An efficient domain-independent algorithm for detecting approximately duplicate database records

Alvaro E. Monge* **Charles P. Elkan†**
Department of Computer Science and Engineering
University of California, San Diego
La Jolla, California 92093–0114
{amonge,elkan}@cs.ucsd.edu

Abstract

Detecting database records that are approximate duplicates, but not exact duplicates, is an important task. Databases may contain duplicate records concerning the same real-world entity because of data entry errors, because of unstandardized abbreviations, or because of differences in the detailed schemas of records from multiple databases, among other reasons. In this paper, we present an efficient algorithm for recognizing clusters of approximately duplicate records. Three key ideas distinguish the algorithm presented. First, a version of the Smith-Waterman algorithm for computing minimum edit-distance is used as a domain-independent method to recognize pairs of approximately duplicate records. Second, the union/find algorithm is used to keep track of clusters of duplicate records incrementally, as pairwise duplicate relationships are discovered. Third, the algorithm uses a priority queue of cluster subsets to respond adaptively to the size and homogeneity of the clusters discovered as the database is scanned. This typically reduces by over 75% the number of times that the expensive pairwise record matching (Smith-Waterman or other) is applied, without impairing accuracy. Comprehensive experiments on synthetic databases and on a real database of bibliographic records confirm the effectiveness of the new algorithm.

1 Introduction

In this paper we study the problem of detecting records in a database that are duplicates of each other, but not necessarily textually identical. This is a common problem in environments where multiple databases must be combined, or where records contain erroneous and/or missing information.

In general, we are interested in situations where several records may refer to the same real-world entity, while not being syntactically equivalent. A set of records that refer to the same entity can be interpreted in two ways. One way is to view one of the records as correct and the other records as duplicates containing erroneous information. The task then is to cleanse the database of the duplicate records [SSU95, HS95]. Another interpretation is to consider each matching record as a partial source of information. The aim is then to merge the duplicate records, yielding one record with more complete information [Hyl96].

2 Previous duplicate detection algorithms

The standard method of detecting exact duplicates in a table is to sort the table and then to check if neighboring tuples are identical [BD83]. The approach can be extended to detect approximate duplicates. The idea is to do sorting to achieve preliminary clustering, and then to do pairwise comparisons of nearby records [NKAJ59, GBDH76]. Sorting is typically based on an application-specific key chosen to make duplicate records likely to appear near each other.

[HS95] compare nearby records by sliding a window of fixed size over the sorted database. If the window has size W then record i is compared with records $i-W+1$ through $i-1$ if $i \geq W$ and with records 1 through $i-1$ otherwise. The number of comparisons performed is $O(TW)$ where T is the total number of records in the database.

In order to improve accuracy, the results of several passes of duplicate detection can be combined [NKAJ59, KA85]. Typically, combining the results of several passes over the database with small window sizes yields better accuracy for the same cost than one pass over the database with a large window size.

[HS95] combine the results of multiple passes by explicitly computing the transitive closure of all discovered pairwise "is a duplicate of" relationships. If record R_1 is a duplicate of record R_2, and record R_2 is a duplicate of record R_3, then by transitivity R_1 is a duplicate of record R_3.

[Hyl96] uses a different, more expensive, method to do a preliminary grouping of records. Each record is considered separately as a "source record" and used to query the remaining records in order to create a group of potentially matching records. Then each record in

*Research supported by a Doctoral Fellowship from AT&T Laboratories.

†Research supported by a Hellman Faculty Fellowship from UCSD.

the group is compared with the source record using his pairwise matching procedure.

3 Pairwise record matching algorithms

Every duplicate detection method proposed to date, including ours, requires an algorithm for detecting "is a duplicate of" relationships between pairs of records. Typically this algorithm is relatively expensive computationally, and grouping methods as described in the previous section reduce the number of times that it must be applied.

The pairwise record matching algorithms used in most previous work have been application-specific. For example, [HS95] use production rules based on domain-specific knowledge, which are first written in OPS5 and then translated by hand into C. In [ME96], we studied three record matching algorithms and evaluated their performance on real-world datasets. One of the algorithms proposed is an edit-distance algorithm that is a variant of the well-known Smith-Waterman algorithm, which was originally developed for finding evolutionary relationships between biological protein or DNA sequences. In this paper, we use it as a domain-independent algorithm to detect duplicate records.

We call a record matching algorithm domain-independent if it can be used without any modifications in a range of applications. By this definition, the Smith-Waterman algorithm is domain-independent under certain assumptions: that records have the same high-level schema and that they be made up of alphanumeric characters. The second assumption is needed because any edit-distance algorithm assumes that records are strings over some fixed alphabet of symbols. Naturally this assumption is true for a wide range of databases, including those with numerical fields such as social security numbers that are represented in decimal notation. The first assumption is needed because the Smith-Waterman algorithm does not address the problem of duplicate records containing fields which are transposed.

We illustrate domain-independence by using the same Smith-Waterman algorithm in two different applications. One application is to detect duplicate records in randomly created mailing lists. The other is to cluster a database of bibliographic records into groups of records which refer to the same publication.

Briefly, given two sequences, the algorithm of [SW81] uses dynamic programming to find the lowest cost series of changes that converts one sequence into the other. Costs for individual changes, which are mutations, insertion, or deletions, are parameters of the algorithm. Intuitively, since the algorithm allows for gaps of unmatched characters, it should perform well for many abbreviations, and when records have missing information or minor syntactical differences, including typographical mistakes.

4 Computing the transitive closure of duplicate relationships

Under the assumption of transitivity, the problem of detecting duplicates in a database can be described in terms of determining the connected components of an undirected graph. Let the vertices of a graph G represent the records in a database of size T. Initially, G will contain T unconnected vertices, one for each record in the database. There is an undirected edge between two vertices if and only if the records corresponding to the pair of vertices are found to match. When considering whether to apply the expensive pairwise record matching algorithm to two records, we can query the graph G. If both records are in the same connected component, then it has been determined previously that they are approximate duplicates, and the comparison is not needed. If they belong to different components, then it is not known whether they match or not. If comparing the two records results in a match, their respective components should be combined to create a single new component. This is done by inserting an edge between the vertices that correspond to the records compared.

At any time, the connected components of the graph G correspond to the transitive closure of the "is a duplicate of" relationships discovered so far. Consider three records R_u, R_v, and R_w and their corresponding nodes u, v, and w. When the fact that R_u is a duplicate of record R_v is detected, an edge is inserted between the nodes u and v, thus putting both nodes in the same connected component. Similarly, when the fact that R_v is a duplicate of R_w is detected. Transitivity of the "is a duplicate of" relation is equivalent to reachability in the graph. Since w is reachable from u (and vice versa), the corresponding records R_u and R_w are duplicates.

There is a well-known data structure that efficiently solves the problem of determining and maintaining the connected components of an undirected graph, called the union-find data structure [Tar75, CLR90]. This data structure keeps a collection of disjoint updatable sets, where each set is identified by a representative member of the set. The data structure has two operations:

$Union(x, y)$ combines the sets that contain node x and node y, say S_x and S_y, into a new set that is their union $S_x \cup S_y$. A representative for the union is chosen, and the new set replaces S_x and S_y in the collection of disjoint sets.

$Find(x)$ returns the representative of the unique set containing x. If $Find(x)$ is invoked twice without modifying the set between the requests, the answer is the same.

To find the connected components of a graph G, we first create $|G|$ singleton sets, each containing a single node from G. For each edge $(u, v) \in E(G)$, if $Find(u) \neq Find(v)$ then we perform $Union(u, v)$. At any time, two nodes u and v are in the same connected component if and only if their sets have the same representative, that is if and only if $Find(u) = Find(v)$.

5 The overall priority queue algorithm

This section describes the high-level strategy used in our algorithm. As done by other algorithms, we do multiple passes of sorting and scanning. Unlike previous algorithms, we propose to sort the records in each pass according to domain-independent sorting criteria. Our algorithm uses two passes, the first pass treats each record as one long string and sorts these lexicographically, reading from left to right. The second pass does the same reading from right to left.

The algorithm uses a priority queue of sets of records belonging to the last few clusters detected. The algorithm scans the database sequentially and determines whether each record scanned is or is not a member of a cluster represented in a priority queue. To determine cluster membership, the algorithm uses the *Find* operation from the previous section. If the record is already a member of a cluster in the priority queue, then the next record is scanned. If the record is not already a member of any cluster kept in the priority queue, then the record is compared to representative records in the priority queue using the Smith-Waterman algorithm. If one of these comparisons succeeds, then the record belongs in this cluster and the *Union* operation is performed on the two sets. On the other hand, if all comparisons fail, then the record must be a member of a new cluster not currently represented in the priority queue. Thus, the record is saved in the priority queue as a singleton set. The algorithm is explained in more detail in the remainder of this section.

The priority queue contains a fixed number of sets of records. In most of our experiments this number is 4. Each set contains one or more records from a detected cluster. For practical reasons, entire clusters should not always be saved since they may contain many records. On the other hand, a single record may be insufficient to represent all the variability present in a cluster. The set representing the cluster with the most recently detected cluster member has highest priority in the queue, and so on. Intuitively, the sets in the priority queue represent the last few clusters detected. While in most of our experiments the algorithm keeps only the last 4 clusters detected, this translates to many individual records due to a couple of reasons. First, the sorting pass brings similar records close to each other and thus most of the members of a cluster are found sequentially in the database. The second reason is that only a fraction of these records are kept in each set that is in the priority queue. The fraction that is kept makes up a representation of the cluster that the records belong to.

Our algorithm scans through the database sequentially. Suppose that record R_j is the record currently being considered. The algorithm first tests whether R_j is already known to be a member of one of the clusters represented in the priority queue. This test is done by comparing the cluster representative of R_j to the cluster representative of each set in the priority queue. If one of these comparisons is successful, then R_j is already known to be a member of the cluster represented by the set in the priority queue. The set is given the highest priority and we continue with the next record.

Whatever their result, these comparisons are computationally inexpensive because they are done just with *Find* operations. In the first pass, *Find* comparisons are guaranteed to fail, and thus avoided, since we scan the records in the database sequentially and this is the first time each record is encountered.

Second, in the case where R_j is not a known member of an existing priority queue cluster, we use the Smith-Waterman algorithm to compare R_j with records in the priority queue. For each set in the priority queue, the algorithm scans through the records R_i of the set. R_j is then compared to R_i using the Smith-Waterman algorithm. If this operation detects a match, then R_j's cluster is combined with R_i's cluster, using a $Union(R_i, R_j)$ operation. The *Union* operation modifies the union-find data structure that is keeping track of the detected clusters; it does not change the priority queue. In addition, R_j may also be included in the priority queue set that represents R_i's cluster. Specifically, R_j is included if its matching score is *below* a certain threshold. This inclusion threshold is higher than the threshold for declaring a match, but lower than 1.0. Intuitively, if R_j is very similar to R_i it is not necessary to include it in the subset representing the cluster, but if R_j is only somewhat similar, i.e. its degree of match is below the inclusion threshold, then including R_j in the subset will help in detecting future members of the cluster.

On the other hand, if the Smith-Waterman comparison between R_i and R_j yields a very low score, below a certain "bad miss" threshold, then the algorithm continues directly with the next highest priority set in the queue. The intuition here is that if R_i and R_j have no similarity at all, then comparisons of R_j with other members of the cluster containing R_i will likely also fail.

Finally, if R_j is compared to members of each set in the priority queue without detecting that it is a duplicate of any of these, then R_j must be a member of a cluster not currently represented in the priority queue. R_j is thus saved as a singleton set in the priority queue, with the highest priority. If this action causes the size of the priority queue to exceed its limit then the lowest priority set is removed.

Note that the clusters are maintained in memory by using only the record identifiers, not the actual record. The union-find data structure from the previous section is used to manage all the clusters being detected as the algorithm executes. When we refer to records found in the priority queue, the full record is stored in addition to the record identifier. We can do this because only a small portion of the total number of records in the database is ever kept in the priority queue.

6 Experimental results

The first experiments reported here use databases that are mailing lists generated randomly by software designed and implemented by [Her96]. Each record in a mailing list contains nine fields: social security number, first name, middle initial, last name, address, apartment, city, state, and zip code. All field values are chosen randomly and independently. Personal names are chosen from a list of 63000 real names. Address fields are chosen from lists of 50 state abbreviations, 18670 city names, and 42115 zip codes.

Once the database generator creates a random record, it creates a random number of duplicate records according to a fixed probability distribution. When it creates a duplicate record, the generator introduces errors (i.e. noise) into the record. Possible errors range from small typographical slips to complete name and address changes. The generator introduces typographical errors according to frequencies known from previous research on spelling correction algorithms [Kuk92]. Edit-distance algorithms are designed to detect some of the errors introduced, however, our algorithm was developed without knowledge of the particular error probabilities used by the database generator of [Her96]. The pairwise record matching algorithm of [HS95] has special rules for transpositions of entire words, complete changes in names and zip codes, and social security number omissions, while our Smith-Waterman algorithm variant does not.

Table 1 contains example pairs of records chosen as especially instructive by [Her96], with pairwise scores assigned by the Smith-Waterman algorithm. The first pair is correctly detected to be duplicates by the rules of [HS95]. The Smith-Waterman algorithm classifies it as duplicate given any threshold below 0.68. The equational theory does not detect the second pair as duplicates. The Smith-Waterman algorithm performs correctly on this pair when the duplicate detection threshold is set at 0.41 or lower. Finally, the equational theory falsely finds the third and fourth pairs to be duplicates. The Smith-Waterman algorithm performs correctly on these pairs with a threshold of 0.37 or higher. These examples suggest that we should choose a threshold around 0.40. However this threshold is somewhat aggressive. Small further experiments show that a more conservative threshold of 0.50 detects most real duplications while keeping the number of false positives negligible.

The measure of accuracy used in this paper is based on the number of clusters detected that are "pure". A cluster is pure if and only if it contains only records that belong to the same true cluster of duplicates. A cluster detected by a duplicate detection algorithm can be one of the following:

1. the cluster is equal to a true cluster or,
2. the cluster is a subset of a true cluster or,
3. the cluster contains some of two or more true clusters.

By our definition, a pure cluster falls in either of the first two cases above. The last case is what we refer to as "impure" clusters. A good detection algorithm will have 100% of the detected clusters as pure and 0% as impure.

The sections that follow provide the results from experiments performed in this study. The figures show the number of pure and impure clusters detected by our algorithm. We also show the number of true clusters in the database, and the number of clusters detected by the method of [HS95]. Unfortunately the Merge/Purge engine software does not distinguish between pure and impure clusters, so the results reported here for it slightly overstate its accuracy.

6.1 Varying number of duplicates per record

A good duplicate detection algorithm should be almost unaffected by changes in the number of duplicates that each record has. To study the effect of increasing this number, we varied the number of duplicates per record using a Zipf distribution. Zipf distributions give high probability to small numbers of duplicates, but still give non-trivial probability to large numbers of duplicates. A Zipf distribution has two parameters $0 \leq \theta \leq 1$ and $1 \leq D$. For $1 \leq i \leq D$ the probability of i duplicates is $ci^{\theta-1}$ where the normalization constant $c = 1/\sum_{i=1}^{D} i^{\theta-1}$. Having a maximum number of duplicates D is necessary because $\sum_{i=1}^{\infty} i^{\theta-1}$ diverges if $\theta \geq 0$.

Four databases were created, each with a different value of the parameter θ from the set $\{0.1, 0.2, 0.4, 0.8\}$. The maximum number of duplicates per original record was kept constant at 20. The noise level was also maintained constant. The sizes of the databases ranged from 301153 to 480239 total records.

In all experiments, the Merge/Purge engine was run with fixed window of size 10, as in most experiments performed by [HS95]. Our duplicate detection algorithm used a priority queue containing at most 4 sets of records. This number was chosen to make the accuracy

Equational theory	Smith-Waterman score	Soc. Sec. number	Name	Address	City, State Zip code
True positive	0.6851	*missing*	Colette Johnen	600 113th St. apt. 5a5	*missing*
		missing	John Colette	600 113th St. ap. 585	*missing*
False negative	0.4189	152014425	Bahadir T Bihsya	220 Jubin 8s3	Toledo OH 43619
		152014423	Bishya T ulik	318 Arpin St 1p2	Toledo OH 43619
False positive	0.3619	274158217	Frankie Y Gittler	PO Box 3628	Gresham, OR 97080
		267415817	Erlan W Giudici	PO Box 2664	Walton, OR 97490
	0.1620	760652621	Arseneau N Brought	949 Corson Ave 515	Blanco NM 87412
		765625631	Bogner A Kuxhausen	212 Corson Road 0o3	Raton, NM 87740

Table 1: Example pairs of records and the status of the matching algorithms.

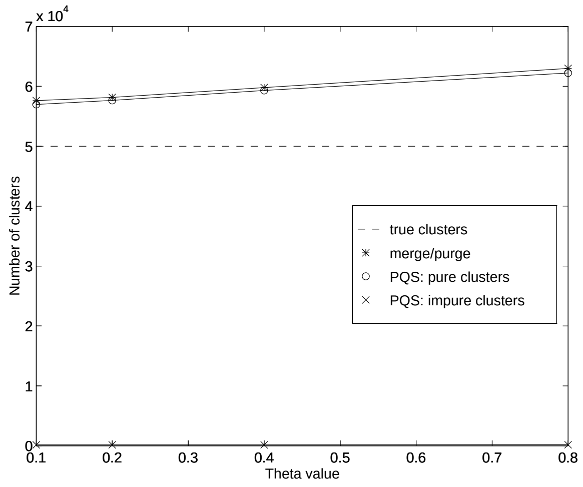

Figure 1: Accuracy results for varying the number of duplicates per original record using a Zipf distribution.

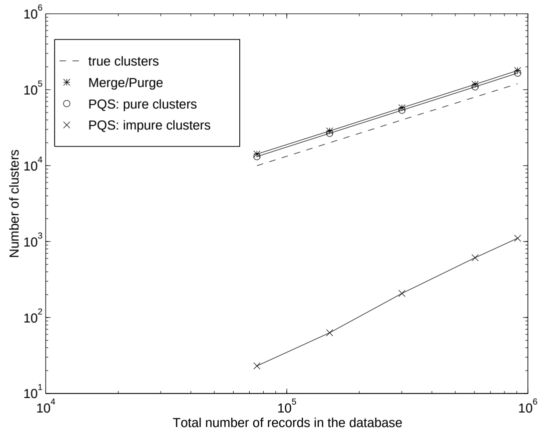

Figure 2: Accuracy for varying DB sizes (log-log plot).

of both algorithms approximately the same. Of course, it is easy to run our algorithm with a larger priority queue in order to obtain greater accuracy.

Figure 1 shows that our algorithm performs slightly better than the Merge/Purge engine. The number of pure clusters detected by both algorithms increases slowly as the value of theta is increased. This increase constitutes a decrease in accuracy since we want to get as close to the number of true clusters as possible. As desired, the number of impure clusters remains very small throughout. The fact that nearly 100% of the detected clusters are pure suggests that we could relax various parameters of our algorithm in order to combine more clusters, without erroneously creating too many impure clusters.

6.2 Varying the size of the database

Here we study how the size of the database affects the accuracy of duplicate detection. We consider databases with respectively 10000, 20000, 40000, 80000 and 120000 original records. In each case, duplicates are generated using a Zipf distribution with a high noise level, Zipf parameter $\theta = 0.40$, and 20 maximum duplicates per original record. The largest database considered here contains over 900000 records in total.

Figures 2 and 3 clearly display the benefits of our algorithm. Figure 2 shows that the number of clusters detected by both strategies is similar, with our strategy having slightly better accuracy. While both algorithms are detecting nearly the same number of clusters, our algorithm is doing many fewer pairwise record comparisons. This is shown in figure 3. For the largest database tested, our strategy performed about 3.4 million comparisons, while the Merge/Purge engine performed about 18 million comparison, over 5 times as many comparisons as our strategy. This savings in number of comparisons performed is crucial when dealing with very large databases. Our algorithm responds adaptively to the size and the homogeneity of the clusters discovered as the database is scanned. These results do not depend on the record matching algorithm which is used. Instead, the savings is due to the maintenance of the clusters in the union-find data structure and the use of the priority queue to determine which records to compare. In addition to these benefits, the experiments also show that there is no loss in accuracy when using the Smith-Waterman algorithm over one which uses domain-specific knowledge.

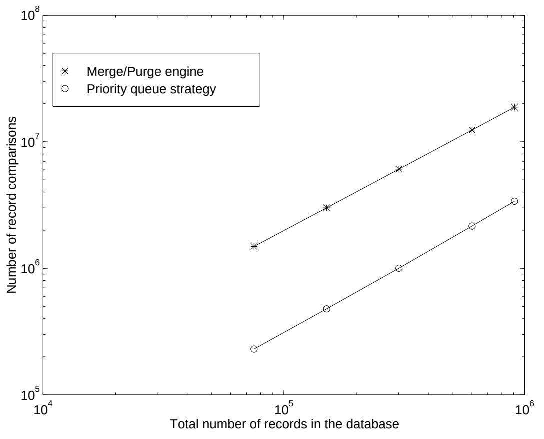

Figure 3: Number of comparisons (log-log plot).

Cluster size	Number of clusters	Number of records	% of all records
1	118149	118149	46.40%
2	28323	56646	22.25%
3	8403	25209	9.90%
4	3936	15744	6.18%
5	1875	9375	3.68%
6	1033	6198	2.43%
7	626	4382	1.72%
8+	1522	18915	7.40%
total	163867	254618	100.00%
[Hyl96]	162535	242705	100.00%

Table 2: Results on a database of bibliographic records

7 Experiments with a real database

Here we study the effectiveness of our algorithm on a real database of bibliographic records describing documents in various fields of computer science published in several sources. The database is a slightly larger version of one used by [Hyl96]. The task here is to create clusters that contain all records about the same entity. An entity in this context is one document, called a "work", that may exist in several versions. A work may be a technical report which later appears in the form of a conference paper, and still later as a journal paper. While the bibliographic records contain many fields, the algorithm of [Hyl96] considers only the author and the title of each document.

The bibliographic records were gathered from two major collections available over the Internet. The primary source is *A Collection of Computer Science Bibliographies* assembled by Alf-Christian [Ach96]. The secondary source is a collection of computer science technical reports produced by five major universities in the CS-TR project [YGM95, Kah95]. Since the records come from collections of bibliographies, the database contains multiple BibTeX records for the same document. In addition, due to the different sources, the records are subject to typographical errors, errors in the accuracy of the information they provide, variation in how they abbreviate author names, and more. In total, the database we use contains 254618 records.

To apply our duplicate detection algorithm to the database, first we created simple representative records from the complete BibTeX records. Each derived record contained the author names and the document title from one BibTeX record. As in all other experiments, our algorithm used two passes over the database and used the same domain-independent sorting criteria. Small experiments allowed us to determine that the best Smith-Waterman algorithm threshold for a match in this database was 0.60. The threshold is higher for this database because it has less noise than the synthetic databases used in other experiments.

Results for our algorithm using a priority queue size of 4 are presented in Table 2. The algorithm detected a total of 163867 clusters, with an average of 1.60 records per cluster. We do not know the true number of duplicate records in this database. However, based on visual inspection, the great majority of detected clusters are pure. In addition, our results are comparable to the results of [Hyl96] on almost the same database. Both algorithms detected almost the same number of clusters of each different size between 1 and 7. However our algorithm did detect more larger clusters. Hylton reports detecting 772 clusters of size 8 or larger, accounting for 3.1% of all the records in the database. Our algorithm detects 1522 clusters of at least 8 records, accounting for 7.4% of all records. Manual inspection of these clusters shows that they are mostly impure. It would be beneficial to rerun our algorithm on these records with a higher Smith-Waterman algorithm threshold and a larger priority queue size, in order to subdivide the impure clusters.

Hylton reports making 7.5 million comparisons to determine the clusters. Our algorithm performed just over 1.6 million comparisons. This savings of over 75% is comparable to the results on synthetic databases.

8 Conclusion

The duplicate detection methods described in this paper improved previous related work in three ways. Our first contribution is an approximate record matching algorithm that is relatively domain-independent. Our second contribution is to show how to compute the transitive closure of "is a duplicate of" relationships incrementally, using the union-find data structure. Last but not least, our third contribution is a heuristic method for minimizing the number of expensive pairwise record comparisons that must be performed while comparing individual records with potential duplicates. It is important to note that our second and third contributions can be combined with any pairwise record matching algorithm.

9 Acknowledgments

We wish to thank Sal Stolfo and Mauricio Hernández for highly useful discussions. Mauricio Hernández most generously made available to us the mailing list database generator that he implemented along with parameter settings used in his experiments. He also provided the Merge/Purge engine used as a standard for comparison in our experiments. Larry Carter and Kang-Su Gatlin kindly provided the implementation of the Smith-Waterman algorithm that we adapted to run in linear time.

References

[Ach96] Alf-Christian Achilles. A collection of computer science bibliographies. URL, 1996. `http://liinwww.ira.uka.de/bibliography/index.html`.

[BD83] D. Bitton and D.J. DeWitt. Duplicate record elimination in large data files. *ACM Transactions on Database Systems*, 8(2):255–65, 1983.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Roland L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[GBDH76] C. A. Giles, A. A. Brooks, T. Doszkocs, and D.J. Hummel. An experiment in computer-assisted duplicate checking. In *Proceedings of the ASIS Annual Meeting*, page 108, 1976.

[Her96] Mauricio Hernández. *A Generalization of Band Joins and the Merge/Purge Problem*. Ph.D. thesis, Columbia University, 1996.

[HS95] M. Hernández and S. Stolfo. The merge/purge problem for large databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 127–138, May 1995.

[Hyl96] Jeremy A. Hylton. Identifying and merging related bibliographic records. M.S. thesis, MIT, 1996. Published as MIT Laboratory for Computer Science Technical Report 678.

[KA85] Beth Kilss and Wendy Alvey, editors. *Record linkage techniques, 1985: Proceedings of the Workshop on Exact Matching Methodologies*, Arlington, Virginia, 1985. Internal Revenue Service, Statistics of Income Division.

[Kah95] Robert E. Kahn. An introduction to the CS-TR project [WWW document]. URL, December 1995. `http://www.cnri.reston.va.us/home/cstr.html`.

[Kuk92] K. Kukich. Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439, December 1992.

[ME96] Alvaro E. Monge and Charles P. Elkan. The field matching problem: Algorithms and applications. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, pages 267–270. AAAI Press, August 1996.

[NKAJ59] Howard B. Newcombe, J. M. Kennedy, S. J. Axford, and A. P. James. Automatic linkage of vital records. *Science*, 130:954–959, October 1959. Reprinted in [KA85].

[SSU95] A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database research: achievements and opportunities into the 21st century. A report of an NSF workshop on the future of database research, May 1995.

[SW81] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[Tar75] Robert E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22(2):215–225, 1975.

[YGM95] T.W. Yan and H. Garcia-Molina. Information finding in a digital library: the Stanford perspective. *SIGMOD Record*, 24(3):62–70, September 1995.