A DICTIONARY-BASED APPROACH TO FAST AND ACCURATE NAME MATCHING IN LARGE LAW ENFORCEMENT DATABASES

Olcay Kursun*, Anna Koufakou†, Bing Chen†, Michael Georgiopoulos†, Kenneth M. Reynolds‡, Ron Eaglin*

* Department of Engineering Technology
† School of Electrical Engineering and Computer Science
‡ Department of Criminal Justice and Legal Studies
University of Central Florida, Orlando, FL 32816
{okursun,akoufako,bchen,michaelg,kreynold,reaglin}@mail.ucf.edu

Abstract. In the presence of dirty data, a search for specific information by a standard query (e.g., search for a name that is misspelled or mistyped) does not return all needed information. This is an issue of grave importance in homeland security, criminology, medical applications, GIS (geographic information systems) and so on. Different techniques, such as soundex, phonix, n-grams, edit-distance, have been used to improve the matching rate in these name-matching applications. There is a pressing need for name matching approaches that provide high levels of accuracy, while at the same time maintaining the computational complexity of achieving this goal reasonably low. In this paper, we present ANSWER, a name matching approach that utilizes a prefix-tree of available names in the database. Creating and searching the name dictionary tree is fast and accurate and, thus, ANSWER is superior to other techniques of retrieving fuzzy name matches in large databases.

1 Introduction

With the advances in computer technologies, large amounts of data are stored in data warehouses (centralized or distributed) that need to be efficiently searched and analyzed. With the increased number of records that organizations keep the chances of having “dirty data” within the databases (due to aliases, misspelled entries, etc.) increases as well [1, 2]. Prior to the implementation of any algorithm to analyze the data, the issue of determining the correct matches in datasets with low data integrity must be resolved.

The problem of identifying the correct individual is indeed of great importance in the law enforcement and crime analysis arenas. For example, when detectives or crime analysts query for individuals associated with prior burglary reports, they need to be able to examine all the records related to these individuals, otherwise they might miss important clues and information that could lead to solving these cases. As mentioned earlier, missing names (and thus records) becomes a problem mainly due to common typing and misspelling errors. However, in the case of crime related applications, this problem becomes even bigger due to other reasons, most important of which being that criminals try to modify their name and other information in order to deceive the law enforcement personnel and thus evade punishment. The other reason is that for a large number of cases, the name information might come from witnesses, informants, etc., and therefore this information (for example the spelling of a name) is not as reliable as when identification documents are produced. This is also true in the field of counterterrorism, where a lot of information comes from sources that might be unreliable, but which still needs to be checked nevertheless. It is evident then that it is imperative to have an efficient and accurate name matching technique that will guarantee to return all positive matches of a given name. On the other hand, the returned matches should not have too many false-positives, as the person who is investigating a crime is likely to be overwhelmed by unrelated information, which will only delay the solution of a case.

In this paper, we focus on the problem of searching proper nouns (first and last names) within a database. The application of interest to us is in law enforcement; however, there are many other application domains where availability of accurate and efficient name search tools in large databases is imperative, such as in medical, commercial, or governmental fields [3, 4].

There are two main reasons for the necessity of techniques that return fuzzy matches to name queries: (1) the user does not know the correct spelling of a name; (2) names are already entered within the database with errors because of typing errors, misreported names, etc [5]. For example, record linkage, defined as finding duplicate records in a file or matching different records in different files [6, 7], is a valuable application where efficient name matching techniques must be utilized.

2 Existing Methods

The main idea behind all name matching techniques is comparing two or more strings in order to decide if they both represent the same string. The main string comparators found in the literature can be divided in phonetic and spelling based. *Soundex* [8] is used to represent words by phonetic patterns. Soundex achieves this goal by encoding a name as the first letter of the name, followed by a three-digit number. These numbers correspond to a numerical encoding of the next three letters (excluding vowels and consonants h, y, and w) of the name [11]. The number code is such that spelled names that are pronounced similar will have the same soundex code, e.g., “Allan” and “Allen” are both coded as

"A450". Although soundex is very successful and simple, it often misses legitimate matches, and at the same time, detects false matches. For instance, "Christie" (C623) and "Kristie" (K623) are pronounced similarly, but have different soundex encodings, while "Kristie" and "Kirkwood" share the same soundex code but are entirely different names.

On the contrary, spelling string comparators check the spelling differences between strings instead of phonetic encodings. One of the well-known methods that is used to compare strings is measuring their "edit distance", defined by Levenshtein [9]. This can be viewed as the minimum number of characters that need to be inserted into, deleted from, and/or substituted in one string to create the other (e.g., the edit distance of "Mich*a*el" and "Mi*t*chel*l*" is three). Edit-distance approaches can be extended in a variety of ways, such as taking advantage of phonetic similarity of substituted characters (or proximity of the corresponding keys on the keyboard) or checking for transposition of neighboring characters as another kind of common typographical error [10] (e.g., "Baldwin" vs. "Badlwin"). The name-by-name comparison by edit distance methods throughout the entire database renders the desired accuracy, at the expense of exhibiting high complexity and lack of scalability.

In this paper, we propose a string-matching algorithm, named ANSWER (Approximate Name Search With ERrors), that is fast, accurate, scalable to large databases, and exhibiting low variability in query return times (i.e., robust). This string comparator is developed to establish the similarity between different attributes, such as first and last names. In its application to name matching, ANSWER is shown to be even faster than phonetic-based methods in searching large databases. It is also shown that ANSWER's accuracy is close to those of full exhaustive searches by spelling-based comparators. Similar work to ours has been described by Wang et. al. [4], which focuses in identifying deceptive criminal identities, i.e. in matching different records that correspond to the same individual mainly due to false information provided by these individuals.

3 The Operational Environment -- FINDER

One of the major advantages of our research is that we have a working test-bed to experiment with (FINDER – the Florida Integrated Network for Data Exchange and Retrieval). FINDER (see Fig. 1) has been a highly successful project in the state of Florida that has addressed effectively the security and privacy issues that relate to information sharing between above 120 law enforcement agencies. It is operated as a partnership between the University of Central Florida and the law-enforcement agencies in Florida sharing data – referred to as the Law Enforcement Data Sharing Consortium. Detailed information about the organization of the data sharing consortium and the FINDER software is available at http://finder.ucf.edu.

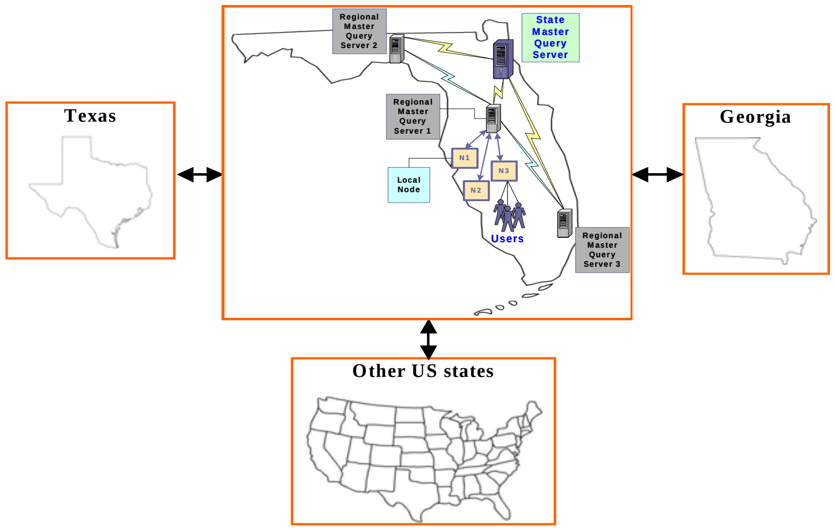
Regional Master Query Server 2
State Master Query Server
Regional Master Query Server 1
Local Node
N1
N2
N3
Users
Regional Master Query Server 3
Texas
Georgia
Other US states

Fig. 1. The general overview of the FINDER network in Florida and expanded to other states

Part of the constraints of the FINDER system and also most law enforcement records management systems is that once the data has entered into the system it must remain intact in its current form. This includes data that have been erroneously entered, and consequently they contain misspellings. This problem was identified by the FINDER team and has also been substantiated in the literature [1, 12, 13, 14]. A simple illustration related to name matching, utilizing dirty data available in the FINDER system, is shown in Table 1, which emphasizes both the level of data integrity and the challenges of using standard SQL queries to retrieve records from a law enforcement database (also known as merge/purge problems [14]). In Table 1, we are depicting the results of an SQL query on “Joey Sleischman”. An SQL query will miss all the records but the first one. The other records could be discovered only if we were to apply an edit distance algorithm on all the existing records in the database, an unsuitable approach though, due to its high computational complexity, especially in large databases. In particular, the rest of the records (besides the exact match), shown in Table 1 were identified by comparing the queried record (“Joey Sleischman”) against all records in the database (by applying the edit distance approach). The Last Name, First Name, DOB (Date of Birth), and Sex were used as parameters in this search. In order to detect the matching records, we assigned weights to the fields: Last Name (40%), First Name (20%), DOB (30%), and Sex (10%). We used the edit distance algorithm [9] for determining the degree of match between fields.

Table 1. Example of the Data Integrity Issues within the FINDER data.

Last Name	First Name	DOB	Sex	Match
INPUT QUERY:				
SLEISCHMAN	*JOEY*	*1/21/1988*	*M*	*≥ 85%*
MATCHING RECORDS:				
SLEISCHMAN	JOEY	1/21/1988	M	100%
SLEICHMAN	JOEY	7/21/1988	M	91%
SLEISCHMANN	JOSEPH	1/21/1988	M	88%
SLEISCHMANN	JOSPEH	1/21/1988	M	88%
SLEISHMAN	JOEY		M	87%
SLEISCHMANN	JOEY		M	87%
SLEISHCHMANN	JOSEPH	1/21/1988	M	86%
SLESHMAN	JOEY		M	85%

As it can be seen in Table 1, the edit distance algorithm provides an excellent level of matching, but the algorithm requires a full table scan (checking all records in the database). This level of computational complexity makes it unsuitable as a technique for providing name matching in applications, such as FINDER, where the number of records is high and consistently increasing. In the next sections, we are discussing in detail a name matching approach that alleviates this computational complexity.

4 The PREFIX Algorithm

In order to reduce the time complexity of the full-search of partially matching names in the database (of crucial importance in homeland security or medical applications), we propose a method that constructs a structured dictionary (or a tree) of prefixes corresponding to the existing names in the database (denoted PREFIX). Searching through this structure is a lot more efficient than searching through the entire database.

The algorithm that we propose is dependent on a maximum edit distance value that is practically reasonable. Based on experimental evidence, it has been stated that edit distance up to three errors performs reasonably well [15]. For example, "Michael" and "Miguel" are already at an edit distance of three. Let k represent the maximum number of errors that is tolerated in the name matching process. Using a minimal k value that works well in the application at hand would make the search maximally fast. Setting k to zero would equal to an exact search which is currently available in any query system. Increasing k increases the recall (i.e., it will not miss any true matches), even though this implies a very slow search and an increase in the number of false positives.

PREFIX relies on edit distance calculations. Its innovation though lies on the fact that it is not searching the entire database to find names that match the query entry but accomplishes this goal by building a dictionary of names. One might think that it would not be very efficient to have such a dictionary due to the fact that we would still need to search the whole dictionary, as the spelling error could happen anywhere in the string, such as “Smith” vs. “Rmith”. However, our algorithm can search the dictionary very fast, using a tree-structure, by eliminating the branches of the tree that have already been found to differ from the query string by more than *k*.

There are two key points to our approach: (1) Constructing a tree of prefixes of existing names in the database and searching this structure can be much more efficient than a full scan of all names (e.g., if “Jon” does not match “Paul”, one should not consider if “Jonathan” does); (2) such a prefix-tree is feasible and it will not grow unmanageably big. This is due to the fact that many substrings would hardly ever be encountered in valid names (e.g., a name would not start with a “ZZ”); consequently, this cuts down significantly the number of branches that can possibly exist in the tree. Similar data structures are proposed in the literature [16, 17] but they are not as suitable as ours when it comes to DBMS implementation (see Section 7).

The PREFIX algorithm creates a series of prefix-tables T_1, T_2, T_3…, where T_n will link (index) T_{n+1}. T_n will contain all *n*-symbol-long prefixes of the names in the database. These tables correspond to the levels of the prefix-tree. The reason that we use tables is to facilitate the implementation of this approach in any database system. T_n will have the following fields: current symbol (the n^{th} symbol), previous symbol ($n\text{-}1^{st}$ symbol), next symbol ($n+1^{st}$ symbol), its links (each link will point to the index of an entry in the prefix-table T_{n+1} with this current entry as the prefix, followed by the symbol in the “next symbol” field), and a field called Name that indicates whether or not the prefix itself is already a name (e.g., Jimm is a prefix of Jimmy but it may not be a valid name). Note that in the links field we cannot have more than 26 links because there are only 26 letters in the English alphabet. Also note that the first prefix-table (T_0) will not utilize the previous symbol field.

Suppose that our database contains “John”, “Jon”, “Jonathan”, “Olcay”, “Jim”, “Oclay”, and “Jimmy”. After building the prefix-dictionary shown in Fig. 2, it can be used as many times as needed for subsequent queries. It is very simple to update the dictionary when new records are added (the same procedure explained above, when creating the tables in the first place, is used to add records one by one). Each level *i* in Fig. 2 is a depiction of the prefix-table T_i (for example the third table consists of JOH, JON, OLC, JIM, OCL). The dark-colored nodes in Fig. 2 are the prefixes that are actually valid names as well.

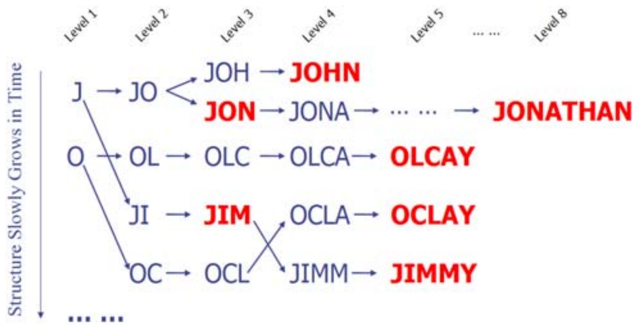

Fig. 2. The tree obtained by processing “John”, “Jon”, “Jonathan”, “Olcay”, “Jim”, “Oclay”, and “Jimmy”.

The advantage of PREFIX is that when we search for approximate name matches, we can eliminate a sub-tree of the above-depicted tree (a sub-tree consists of a node and all of its offspring nodes and branches). Suppose that we search for any similar names with no more than one edit-error to the name “Olkay”. When the algorithm examines level two of the tree, (i.e., the prefix-table T_2), it will find that the node JI is already at a minimum edit distance of two from “Olkay”. Therefore any node that extends from JI-node should not be considered any further. That is, any name that starts with a JI is not going to be within the allowable error margin.

5 The ANSWER Algorithm

To use the PREFIX algorithm for a full name query rather than a single string query (such as a last name or a first name only), we apply the following steps: (1) build prefix-dictionary for the last names; (2) for a given full name query, search the tree for similar last names; (3) apply edit-distance algorithm on the returned records to obtain the ones that also have matching first names. In step 1, we could have built a prefix-tree for first names and in step 2, we could have obtained matching first names by scanning this tree; however, it would not have been as efficient as the stated PREFIX algorithm because first names are, in general, less distinct; consequently, by using first names at the beginning of the search process would have reduced our capability of filtering out irrelevant records.

The PREFIX algorithm offers a very efficient search of names. Nevertheless, it does not provide any direct way of utilizing a given first name along with the last name of a query because it does not use the first name information during the search of the tree. We propose the ANSWER (Approximate Name Search

With ERrors) algorithm for fast and still highly accurate search of full names based on the PREFIX idea. In the process of building the prefix-dictionary, ANSWER takes every full name in the database, and using the PREFIX algorithm, it creates required nodes and links for the last names in the tree. It also augments each node in the tree by 26 bits, each bit representing whether any last name on that branch has an associated first name starting with the corresponding letter in the alphabet. For example, if the last name "Doe" could be found in the database only with the first names "Jon", "John", and "Michael", the corresponding nodes in the "Doe" branch in the tree would be "linked" with "J" and "M", meaning that the last name "Doe" can only have first names starting with "J" or "M".

This architecture allows early (before the edit-distance exceeds the predefined threshold *k*) pruning of tree nodes based on the first letter of the first name of the query. For example, if the query name was "John Doe", ANSWER would prune, say the F-node, if there were no last names starting with letter "F" associated with a first name that starts with "J", the first letter of "John". Based on our preliminary experiments and what we deduced from the literature [5, 11], it is unlikely that both first name and last name initials are incorrect (e.g., "Zohn Foe" is not an expectable match for "John Doe"). On the other hand, PREFIX would not prune the F-node right away because it does not take into consideration the first name at all, and there could be a last name similar to DOE that starts with "F" (e.g., "Foe"). Thus, PREFIX would scan more branches and take longer than ANSWER. Moreover, even though ANSWER is not an exhaustive search algorithm, it exhibits high hit rate as explained in the following section.

6 Experimental Results

In order to assess the performances of our exhaustive search engine PREFIX and its heuristic version ANSWER, we conducted a number of experiments. After creating the prefix-dictionary tree, we queried all distinct full names available in the FINDER database and measured the *time* taken by PREFIX and ANSWER in terms of the number of columns computed in the calculation of edit-distance calls (how edit-distance computation works was explained in Section 4.2). This way, the effect of factors such as operating system, database server, programming language, are alleviated. Furthermore, we compared PREFIX's and ANSWER's performance with other name matching techniques. In particular, we compared PREFIX and ANSWER with two other methods: Filtering-based soundex approach applied on (1) only last name (SDXLAST); (2) first or last names (SDXFULL). SDXLAST is a simple method that is based on the commonly used soundex schema that returns records with soundex-wise-matching last names, and then applies the edit-distance procedure (just as in our methods, the edit-distance calls terminate the computation once the maximum allowable edit errors k is exceeded) to the last names to eliminate the false positives, and applies the edit-distance procedure once more on the first names of the remaining last names, in order to obtain the final set of matching full names.

It is worth noting though, that the hit rate obtained by using only the soundex-matches for the last names is insufficient due to inherent limitations of the soundex scheme [11]. For example, searching for “Danny Boldwing” using SDXLAST would not return “Danny Bodlwing” because the soundex code for “Boldwing” does not match the soundex code for “Bodlwing”. Therefore, we devised an extension of SDXLAST in order to enhance its hit rate. We called this new method SDXFULL. SDXFULL selects records with soundex-wise-matching last names *or* soundex-wise-matching first names. As a result, if “Danny Boldwing” is the input query, SDXFULL would return not only “Danny Bodlwing” and “Dannie Boldwing” as possible true positives, but it would also return many false positives such as “Donnie Jackson” or “Martin Building”. These false positives will be eliminated (by applying edit distance calculations to all these returned records) as in the SDXLAST method. Thus, it is expected that SDXFULL will have a higher recall (true positives) than SDXLAST but longer run-time since it also returns a larger number of false positives). The low recall rate of soundex is the reason for not comparing our methods with other phonetic-type matching methods, such as *Phonix* [11]. Phonix assigns unique numerals to even smaller groups of consonants than soundex, and it is thus expected to have an even lower recall rate than the already unacceptable recall rate observed in SDXLAST [11].

Our database contains about half a million (414,091 to be exact) records of full names, out of which 249,899 are distinct. In order to evaluate the behavior of these four name matching methods as the number of records in the database increases, we have applied each one of the aforementioned methods (PREFIX, ANSWER, SDXLAST, and SDXFULL) to the database at different sizes. For our experiments, we have chosen 25% (small), 50% (medium), 75% (large), and 100% (x-large) of records as the working set sizes. Note that a different prefix-dictionary is used for different set sizes, as the number of records in the database expands from the small to x-large sizes. We used the PREFIX algorithm as the baseline for our algorithm comparisons, since it performs an exhaustive search. For our experiments we used a maximum number of allowable edit-distance of 2 (k=2), for both last and first names. Thus, for every query by the exhaustive search, we have selected from the database all the available full names of which neither the last nor the first name deviates by more than two errors from the last and the first names, respectively, of the query. Of course, this does not mean all of these records with an edit distance of two or less refer to the same individual but this was the best that we could use as a baseline for comparisons because these names were at least interesting in respect that they were spelled similarly.

Fig. 3a plots the graph of average run-times for queries of each approach as a function of the database size. Note that in some applications, the hit-rate of the search can be as important as (if not more important than) the search time. Therefore, in order to quantify the miss rate, we have also computed the average hit-rates (the ratio of the true positives identified versus the total number of actual positives) for these methods (Fig. 3b). SDXLAST is the fastest search; however, it has the lowest hit-rate amongst all the algorithms. Furthermore, SDXLAST's hit-rate is unacceptably low for many applications [5, 11]. The ANSWER search is the next fastest for large databases (except for SDXLAST, which has a small hit rate). ANSWER is also significantly more accurate than the SDXFULL search. SDXFULL executes a simple search that fails when there are errors in both the last and the first names (this happens in increasingly more than 15% of the records). For instance, some of the records that are found by ANSWER but missed by SDXFULL are "Samantha Etcheeson" versus "Samatha Etcheenson" or "Yousaf Kodyxr" versus "Youse Rodyxr".

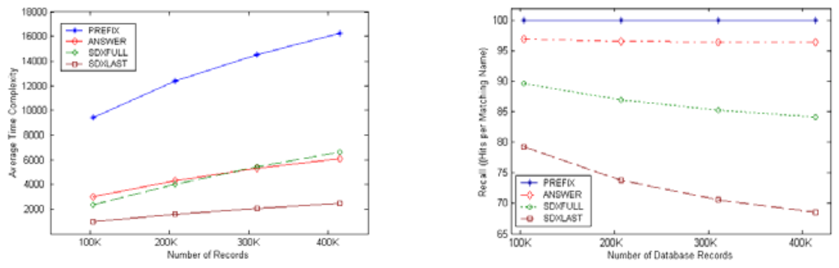

Fig. 3. Scalability versus database size. **(a)** Run-times; **(b)** Recall rates.

7 DBMS Implementation

ANSWER offers a very efficient search of names. Its database system implementation is not as fast, however it still remains to be a crucial tool of querying because it is a full search tool. Other techniques we could use are either partial searches with 60%-70% recall rates (such as soundex or phonix), or very slow (e.g. one pass over all the distinct full names with Levenshtein comparison takes about 10 minutes in database implementation). Soundex takes about half a second to query a name. However, it misses matching records due to its weak heuristicity.

This does not come as a surprise because it is a problem in general that algorithms implemented offline that use data outside the database system can employ efficient structures that reside in memory and a minimal number of database scans, and thus exhibit better performance than the equivalent database implementations. From the performance perspective, data mining algorithms that are implemented with the help of SQL are usually considered inferior to algorithms that process data outside the database systems. One of the important reasons is that offline algorithms employ sophisticated in-memory data structures and can scan the data as many times as needed without much cost due to speed of random access to memory [18]. Our initial experiments with early efforts of DBMS implementation resulted that the run time for ANSWER for *k*=1 is under a second. For k=2, the search time is in order of 5 seconds.

Disk-access is a very costly operation in database systems. Therefore, we will have to reduce the number of database accesses needed for searching the tree. One idea is to use the breadth-first search algorithm. For a query with a searched name of length *n* and *MaxError* tolerable edit distance, the upper bound of the number of database accesses is, therefore, $n + MaxError$.

In order to further reduce database access, when we import the names into prefix tables, we can load the names in partial order so that the similar names are stored together in prefix name tables. Names whose initial letters are "AA" are firstly imported, then those with "AB", "AC", and until "ZZ". This way, when we query names, database server will automatically load and cache the data blocks with similar prefixes, thus we can facilitate I/O accesses by reducing the number of memory blocks to be retrieved.

8 Summary, Conclusions and Future Work

Dirty data is a necessary evil in large databases. Large databases are prevalent in a variety of application fields such as homeland security, medical, among others. In that case, a search for specific information by a standard query fails to return all the relevant records. The existing methods for fuzzy name matching attain variable levels of success related to performance measures, such as speed, accuracy, consistency of query return times (robustness), scalability, storage, and even ease of implementation.

Name searching methods using name-by-name comparisons by edit distance (i.e., the minimum number of single characters that need to be inserted into, deleted from, and/or substituted in one string to get another) throughout the entire database render the desired accuracy, but they exhibit high complexity of run time and thus are non-scalable to large databases. In this paper, we have introduced a method (PREFIX) that is capable of an exhaustive edit-distance search at high speed, at the expense of some additional storage for a prefix-dictionary tree constructed. We have also introduced a simple extension to it, called ANSWER that has run-time complexity comparable to soundex methods, and it maintains robustness and scalability, as well as a comparable level of accuracy compared to an exhaustive edit distance search. ANSWER has been tested, and its advantages have been verified, on real data from a law-enforcement database (FINDER).

References

1. Kim, W. (2002) “On Database Technology for US Homeland Security”, *Journal of Object Technology*, vol. 1(5), pp. 43–49.
2. Taipale, K.A. (2003) “Data Mining & Domestic Security: Connecting the Dots to Make Sense of Data”, *The Columbia Science & Technology Law Review*, vol. 5, pp. 1–83.
3. Bilenko, M., Mooney, R., Cohen, W., Ravikumar, P., Fienberg, S. (2003) “Adaptive name matching in information integration”, *IEEE Intelligent Systems*, vol. 18(5), pp. 16–23.
4. Wang, G., Chen, H., Atabakhsh, H. (2004) “Automatically detecting deceptive criminal identities”, *Communications of the ACM,* March 2004, vol. 47(3), pp. 70–76.
5. Pfeifer, U., Poersch, T., Fuhr, N. (1995) “Searching Proper Names in Databases”, *Proceedings of the Hypertext - Information Retrieval – Multimedia (HIM 95)*, vol. 20, pp. 259–276.
6. Winkler, W.E. (1999) “The state of record linkage and current research problems”, *Proceedings of the Section on Survey Methods of the Statistical Society of Canada*.
7. Monge, A.E. and Elkan, C.P. (1997) “An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records”, Proceedings of the ACM-SIGMOD Workshop on Research Issues on Knowledge Discovery and Data Mining, Tucson, AZ.
8. Newcombe, H.B., Kennedy J.M., Axford S.J., James, A.P. (1959) “Automatic linkage of vital records”, *Science,* vol. 3381, pp. 954–959.
9. Levenshtein, V.L. (1966) “Binary codes capable of correcting deletions, insertions, and reversals”, *Soviet Physics*, Doklady, vol. 10, pp. 707–710.
10. Jaro, M.A. (1976) “UNIMATCH: A Record Linkage System: User’s Manual. Technical Report”, *U.S. Bureau of the Census*, Washington, DC.
11. Zobel, J., Dart, P. (1995) “Finding approximate matches in large lexicons”, *Software-Practice and Experience,* vol. 25(3), pp. 331–345.
12. Wilcox, J. (1997) “Police Agencies Join Forces To Build Data-Sharing Networks: Local, State, and Federal Crimefighters Establish IT Posses”, *Government Computer News*, Sept. 1997.
13. Maxwell, T. (2005) “Information, Data Mining, and National Security: False Positives and Unidentified Negatives”, *Proceedings of the 38th Hawaii International Conference on System Science*.
14. Hernandez, M., and Stolfo, S. (1998) “Real-world Data is Dirty: Data Cleansing and the Merge/purge Problems”, *Data Mining Knowledge Discovery*, vol. 2, pp. 9-37, 1998.
15. Mihov, S., Schulz, K.U. (2004) “Fast Approximate Search in Large Dictionaries”, *Journal of Computational Linguistics*, vol. 30(4), pp. 451–477.
16. Aoe, J., Morimoto, K., Shishibori M., Park, K. (2001) “A Trie Compaction Algorithm for a Large Set of Keys”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 8(3), pp. 476–491.
17. Navarro, G. (2001) “A Guided Tour to Approximate String Matching”, *ACM Computing Surveys,* vol. 33(1), pp.31-88.