ANSWER: *APPROXIMATE NAME SEARCH WITH ERRORS* IN LARGE DATABASES BY A NOVEL APPROACH BASED ON PREFIX-DICTIONARY

OLCAY KURSUN*, ANNA KOUFAKOU†, ABHIJIT WAKCHAURE†, MICHAEL GEORGIOPOULOS†,
KENNETH REYNOLDS‡, RONALD EAGLIN*

* *Department of Engineering Technology*
University of Central Florida, Orlando, FL 32816
{okursun@mail.ucf.edu, reaglin@mail.ucf.edu}

† *Department of Electrical and Computer Engineering*
University of Central Florida, Orlando, FL 32816
{akoufako@ucf.edu, abhijitw@mail.ucf.edu, michaelg@mail.ucf.edu}

‡ *Department of Criminal Justice and Legal Studies*
University of Central Florida, Orlando, FL 32816
kreynold@mail.ucf.edu

Received (Date Month Year)
Revised (Date Month Year)
Accepted (Date Month Year)

The obvious need for using modern computer networking capabilities to enable the effective sharing of information has resulted in data-sharing systems, which store, and manage large amounts of data. These data need to be effectively searched and analyzed. More specifically, 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, as required in homeland security, criminology, and medical applications, amongst others. Different techniques, such as soundex, phonix, n-grams, edit-distance, have been used to improve the matching rate in these name-matching applications. These techniques have demonstrated varying levels of success, but there is a pressing need for name matching approaches that provide high levels of accuracy in matching names, while at the same time maintaining low computational complexity. In this paper, such a technique, called ANSWER, is proposed and its characteristics are discussed. Our results demonstrate that ANSWER possesses high accuracy, as well as high speed and is superior to other techniques of retrieving fuzzy name matches in large databases.

Keywords: alias finding, data mining, data querying, data sharing, dirty data, duplicate elimination, edit distance, fuzzy name matching, record matching, soundex.

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. 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 law enforcement; however, there are many other application domains where availability of accurate and efficient name search tools in large databases is important, such as in medical, commercial, or governmental fields [3, 4]. There are two main reasons for the existence 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.

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 distinguished 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. The number code is such that spelled names that are pronounced similarly 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 string comparator methods measures the string "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. There are some other name-matching approaches that are different than the string comparisons mentioned above. For a comparative review of different name matching algorithms and their efficiency and performance, the reader is also referred to [5, 10, 11], and to the more recent [3]. 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.

2. 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 has been a highly successful project in the state of Florida, and has effectively addressed the security and privacy issues pertinent to information sharing between various law enforcement agencies/users. 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. The FINDER software is a web-based software that uses a web service interface to respond to queries from participating agencies and send query results back to a requesting agency. The software is capable of sending queries to multiple agencies simultaneously and compiling the results into a format that is useful to the requesting law enforcement officer. This approach has allowed participating agencies to meet the political and organizational constraints of sharing data, such as (1) they retain local control of their own data; and (2) they know who has accessed their own data (through log files). Furthermore, FINDER has been a low-cost approach to the participating agencies. As of March 2006, over 120 agencies have executed contracts with the University of Central Florida (UCF) to share data using the FINDER system, and many more have access to the data through guest accounts at neighboring agencies. Detailed information about the organization of the data sharing consortium and the FINDER software is available at http://finder.ucf.edu. Part of the constraints of the FINDER system and also most law enforcement records management systems is that once the data has been 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.

3. 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, name matching applications that produce names of edit distance up to three errors perform reasonably well [12] (e.g., “Michael” and “Miguel” have an edit distance of three). Let *k* represent the maximum number of errors 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 decrease in the precision (i.e., it will return many more false positives).

PREFIX relies on edit distance calculations. Its innovation, though, lies in 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 entire dictionary, as the spelling error could happen anywhere in the string, such as “Smith” vs. “Rmith”. However, our algorithm can search the dictionary very quickly, 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, as 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 large. 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.

The PREFIX algorithm creates a series of prefix-tables T_1, T_2, T_3… T_n, where T_n is linked to (index) T_{n+1}. T_n contains 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-1^{st} symbol), next symbol (n+1^{st} symbol), its links ("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. 1, it can be used as many times as needed by 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. 1 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. 1 are the prefixes that are actually valid names as well. Essentially, we create a list of all possible prefixes in order of appearance (no sorting of the tables is needed). Despite the fact that the prefix tables are not sorted, each one is implicitly indexed by the entries of the preceding table. This is an important advantage, because without this hierarchical structure, the search time would be linear due to the need of checking all the names in the database, one by one. Such a full scan of all the records is a costly database query.

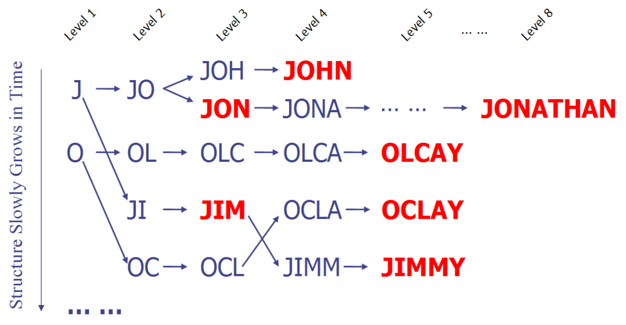

Fig. 1. The tree obtained by insertion of "John", "Jon", "Jonathan", "Olcay", "Jim", "Oclay", and "Jimmy".

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 the 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.

There are techniques similar to PREFIX method [13, 14, 15]. However, our method is simpler in terms of database implementation. Usage of tables to implement the tree (as explained above) and the optimization of the database structure by an indexing scheme make our approach fast and accurate at the expense of additional storage compared to a DAWG [13].

4. 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 data 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 “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, e.g. 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”. PREFIX would not prune the F-node right away because there could be a last name similar to DOE that starts with “F” (e.g., “Foe”). However, 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”). Finally, even though ANSWER is not a comprehensive search algorithm, it exhibits high hit rate and it is faster than PREFIX.

5. Experimental Results

In order to assess the performances of our comprehensive 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 3.2). 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: (1) filtering based soundex approach applied only to the last name (SDXLAST); (2) filtering based soundex approach applied on first or last names (SDXFULL). In our experiments, we also measured the *precision* and the *recall* of the compared methods. Recall (also known as "hit rate") is the ratio of the true positives identified versus the total number of actual positives. Precision is the ratio of the number of true positives identified versus the number of candidate records that are selected by a method. SDXLAST is a simple method that is based on the commonly used soundex schema that returns records with soundex-wise-matching last names. It then applies the edit-distance procedure (as it is the case with our approach, 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. Finally, it 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. 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 has a higher recall (true positives) than SDXLAST but longer run-time since it also returns a larger number of false positives).

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 for different sizes of records. In our experiments, we chose 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 a comprehensive 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 comprehensive 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.

Fig. 2a 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 for these methods (see Fig. 2b). 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). Fig. 2c shows the precision of each method (the ratio of the number of true positives to the number of candidate records). ANSWER's precision is superior to all the other techniques because the last names of all the candidate records satisfy two strict conditions: (1) small edit-distance to query last name, (2) existence of at least one association of a name that starts with the initial letter of the query first name. ANSWER uses these two conditions in a very efficient way by using the prefix-tree that eliminates irrelevant records quickly, while returning candidates that are very likely to be true positives. *Effectiveness* [16] is a frequently used combination value of P (precision) and R (recall). It is computed as $2 \cdot P \cdot R / (P + R)$. Fig. 2d shows the plot of effectiveness as a function of the database size. ANSWER is the most effective solution to name matching, and the curves in Fig. 2d indicate that it will remain to be the most effective, as the database grows even further.

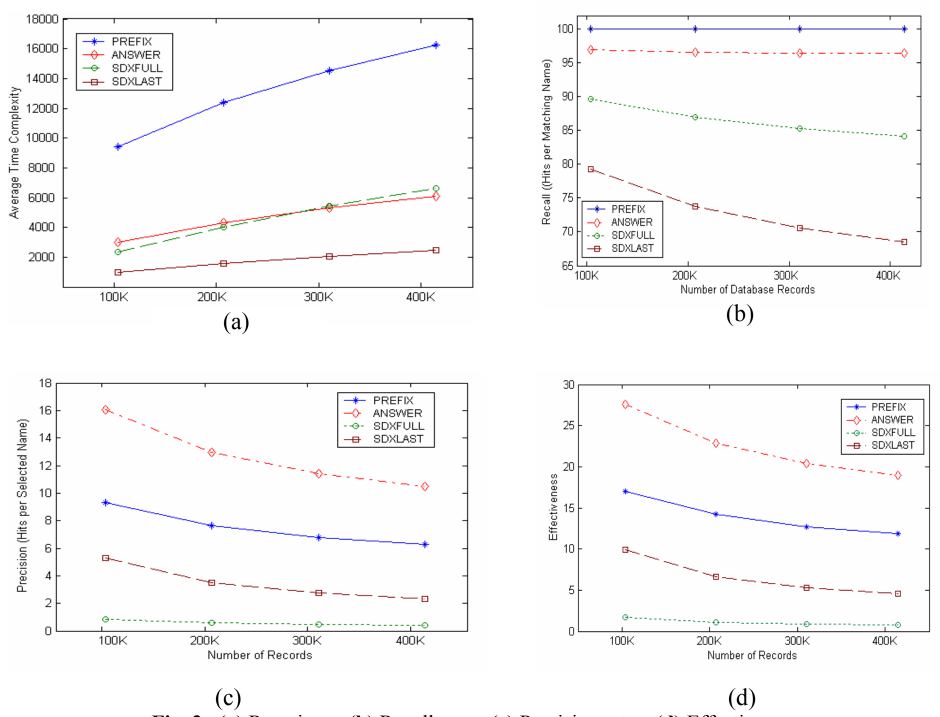

Fig. 2. **(a)** Run-times, **(b)** Recall rates, **(c)** Precision rates, **(d)** Effectiveness of the compared name searching approaches versus the database size.

6. Conclusions

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 the presence of dirty data, a search for specific information by a standard query fails to return all the relevant records. In this paper, we have introduced a method (PREFIX) that is capable of a comprehensive name matching 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. ANSWER has compared very favorably in comparison with a variety of other methods. In particular, the ANSWER name search 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). Furthermore, it should be evident that ANSWER's potential extends beyond the field of law enforcement, to other application fields where a name-searching algorithm with low search time, high accuracy, and good precision is needed.

Acknowledgements

The authors acknowledge helpful discussions with detectives from the Orange County Sheriff's Office. This work was supported, in part by, the Orange County Sheriff's Office, the State of Florida, and National Institute of Justice grant 2004-IJ-CX-K030.

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*, 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. Koufakou, A., Wakchaure, A., Kursun, O., Georgiopoulos, M., Reynolds, K., Eaglin, R. (2005) "Burglary Data Mining - A Three Tiered Approach: Local, State, And Nation-Wide", *The Second GIS Symposium 2005 at Troy State University*, Troy, AL.
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. Mihov, S., Schulz, K.U. (2004) "Fast Approximate Search in Large Dictionaries", *Journal of Computational Linguistics*, vol. 30(4), pp. 451–477.
13. Aoe, J., Morimoro, K., Shishibori, M., and Park, K. (1996) "A Trie Compaction Algorithm for a Large Set of Keys", *IEEE Trans. Knowledge and Data Engineering*, Vol.8 (3), pp.476-491.
14. Navarro, G. (1999) "A guided tour to approximate string matching", Technical Report TR/DCC-99-5, Dept. of Computer Science, Univ. of Chile
15. Gravano, L., Ipeirotis, P., Jagadish, H. V., Koudas, N., Muthukrishnan, S., Srivastava, D. (2001) "Approximate String Joins in a Database (Almost) for Free", *Very Large Databases*, pp. 491-500.
16. van Rijsbergen, C. J. (1979) *Information Retrieval*, 2nd ed, London: Butterworths.