., Ganti, V., Kaushik, V.; "Efficient Exact Set Similarity Joins". VLDB pp. 918-929 2006

[Avotaynu 2012] The Avotaynu Journal of Jewish Genealogy Website. http://www.avotaynu.com/soundex.htm. Retrieved 2012-11-03

[Baxter, Christen and Churches 2003] 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)

[Benjelloun, Garcia-Molina, Menestrina, Su, Whang and Widom 2008] Benjelloun, O., Garcia-Molina, H., Menestrina, D., Su, Q., Whang, S., Widom, J.: Swoosh : a generic, approach to entity resiolution, The VLDB Journal (2008), Volume: 18, Issue: 1, Publisher: Springer, Pages: 255-276

[Bhattacharya and Getoor 2004] Bhattacharya, I., Getoor, L., Iterative record linkage for cleaning and integration, DMKD '04 Proceedings of the 9th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery

[Bilenko and Mooney 2003] Mikhail Bilenko , Raymond J. Mooney, Adaptive duplicate detection using learnable string similarity measures, Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, August 24-27, 2003, Washington, D.C.

[Bleiholder, Khuller, Naumann, Raschid and Wu 2006] J. Bleiholder, S. Khuller, F. Naumann, L. Raschid and Y. Wu: Query planning in the presence of overlapping sources, EDBT 2006

[Bleiholder, Naumann 2008] Bleiholder, J., Naumann, F.: Data Fusion, ACM Computing Surveys Vol. 41 No. 1 Article 1 2008

[Boitsov 2001] Boitsov, L.: "Using Signature Hashing for Approximate String Matching", Prikladnaya Matematika I Informatiks, No. 8, pp. 135-153, 2001

[Boytsov 2011] Boytsov, L; Indexing Methods for Approximate Dictionary Searching: Comparative Analysis; ACM Journal of Experimental Algorithmics, Vol. 16, No. 1, Article 1, May 2011

[Boytsov 2012] Boytsov, L; Super-Linear Indices for Approximate Dictionary Searching; SISAP 2012, LNCS 7404, pp. 162-176, 2102

[Campbell 2005] Campbell, K.: "Rule Your Data With The Link King", SUGI30 Proceedings, 2005

[Campbell and Deck 2008] 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)

[Caversham 2012] Caversham Project Website. http://caversham.otago.ac.nz/. Retrieved 2012-11-03

[Chang and Lin 2001] Chang, C.-C. & Lin, C.-J. (2001), LIBSVM: A library for support vector machines, manual. Department of Computer Science, National Taiwan University. Software available at: http://www.csie.ntu.edu.tw/~cjlin/libsvm

[Chaudhuri, Ganti and Kaushik 2006] 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

[Christen 2007] Christen, P. (2007), A two-step classification approach to unsupervised record linkage, in 'Australasian Data Mining Conference' (AusDM'07), Gold Coast, Conferences in Research and Practice in Information Technology (CRPIT), vol. 70.

[Christen 2008a] Christen, P.; Febrl - An Open Source Data Cleaning, Deduplication and Record Linkage System with a Graphical User Interface, Proc. of the ACM SIGKDD, Las Vegas, August 2008

[Christen 2008b] Christen, P.; Febrl - A Freely Available Record Linkage System with a Graphical User Interface, Proc, of the Australasian Workshop on Health Data and Knowledge Management (HDKM), Wollongong, January 2008

[Christen and Belacic 2005] Christen, P. & Belacic, D.; Automated probabilistic address standardisation and verification, in 'Australasian Data Mining Conference' (AusDM 2005), Sydney.

[Christen and Churches 2003] Christen, P., and Churches, T. (2003), "Febrl - Freely extensible biomedical record linkage" (user manual)

[Churches, Christen, Lim and Zhu 2002] Churches, T., Christen, P., Lim, K. & Zhu, J.X.; Preparation of name and address data for record linkage using hidden Markov models, BioMed Central Medical Informatics and Decision Making, vol. 2, no. 9. 2002

[Cohen and Ravikumar 2003] 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.

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

[Cusmai, Aiello, Scannapieco, and Catarci 2008] Cusmai, Francesco; Aiello, Carola; Scannapieco, Monica; Catarci, Tiziana. "EARL: an Evolutionary Algorithm for Record Linkage". VLDB '08, August 24-30, 2008, Auckland, New Zealand

[Daechsel 1979a] Daechsel Werner F.O, Will Canada Adopt the Universal Health Number?, Canadian Hospital, 1972 March, pp. 21-22 .

[Daechsel 1979b] Daechsel, Werner F.O.: Problems Arising from Universal Numbering and Record Linkage Systems, Canadian Hospital, 1972 March, pp. 23-24.

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

[Date, Darwen and Lorentzos 2003] Date, C., Darwen, H., Lorentzos, N.: (January 2003). Temporal Data and the Relational Model: A Detailed Investigation into the Application of Interval and Relation Theory to the Problem of Temporal Database Management. Oxford: Elsevier LTD

[Dempster, Laird and Rubin 1977] Dempster, A. P. , Laird, N. M., and Rubin, D.B. (1977), "Maximum Likelihood From Incomplete Data Via the EM Algorithm Expectation-maximization (EM) algorithm is a method for maximum likelihood estimation in problems involving incomplete data," Journal of the Royal Statistical Society, 39, 1-38

[Dice 1945] Dice, Lee R. (1945). "Measures of the Amount of Ecologic Association Between Species". Ecology 26 (3): 297–302. doi:10.2307/1932409. JSTOR 1932409

[Dong, Halevy and Madhavan 2005] Dong, X.,Halevy, A.Y.,Madhavan, J.: Reference reconciliation in complex information spaces. In: Proceedings of ACM SIGMOD (2005)

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

[Eckerson 2002] Eckerson, Wayne W. (2002). "Data Quality and the Bottom Line: Achieving Business Success through a Commitment to High Quality Data". Data Warehousing Institute report 2002

[Elfeky, Verykios, and Elmagarmid 2002] M. G. Elfeky, V. S. Verykios, and A. K. Elmagarmid. TAILOR: A record linkage toolbox. In Proc. of the ICDE, 2002.

[Elmagarmid, Ipeirotis and Verykios 2007] Elmagarmid, Ahmed; Panagiotis G. Ipeirotis, Vassilios Verykios (2007). “Duplicate Record Detection: A Survey.” IEEE Transactions on Knowledge and Data Engineering 19 (1): pp. 1–16.

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

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

[Giegerich and Kurtz 1999] Giegerich, R., Kurtz, S., Stoye, J.: “Efficient Implimentation of Lazy Suffix Trees”, In Proc. 3rd Workshop on Algorithm Engineering (WAE’99), LNCS 1668, pp. 30-42, 1999.

[Goiser and Christen 2006] Goiser K. & Christen, P. (2006), Towards automated record linkage, in ‘Australasian Data Mining Conference’ (AusDM’06), Sydney, Conferences in Research and Practice in Information Technology (CRPIT), vol. 61, pp. 23–31.

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

[Gu and Baxter 2004].“Adaptive Filtering for Efficient Record Linkage”, SIAM 2004

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

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

[Hernandez and Stolfo 1998] 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)

[Hong, Yang, Kang and Lee 2008] Hong, Yoojin; Yang, Tao; Kang, Jaewoo; Lee, Dongwon (2008). “Record Linkage as DNA Sequence Alignment Problem”. VLDB ‘08, August 24-30, 2008, Auckland, New Zealand

[Jaccard 1901] Jaccard, P. (1901) Distribution de la flore alpine dans le bassin des Dranses et dans quelques régions voisines. Bulletin de la Société Vaudoise des Sciences Naturelles 37, 241-272.

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

[Jupin and Shi 2011] Aggregate-Link-Match and Original CARES Matching Algorithms: Progress Report for Development and Experiments 12/22/2011

[Jupin, Shi and Obradovic 2012] Jupin, J., Shi, J. and Obradovic, Z. (2012) "Understanding Cloud Data Using Approximate String Matching and Edit Distance," Proc. 2nd International Workshop on Sustainable HPC Cloud at the IEEE Supercomputing Conference 2012, Salt Lake City, Utah, Nov. 2012.

[Jupin Prelim 1 2011] Jupin, J.; Preliminary Examination 1; 2011-09-12

[Jurczyk, Lu, Xiong, Cragan and Correa 2008a] P. Jurczyk, J. Lu, L. Xiong, J. Cragan, A. Correa: “FRIL: A Tool for Comparative Record Linkage”, AMIA Annu Symp Proc. 2008; 2008: 440–444. Published online 2008

[Jurczyk, Lu, Xiong, Cragan and Correa 2008b] P. Jurczyk, J. Lu, L. Xiong, J. Cragan, A. Correa: “Fine-Grained Record Integration and Linkage Tool”, Birth Defects Research (Part A) 82:822-829 (2008), Wiley-Liss, Inc.

[Kardes, Konidena, Agrawal, Huff and Sun 2013] Kardes, H., Konidena, D., Agrawal, S., Micah Huff, M., Sun, A.: Graph-based Approaches for Organization Entity Resolution in MapReduce, Proceedings of the TextGraphs-8 Workshop, pages 70-80, Seattle, Wash., USA, 2103

[Knuth 1973] Knuth, Donald E. (1973). The Art of Computer Programming: Volume 3, Sorting and Searching, 1st ed.. Addison-Wesley.

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

[Kuch 1972] T. D. C. Kuch, Record linkage and the universal identifier, ACM SIGCAS Computers and Society, Volume 3 Issue 4, December 1972, ACM New York, NY, USA

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

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

[Li, Dong, Maurino and Srivastava] Pei Li, Xin Luna Dong, Andrea Maurino, and Divesh Srivastava. Linking Temporal Records. In VLDB 2011.

[Ling, Tompa and Kameda 1981] T.-W. Ling, F.W. Tompa, and T. Kameda, "An Improved Third Normal Form for Relational Databases", ACM Transactions on Database Systems, 6(2), June 1981, 329-346.

[Link Plus Guide 2007] The Link Plus Users Guide Version 2.0 (2007)

[LinkageWiz 2009] Record linkage software. Version 5.0. LinkageWiz Inc. Available from http://www.linkagewiz.net/. Retrieved 2009-04-20

[Lunen 2012] Lunen, A.: Way to go: A Graph Mining Based Approach to Record Linkage for Sparse Data, International Journal of Arts & Sciences 2012

[McCallum, Nigam and Ungar 2000] 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)

[Mexican Nicknames] Mexican nicknames: http://www.mexicofile.com/hispanicapodosnicknamesprofessionaffectionandphysicalcharacteristics.htm

[Microsoft 2012] “SOUNDEX (Transact-SQL)” at Microsoft MSDN Documentation Website. http://msdn.microsoft.com. Retrieved 2012-11-03.

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

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

[MySQL 2012] "MySQL :: MySQL 5.5 Reference Manual :: 12.5 String Functions - SOUNDEX" at MySQL Documentation Website. http://dev.mysql.com. Retrieved 2012-11-03

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

[NARA 2007] "The Soundex Indexing System". National Archives and Records Administration (NARA). http://www.archives.gov/research/census/soundex.html. Retrieved 2007-05-30

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

[Navarro, Baeza-yates, Sutinen, and Tarhio 2000] Navarro, G., Baeza-yates, R., Sutinen, E., and Tarhio, J.: "Indexing Methods for Approximate String Matching", IEEE Data Engineering Bulletin 2000

[NCHS 2010] Linkages between Survey Data from the National Center for Health Statistics and Medicare Program Data from the Centers for Medicare and Medicaid Services;
http://www.cdc.gov/nchs/data/datalinkage/cms_medicare_methods_report_final.pdf; 2010

[Newcombe 1988] Newcombe, H. B. (1988). "Handbook of Record Linkage Methods for Health and Statistical Studies. Administration and Business." Oxford, UK: Oxford University Press.

[Oracle 2012] "Soundex" at Oracle Documentation Website. http://docs.oracle.com. Retrieved 2012-11-03.

[Li, Dong, Maurino and Srivastava 2011] Pei Li, Xin Luna Dong, Andrea Maurino, and Divesh Srivastava. Linking Temporal Records. In VLDB 2011.

[Postel 1969] Hans Joachim Postel: Die Kölner Phonetik. Ein Verfahren zur Identifizierung von Personennamen auf der Grundlage der Gestaltanalyse. in: IBM-Nachrichten, 19. Jahrgang, 1969, S. 925-931.

[Philips 1990] Hanging on the Metaphone, Lawrence Philips. Computer Language, Vol. 7, No. 12 (December), 1990.

[PostgreSQL 2012] "PostgreSQL: Documentaiton: 9.1: fuzzystrmatch" at PostgreSQL Documentation Website. http://postgresql.com. Retrieved 2012-11-03.

[Ravikumar and Cohen 2004] Ravikumar, P., Cohen, W., A hierarchical graphical model for record linkage, UAI '04 Proceedings of the 20th conference on Uncertainty in artificial intelligence.

[Rezig, Dragut, Ouzzani and Elmagarmid] E. Rezig, E. Dragut, M. Ouzzani, and A. Elmagarmid. Query-Time Record Linkage and Fusion over Web Databases. 31st International Conference on Data Engineering (ICDE'15)

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

[Shen, DeRose, Vu, Doan and Rama-Krishnan 2007] W. Shen, P. DeRose, L. Vu, A. Doan, R. Rama-Krishnan Source-aware Entity Matching: A Compositional Approach. pp.196-205, ICDE 2007.

[Shu and Meng 2009] L. Shu, B. Long, W. Meng. A Latent Topic Model for Complete Entity Resolution. 25th IEEE International Conference on Data Engineering (ICDE), pp.880-891, Shanghai, China, March 2009.

[Shu, Meng and Yu 2007] L. Shu, W. Meng, H. He, C. Yu. Querying Capability Modeling and Construction. 8th International Conference on Web Information Systems Engineering (WISE), 2007.

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

[Stanier 1990] Alan Stanier, September 1990, Computers in Genealogy, Vol. 3, No. 7

[Sun and Naughton 2012] Sun, C., Naughton, J., Barman, C.: "Approximate String Membership Checking: A Multiple Filter, Optimization-Based Approach" ICDE, 2012

[Taft 1970] Taft, R. L. (1970), "Name Search Techniques", Albany, New York: New York State Identification and Intelligence System

[Thoburn, Gu and Rawson 2007] K.K. Thoburn, D. Gu and T. Rawson. Link Plus: Probabilistic Record Linkage Software. 2nd Probabilistic Record Linkage Conference Call, 2007.

[Thor and Rahm 2007] A. Thor, E. Rahm. MOMA – A Map-ping-based Object Matching System. CIDR 2007, 247-258.

[Tversky 1977] Tversky, Amos (1977). "Features of Similarity". Psychological Reviews 84 (4): 327–352.

[U.S. Census 1990] U. S. Census Website: http://www.census.gov/

[U.S. Census 2000] U. S. Census Website: http://www.census.gov/

[U.S. Patent 1,261,167 1918] U.S. Patent No. 1,261,167 Issued on April 2, 1918

[Ukkonen 1985] Ukkonen, E.: “Finding approximate patterns in strings”, Journal of Algorithms, 6:132-137, 1985

[Ukkonen 1993] Ukkonen, E.; Approximate String Matching Over Suffix Trees. CPM 1993. LNCS, Vol. 684, pp. 228-242, Springer, Heidelberg 1993

[Wang and Madnick 1989] Y. Wang and S. Madnick. The Inter-Database Instance Identification Problem in Integrating Autonomous Systems. pp.46-55, ICDE 1989.

[Wang, Feng and Li 2010] Wang, J; Feng, J; Li, G. “TrieJoin: Efficient Trie-based String Similarity Joins with Edit-Distance Constraints”. Proceedings of the VLDB Endowment, Vol. 3, No. 1 (2010)

[Wenfei, Fan, Tian, Dong 2015] Fan, W., Fan, Z., Tian, C. , Dong, X.: Keys for Graphs, VLDB Vol. 8 No. 12 2015

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

[Winkler 1990] 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)

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

BIBLIOGRAPHY

Ee-Peng Lim (1998), “Entity Identification,” Encyclopedia of Computer Science and Technology, Vol. 39, Marcel Decker Inc., New York, pp. 1-14.

Statistics New Zealand (2006). “Data Integration Manual”

Encyclopedia of Public Health (). “Record Linkage”

Math, Myth and Magic (2004). Introduction to Name Search and Matching.

Data Matching - Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection (2012) by Peter Christen. Springer Data Centric Systems and Applications Series

PUBLISHED WORK

Jones, P.; Schwartz, D.; Schwartz, I; Obradovic, Z; Jupin, J.; "Risk Classification And Juvenile Dispositions: What Is The State Of The Art"; Temple Law Review, Volume 79:2, Temple University.

J. Mennis, P. Harris, Z. Obradovic, A. Izenmann, H. Grunwald, Y. Qin, J. Jupin, and B. Lockwood; “Spatial Clustering of Juvenile Delinquency and Recidivism in Philadelphia, Pennsylvania”; Presented at the 103rd Annual Meeting of the Association of American Geographers, April 17-21, 2007, San Francisco, CA.

P. Harris, J. Mennis, Z. Obradovic, A. Izenmann, H. Grunwald, Y. Qin, J. Jupin, and B. Lockwood; “Exploratory Spatial Analysis of Juvenile Delinquency and Recidivism”; Presented at the Ninth Crime Mapping Conference, March 28-31, Pittsburgh, PA.

Mennis, J. , Harris, P. W., Grunwald, H. , Lockwood, B. , Izenman, A. , Obradovic, Z. and Jupin, J. , 2008-11-11 "Characterizing Neighborhoods for Modeling Environmental Effects on Individual Behavioral Outcomes" Paper presented at the annual meeting of the ASC Annual Meeting, St. Louis Adam's Mark, St. Louis, Missouri

Ayuyev, V., Jupin, J., Harris, P. and Obradovic, Z. (2009) "Dynamic Clustering Based Estimation of Missing Values in Mixed Type Data," Proc. 11th Int’l Conf. on Data Warehousing and Knowledge Discovery, Linz, Austria, Sept. 2009, pp. 366-377.

Harris, P., Mennis, J., Obradovic, Z., Izenman, A., Grunwald, H., Lockwood, B., Jupin J., and Chisholm, L.,(2009) "Investigating the Simultaneous Effects of Individual, Neighborhood, and Program Effects on Juvenile Recidivism Using GIS and Spatial Statistics." Research report, the U.S. Department of Justice, National Institute of Justice. (2009)

Izenman, A., Harris, P., Mennis, J., Jupin, J., Obradovic, Z. " Local Spatial Biclustering and Prediction of Urban Juvenile Delinquency and Recidivism," Statistical Analysis and Data Mining Journal, vol. 4, no. 3, pp. 259-275. (2011)

Jupin, J., Shi, J. and Obradovic, Z. (2012) "Understanding Cloud Data Using Approximate String Matching and Edit Distance," Proc. 2nd International Workshop on Sustainable HPC Cloud at the IEEE Supercomputing Conference 2012, Salt Lake City, Utah, Nov. 2012.

Jupin, J., Shi, J.: “Identity Tracking in Big Data – Preliminary Research Using In-Memory Data Graph Models for Record Linkage and Probabilistic Signature Hashing for Approximate String Matching in Big Health and Human Services Databases”, BigDataScience '14 - The Third ASE International Conference on Big Data Science and Computing, August 04 - 07 2014, Beijing, China

Jupin, J., Shi, Y.: PSH: A Probabilistic Signature Hash Method with Hash Neighborhood Candidate Generation for Fast Edit-Distance String Comparison on Big Data, Big Data, Washington D.C. December 2016 – In Review

Jupin, J., Shi, Y.: ALIM: A Foundation for a Graph-Based Data Model to Facilitate Record Linkage on Temporal Data, Big Data, Washington D.C. December 2016 – In Review