EFFICIENT AND EFFECTIVE DATA CLEANSING FOR LARGE DATABASE

LI ZHAO

NATIONAL UNIVERSITY OF SINGAPORE
2002

EFFICIENT AND EFFECTIVE DATA CLEANSING FOR LARGE DATABASE

LI ZHAO

(M.Sc., NATIONAL UNIVERSITY OF SINGAPORE)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2002

Acknowledgments

It's my pleasure to express my greatest appreciation and gratitude to my supervisor: Prof. Sung Sam Yuan. He provided many ideas and suggestions. It has been an honor and pleasure to work with him. Without his support and encouragement, this work would not have been possible.

Also I would like to thank my parents and my wife for their constant encouragement and concern. I am very grateful to their care, support, understanding and love.

Foremost, I am very thankful to NUS for the Research Scholarship, and to the department for providing me excellent working conditions during my research study.

Contents

List of Figures

List of Tables

Summary

Data cleansing recently receives a great deal of attention in data warehousing, database integration, and data mining etc. The mount of data handled by organizations has been increasing at an explosive rate, and the data is very likely to be dirty. Since "dirty in, dirty out", data cleansing is identified as of critical importance for many industries over a wide variety of applications.

Data cleansing consists of two main components, detection method and comparison method. In this thesis, we study several problems in data cleansing, discover similarity properties, propose new detection methods, and extend existing comparison method. Our new approaches show better performance in both efficiency and accuracy.

First we discover two similarity properties, *lower bound similarity property (LP)* and *upper bound similarity property (UP)*. These two properties state that, for any three records A, B and C, $Sim(A, C)$ (similarity of records A and C) can be lower bounded by $L_B(A, C) = Sim(A, B) + Sim(B, C) - 1$, and also upper bounded by $U_B(A, C) = 1 - |Sim(A, B) - Sim(B, C)|$. Then we show that a similarity method, LCSS, satisfies these two properties. By employing LCSS as

the comparison method, two new detection methods, RAR1 and RAR2, are thus proposed. RAR1 does slide a window on sorted dataset. In RAR1, an anchor record is chosen in the window to keep the similarities information with other records in the window. With this information, LP and UP are used to reduce comparisons. Performance tests show that these two methods are much faster and more efficient than existing methods.

To further improve the efficiency of our new methods, we propose a two-stage cleansing method. Since existing similarity methods are very costly, we propose a filtering scheme which runs very fast. The filter is a simple similarity method which only considers the characters in fields of records and does not consider the order of characters. However, the filter may produce some extra false positives. We thus perform pruning with more trustworthy and costly methods on the result obtained by the filter. This technique works because of the duplicate result obtained is normally far less than the initial comparisons taken.

Finally, we propose a dynamic similarity method, which is an extension scheme for existing comparison methods. Existing comparison methods do not address fields with NULL value well, which results in a loss of correct duplicate records. Therefore, we extend them by dynamically adjusting the similarity for field with NULL value. The idea behind dynamic similarity is from approximate functional dependency.

Chapter 1

Introduction

1.1 Background

Data cleansing, also called *data cleaning* or *data scrubbing*, deals with detecting and removing errors and inconsistencies from data in order to improve the quality of data [RD00]. It is a common problem in environments where records contain erroneous in a single database (e.g., due to misspelling during data entry, missing information and other invalid data etc.), or where multiple databases must be combined (e.g., in data warehouses, federated database systems and global web-based information systems etc.).

Motivation for Data Cleansing

The amount of data handled by organizations has been increasing at an explosive rate. The data is very likely to be dirty because of misuse of abbreviations, data

entry mistakes, duplicate records, missing values, spelling errors, outdated codes etc [Lim98]. A list of common causes of dirty data is described in [Mos98]. As the example shown in [LLL01], in a normal client database, some clients may be represented by several records for various reasons: (1) incorrect or missing data values because of data entry errors, (2) inconsistent value naming conventions because of different entry formats and use of abbreviations such "ONE" vs '1', (3) incomplete information because data is not captured or available, (4) clients do not notify change of address, and (5) client mis-spell their names or give false address (incorrect information about themselves). As a result, several records may refer to the same real world entity while not being syntactically equivalent. In [WRK95], errors in databases have been reported to be up 10% range and even higher in a variety of applications.

Dirty data will distort information obtained from it because of the "garbage in, garbage out" principle. For example, in data mining, dirty data will not be able to provide data miners with correct information. Yet it is difficult for managers to make logical and well-informed decisions based on information derived from dirty data. A typical example [Mon00] is the prevalent practice in the mass mail market of buying and selling mailing lists. Such practice leads to inaccurate or inconsistent data. One inconsistency is the multiple representations of the same individual household in the combined mailing list. In the mass mailing market, this leads to expensive and wasteful multiple mailings to the same household. Therefore, data cleansing is not an option but a strict requirement for improving

the data quality and providing correct information.

In [Kim96], data cleansing is identified as critical importance for many industries over a wide variety of applications, including marketing communications, commercial householding, customer matching, merging information systems, medical records etc. It is often studied in association with data warehousing, data mining, and database integration etc. Especially, data warehousing [CD97, JVV00] requires and provides extensive support for data cleansing. They load and continuously refresh huge amounts of data from a variety of sources so the probability that some of the sources contain "dirty data" is high. Furthermore, data warehouses are used for decision making, so that the correctness of their data is vital to avoid wrong conclusions. For instance, duplicated or missing information will produce incorrect or misleading statistics. Due to the wide range of possible data inconsistencies, data cleaning is considered to be one of the major problems in data warehousing. In [SSU96], data cleansing is identified as one of the database research opportunities for data warehousing into the 21^{st} century.

Problem Description and Formalization

Data cleansing generally includes many tasks because the errors in databases are wide and unknown in advance. It recently receives much attention and many research efforts [BD83, Coh98, DNS91, GFSS00, GFS+01a, GFS+01b, GIJ+01, GP99, Her96, HS95, HS98, Kim96, LSS96, LLL00, LLL01, Mon97, Mon00, Mon01, ME96, ME97, Mos98, RD00, RH01, Wal98, WRK95] are focused on it. One such

main and most important task is to de-duplicate records, which is different from, but related to, the schema matching problem [BLN86, KCGS93, MAZ96, SJB96]. Before the de-duplication, there is a pre-processing stage which detects and removes any anomalies in the data records and then provide the most consistent data for the de-duplication. The pre-processing usually (but not limit to) does spelling correction, data type checking, format standardization and abbreviation standardization etc.

Given the database having a set of records, the de-duplication is to detect all duplicates of each record. The duplicates include exact duplicates and also inexact duplicates. The inexact duplicates are records that refer to the same real-world entity while not being synthetically equivalent. If consider the transitive closure, the de-duplication is to detect all clusters of duplicates and each cluster includes a set of records that represent the same entity. The computing of transitive closure is an option in some data cleaning methods, but an inherent requirement in some other data cleansing methods. The transitive closure increases the number of correct duplicate pairs, and also increases the number of false positives (two records are not duplicate but detected as duplicate).

Formally, this de-duplication problem can be identified as follows. Let $\mathcal{D} = \{A_1, A_2, \cdots, A_N\}$ be the database, where A_i, $1 \leq i \leq N$, are records. Let $<A_i, A_j> = T$ denote that records A_i and A_j are duplicate, and

$$Dup(\mathcal{D}) = \{<A_i, A_j> \mid <A_i, A_j> = T,\ 1 \leq i, j \leq N \text{ and } i \neq j\}.$$

That is, $Dup(\mathcal{D})$ is the set of all duplicate pairs in $\mathcal{D}$. Then, given $\mathcal{D}$, the problem

is to find the $Dup(\mathcal{D})$.

Let $A_i \sim A_j$ be the equivalent relation among records that A_j is a duplicate record of A_i under transitive closure. That is $A_i \sim A_j$ if and only if there are records A_{i_1}, A_{i_2}, $\cdots$, A_{i_k}, such that $<A_i, A_{i_1}> = T$, $<A_{i_1}, A_{i_2}> = T$, $\cdots$, and $<A_{i_k}, A_j> = T$. Let $X_{A_i} = \{A_j | A_i \sim A_j\}$. Then $\{X_{A_i}\}$ are equivalent classes under this equivalent relation. Thus for any two records A_i and A_j, we have either $X_{A_i} = X_{A_j}$ or $X_{A_i} \cap X_{A_j} = \emptyset$. If the transitive closure is taken into consideration, the problem is then to find $TC(\mathcal{D}) = \{X_{A_i}\}$. More strictly, it is to find $TC_2(\mathcal{D}) = \{X_{A_i} || X_{A_i}| \geq 2\}$.

Existing Solutions

Given a database, to detect exact duplicates is a simple process and is well addressed in [BD83]. The standard method is to sort the database and then check if the neighboring records are identical. The more complex process is to detect the inexact duplicates, which leads to two problems: (1) which records need to be compared, and (2) how to compare the records to determine whether they are duplicate.

Thus, the (inexact) de-duplication consists of two main components: detection method and comparison method. A detection method determines which records will be compared, and a comparison method decides whether two records compared are duplicate.

In detection methods, the most reliable way is to compare every record with

every other record. Obviously this method guarantees that all potential duplicate records are compared and then provides the best accuracy. However, the time complexity of this method is quadratic. It takes $N(N-1)/2$ comparisons if the database has N records, which will take very long time to execute when N is large. Thus it is only suitable for small databases and is definitely impractical and infeasible for large databases.

Therefore, for large databases, approximate detection algorithms that take far less comparisons (e.g., $O(N)$ comparisons) are required. Some approximate methods have been proposed [DNS91, Her96, HS95, HS98, LLL00, LLL01, LLLK99, Mon97, Mon00, Mon01, ME97]. All these methods have a common feature as they compare each record with only a limited number of records with a good expected probability that most duplicate records will be detected. All these methods can be viewed as the variances of "sorting and then merging within a window". The sorting is to bring potential duplicate records close together. The merging is to limit that each record is only compared with a few neighborhood records.

Based on this idea, Sorted Neighborhood Method (SNM) is proposed in [HS95]. SNM takes only $O(\omega N)$ comparisons by sorting the database on a key and making pair-wise comparisons of nearby records by sliding a window, which has size ω, over the sorted database. Other methods, such as Clustering SNM [HS95], Multi-pass SNM [HS95], DE-SNM [Her96] and Priority Queue [ME97] etc., are further proposed to improve SNM on different aspects (either accuracy or time). More discussions and analysis on these detection methods will be shown in Section 2.2.

Name	Dept.	Age	Gender	Email
Li Zhao	Computer Science	-	-	-
Li Zhai	Computer Science	-	-	-

Table 1.1: Two records with a few information known.

Name	Dept.	Age	Gender	Email
Li Zhao	Computer Science	28	M	lizhao@comp.nus.edu.sg
Li Zhai	Computer Science	28	M	lizhao@comp.nus.edu.sg

Table 1.2: Two records with more information known.

As the detection methods determine which records need to be compared, parewise comparison methods are to decide whether two records compared are duplicate.

The comparison of records to determine their equivalence is a complex inferential process that needs to consider much more information in the compared records than the keys used for sorting. The more information there is in the records, the better inferences can be made.

For example, for the two records in Table 1.1, the values in the "Name" field are nearly identical, the values in the "Dept." field are exactly the same, and the values in the other fields ("Age", "Gender" and "Email") are unknown. We could either assume these two records represent the same person with a type error in the name of one record, or they represent different persons with similar name. Without

any further information, we may perhaps assume the later. However, as the two records shown in Table 1.2, with the values in the "Age", "Gender" and "Email" fields are known, we mostly determine that they represent the same person but with small type error in the "Name" field.

With the complex to compare records, one natural approach is using production rules based on domain-specific knowledge. Equational Theory [HS95] are inferences that dictate the logic of domain equivalence. A natural approach to specifying an equational theory is to use of a declarative rule language. In [HS95], OPS5 [For81] is used to specify the equational theory. Java Expert System Shell (JESS) [FH99], a rule engine and scripting environment, is employed by IntelliClean [LLL00]. The rules are represented as declarative rules in the JESS engine. An example is given in Section 2.3.1

An alternative approach is to compute the degree of similarity for records.

Definition 1.1 *A* similarity function $Sim : \mathcal{D} \times \mathcal{D} \mapsto [0, 1]$ *is a function that satisfies*

1. reflexivity: $Sim(A_i, A_i) = 1.0, \forall A_i \in \mathcal{D}$*; and*

2. symmetry: $Sim(A_i, A_j) = Sim(A_j, A_i), \forall A_i, A_j \in \mathcal{D}$.

Thus the similar of records is viewed as the degree of similarity, which is a value between 0.0 and 1.0. Commonly, 0.0 means certain non-equivalence and 1.0 means certain equivalence [Mon00]. A similarity function is *well-defined* if it satisfies 1) similar records will have large value (similarity) and 2) dissimilar records will have small value.

To determine whether two records are duplicate, a comparison method will typically just compare their similarity to a threshold, say 0.8. If their similarity is larger than the threshold, then they are treated as duplicate. Otherwise, they are treated as non-duplicate. Notice that the threshold are not given at random. It highly depends on the domain and the particular comparison methods in use.

Notice that the definition of Sim is domain-independent and works for databases of any kind of data type. However, this approach is generally based on the assumption that the value of each field is a string. Naturally this assumption is true for a wide range of databases, including those with numerical fields such as social security numbers represented in decimal notation. In [ME97], this assumption is also identified as a main domain-independent factor. Further note that rule-based approach can be applied on various data types, but currently, their discussions and implementations are only on string data as well since the string data is ubiquitous.

With this assumption, comparing two records is equal to compare two sets of strings where each string is for a field. Then any approximate string matching algorithms can be used as the comparison method.

Edit Distance [WF74] is a classic method in comparing two strings and has received much attention and widely used in many applications. It is the minimum number of insertions, deletions, and substitutions needed to transform one string into another. Edit distance returns an integer value but this value can be easily transfered (normalized) to a similarity value. The Smith-Waterman algorithm [SW81], a variant of edit distance, was employed in [ME97]. Longest Com-

mon Subsequence [Hir77], to find the maximum length of a common substring of two strings, is also used to compare two strings. Longest Common Subsequence is often studied associated with Edit Distance, and both can be solved by Dynamic Programming in $O(nm)$ time. *Record Similarity (RS)* was introduced in [LLLK99], in which record equivalence is determined by viewing records similarity at three levels: token, field and record. The string value in each field is parsed as tokens by using a set of delimiters such as space and punctuations. Field weightage was introduced on each field to reflect the different importance. In Section 2.3, we will discuss these comparison methods in more details.

One issue should be addressed is that whether two records are equivalent (duplicate) is a semantical problem, i.e., whether they represent the same real-world entity. However, the record comparison algorithms which solve this problem depend on the syntax of the records. The syntactic calculations performed by the algorithms are only approximates of what we really want - semantic equivalence. In such calculations, errors are possible to occur, that is, correct duplicate records compared may not be discovered and false positives may be introduced.

All feasible detection methods, as we have shown, are approximate. Since none of the detection methods can guarantee to detect all duplicate records, it is possible that two records are duplicate but will not be detected. Further, all comparison methods are also approximate, as shown above, and none of them is completely trustworthy. Thus, no data cleansing method (consisting of detection methods and comparison methods) guarantees that it can find out exactly all the duplicate

pairs, $Dup(\mathcal{D})$. It may not find some correct duplicate pairs and also introduce some false positives.

The accuracy of algorithms corresponding to retrieval effectiveness can be measured by recall and precision [LLL00]. Recall is the proportion of relevant information (i.e. truly matching records) actually retrieved (i.e. detected), while precision is the proportion of retrieved information that is relevant. More precisely, given a data cleansing method, let $DR(\mathcal{D})$ be the duplicate pairs found by it, then $DR(\mathcal{D}) \cap Dup(\mathcal{D})$ is the set of correct duplicate pairs and $DR(\mathcal{D}) - Dup(\mathcal{D})$ is the set of false positives. Thus the recall is $\frac{|DR(\mathcal{D}) \cap Dup(\mathcal{D})|}{|Dup(\mathcal{D})|}$ and false-positive error is $\frac{|DR(\mathcal{D}) - Dup(\mathcal{D})|}{|DR(\mathcal{D})|}$. The false positive error is the antithesis of the precision measure. The recall and precision are two important parameters to determine whether a method is good enough, and whether a method is superior to another one. In addition, time is another important parameter and must be taken into consideration. Surely, comparing each record with every other record and using the most complicate rules as the data cleansing method will obtain the best accuracy. However, it is infeasible for large database since it cannot finish in reasonable time. Generally, more records compared and a more complicate comparison method used will obtain a more accuracy result, but this takes more time. Therefore, there is a tradeoff between accuracy and time and each data cleansing method has its own tradeoff.

1.2 Contributions

Organizations today are confronted with the challenge of handling an everincreasing amount of data. It's not uncommon that that the data handled by organizations has several hundred Megabytes or even several Terabytes. Thus the database may have several millions or even billions records. As the size of the database increases, the time in data cleansing grows linearly. For very large databases, the data cleansing may take a very long time. As the example shown in [HS95], a database with 2,639,892 records was processed in 2172 seconds by SNM. Given a database with 1,000,000,000 records, SNM will need to process approximately $1 \times 10^9 \times \frac{2172}{2639892} s = 8.2276 \times 10^5 s \approx 10$ days. Therefore, more efficient and scalable data cleansing methods are definitely required.

Further, existing comparison methods prove to have good performances in capturing duplicate records. However, they all have a common drawback, i.e., they implicitly assume that the values in all fields are known, and NULL values on fields are simply treated as empty strings. But, in practice, databases to be cleansed very likely have records with NULL values. Treating the NULL values as empty strings is then not a good method and will result in a loss of correct duplicate records. Therefore, more considerations on fields with NULL values are required.

In this thesis, the comparison methods discussed are similarity-based. The major contributions of this thesis are summarized as follows:

(1) *We propose two new data cleansing methods, called RAR1 (Reduction using one Anchor Record) and RAR2 (Reduction using two Anchor Record), which*

are much more efficient and scalable than existing methods.

Existing detection methods are independent from the comparison methods. This independence gives freedom for applications but will result in a loss of useful information, which can be used to save expensive comparisons. Instead, we propose two new detection methods, RAR1 and RAR2 which can efficiently use the information provided by comparison methods, thus saving a lot of unnecessary comparisons.

RAR1 is an extension on the existing method SNM. In SNM, new record moving into the window needs to compare with all other records in the window. However, not all these comparisons are necessary. Instead, in RAR1, an anchor record is chosen in the window. New record is first compared with the anchor record and this similarity information is saved. For the other records in the window, two similarity bound properties are tried to determine whether the new record should compare with them or not. RAR2 is the same as RAR1 but has two anchor records. Detail description for the similarity bound properties, RAR1, and RAR2 is given in Chapter 3.

(2) *We propose a fast filtering scheme for data cleansing. The scheme not only inherits the benefit of RAR1 and RAR2 but also further improves the performance greatly.*

Large proportion of time in data cleansing is spent on the comparisons of records. We can reduce the number of comparisons with RAR1 and RAR2. Then we show how to reduce the time for each comparison by use filtering techniques.

Existing comparison methods (e.g, Edit Distance, Record Similarity) are in $O(nm)$ time thus quite costly. Generally only a few comparisons will detect duplicate records. Intuitively, we can first do a fast comparison as filter to obtain candidate duplicate result, then use existing comparison methods on the candidate duplicate result only. Based on this, we propose a fast filtering scheme with pruning on the result to achieve the best performance on both efficiency and accuracy. Detail discussion on the filtering scheme is shown in Chapter 4.

(3) *We propose a dynamic similarity scheme for handling field with NULL value. This scheme can be seamlessly integrated with all the existing comparison methods.*

Existing comparison methods do not deal with field with NULL values well. We propose the *Dynamic Similarity*, a simple yet efficient method, which dynamically adjusts the similarity for field with NULL value. For each field, there are a set of dependent fields associated with it. For any field with NULL value, the dependent fields will be used to determine its similarity. In Chapter 5, we will discuss this in details.

1.3 Organization of the Thesis

The rest of this thesis is organized as follows.

In the next chapter, we describes the research work that has been done in the data cleansing field. In Chapter 3, we propose two new efficient data cleansing

methods, called RAR1 and RAR2. In Chapter 4, we introduce a filtering scheme that further improves the result on Chapter 3. After that, in Chapter 5, we present a dynamic similarity method, which is an extension scheme for existing comparison methods. Finally, we make some concluding remarks and discuss future works in Chapter 6.

To be focused and consistent, in this thesis, we only discuss my research works on the data cleansing field. Most of the results in this thesis have been presented in [LSQS02, LSSL02, QSLS03, SL02, SLS02]. Other research works can be found in [SLTN03, SSLT02].

Chapter 2

Previous Works

In this chapter, first we simply show the pre-processing stage needed before cleansing. Then we discuss the two components, detection methods and comparison methods, in data cleansing in more details. The detection methods detect which records need to be compared and then let the comparison methods do the actual comparisons to determine whether the records are duplicate. Currently, the detection methods and the comparison methods are independent, that is, any detection method can be combined with any comparison method. With this independence, we separate the discussions of the detection methods and comparison methods in this chapter. This discussion is focused on the algorithm-level data cleansing, which is fundamental in data cleansing and much related to our works. For reader to have more understanding on data cleansing, we also simply introduce other works on data cleansing.

2.1 Pre-processing

Given a database, before the de-duplication, there is generally a pre-processing [HS95, LLL00] on the records in the database. Pre-processing the records will increase the chance of finding duplicate records in the later cleansing. The pre-processing itself is quite important in improving the data quality. In [LLL00], the pre-processing is identified as the first stage in the IntelliClean data cleansing framework.

The main task of the pre-processing is to provide the most consistent data for subsequent cleansing process. The data records are first conditioned and scrubbed of any anomalies that can be detected and corrected at this stage. The following list shows the most common jobs that can be performed in the pre-processing. However, sometimes, some domain-specific jobs are required, which are different from database to database.

Spelling correction Some misspellings may exist in the database, such as "Singapore" may be mistakenly typed as "Singpore". Spelling correction algorithms have received a large amount of attention for decades [Bic87, Kuk92]. Most of the spelling correction algorithms use a corpus of correctly spelled words from which the correct spelling is selected.

Data type check and format standardization Data type check and format standardization can also be performed, such as, in the "data" field, 1 Jan 2002 and 01/01/2002 can be standardized to one fixed format.

Inconsistent abbreviation standardization Inconsistent abbreviations used in

Abbreviation	Word
NUS	National University of Singapore
CS	Computer Science
RD.	Road
RD	Road

Table 2.1: Example of an abbreviation file.

the data can also be resolved. For example, all occurrences of "Rd." and "Rd" in the address field will be replaced by "Road". Occurrences of '1' and 'A' in the sex field will be replaced by 'Male', and occurrences of '2' and 'B' will be replaced by 'Female'. An external source file containing the abbreviations of words is needed. Table 2.1 shows one example.

2.2 Detection Methods

For each record, only a very limited number of records compared with it are duplicate. As we have explained in Section 1.1, all existing (feasible) detection methods are approximate methods and they are the variances of "sorting and then merging within a window". However, they differ on deciding which records are needed to be compared.

Sorted Neighborhood Method (SNM)

The *Sorted Neighborhood Method (SNM)* is proposed in [HS95]. One obvious method for bringing duplicate records close together is sorting the records over the most important discrimination key attribute of the data. After the sort, the comparison of records is then restricted to a small neighborhood within the sorted list. Sorting and then merging within a window is the essential approach of a Sort Merge Band Join as described by DeWitt [DNS91]. SNM can be summarized in three phases:

- **Create Key:** Compute a key for each record in the list by extracting relevant fields or portions of fields;
- **Sort Data:** Sort the records in the data list using the key;
- **Merge:** Move a fixed size window through the sequential list of records limiting the comparisons for duplicate records to those records in the window. If the size of the window is ω records, then every new record entering the window is compared with the previous $\omega - 1$ records to find duplicate records. The first record in the window slides out of the window (see Figure 2-1).

The effectiveness of this approach is based on the quality of the chosen keys used in the sort. The key creation in SNM is a highly knowledge-intensive and domain-specific process [HS98]. Poorly chosen keys will result in a poor quality result, i.e., records that are duplicate will be spread out far apart after the sort and hence will not be discovered. As an example, if the "gender" field in a database is

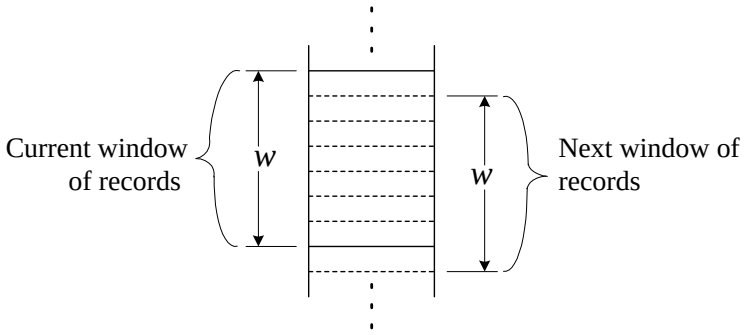

Figure 2-1: The merge phase of SNM.

chosen as the key, obviously, a lot of duplicate records would not be close together. Thus keys should be chosen so that the attributes with the most discriminatory power should be the principal field inspected during the sort. This means that similar and duplicate records should have nearly equal key values. However, since the data is (likely) corrupted and keys are extracted directly from the data, then the key will also be likely corrupted. Thus, a substantial number of duplicate records may not be caught.

Further, the "window size" used in SNM is an important parameter that affects the performance. Increasing the window size will increase the number of duplicate pairs found but also, on the other hand, increase the time taken. The performance result in [HS95] shows that the accuracy increases slowly but the time increases fast when increasing the window size. Thus, increasing the window size does not help much if taking in consideration that the time complexity of the procedure goes up as the window size increase, and it is fruitless at some point to use a larger

window.

Clustering SNM

As the database becomes very large, sorting the data may take a great amount of time although it may not be the dominant cost of cleansing. In [HS95], the authors considered an alternative to sorting based upon first partitioning the dataset into independent clusters using a key extracted from the data. Then SNM is applied to each individual cluster independently. This method is called as *Clustering SNM*.

Since the dataset is partitioned into small clusters and do not need a completely sorted database, the clustering SNM takes less time than SNM (sorting some small datasets is faster than sorting a large dataset). However, two duplicate records may be partitioned in two different clusters, then they cannot be detected, which results in a decrease of the number of correct duplicate results. Thus the clustering SNM provides the trade-off between time and accuracy.

Multi-pass SNM

In general, no single key will be sufficient to catch all duplicate records and the number of duplicate records missed by one run of the SNM can be large. For instance, if an employee has two records in the database, one with social security number 193456782 and another with social security number 913456782, and if the social security number is used as the principal field of the key, then it is very unlikely that both records will be in the same window, i.e., these two records will

be far apart in the sorted database hence they will not be detected.

To increase the number of duplicate records detected, *Multi-pass SNM* [HS95] is then proposed. Multi-pass SNM is to execute several independent runs of SNM, each using a different key and a relatively small window. Each independent run will produce a set of pairs of duplicate records. The results is the union of all pairs discovered by all independent runs, plus all those pairs that can be inferred by transitive closure. The transitive closure is executed on pairs of record id's, and fast solutions to compute transitive closure exist [AJ88, ME97].

This approach works based on the nature of errors in the data. One field (key) having some errors may lead to that some duplicate records cannot be discovered. However, in such records, the probability of error appearing in another filed of the records may indeed not be so large. Thus, the duplicate records missed in one pass would be detected in another pass. So multi-pass increases recall (the percentage of correct duplicate records detected). As the example shown above, if the name in the two records are the same, then a second run with the name field as the principal field will detect them correctly as duplicate records. Theoretically, suppose the probability of duplicate records missed in one pass is p_ω, $0 \leq p_\omega \leq 1$, where ω is the window size, then the probability of duplicate records missed in n independent passes is p_ω^n. So, the correctness for n-passes is $1 - p_\omega^n$, while the correctness for one pass is $1-p_\omega$. Surely, $1-p_\omega^n$ is larger than $1-p_\omega$. For example, if $n = 3$ and $p_\omega = 50\%$, we have $1-p_\omega^n = 1-0.5^3 = 87.5\%$ and $1-p_\omega = 1-0.5 = 50\%$.

The performance result in [HS95] shows that multi-pass SNM can drastically

improve the accuracy of the results of only one run of SNM with varying large windows. Multi-pass SNM can achieve p_c higher than 90%, while SNM generally only gets p_c about 50% to 70%. Particularly, only a small "window size" is needed for the multi-pass SNM to obtain high accuracy, while no individual run with a single key for sorting produces comparable accuracy results with a large window.

One issue in Multi-pass SNM is that it employs transitive closure to increase the number of duplicate records. The transitive closure allows duplicate records to be detected even without being in the same window during an individual window scan. However, the duplicate results obtained may contain errors (false positives), as explained in Section 1.1 that no comparison methods are completely trustworthy, and transitive closure propagates the errors in results. Thus, multi-pass SNM also increases the number of false positives.

Duplication Elimination SNM

Duplicate Elimination SNM (DE-SNM) [Her96] improves SNM by first sorting the records on a chosen key and then dividing the sorted records into two lists: a duplicate list and a non-duplicate list. The duplicate list contains all records with exact duplicate keys. All the other records are put into the non-duplicate list. A small window scan is first performed on the duplicate list to find the lists of matched and unmatched records. The list of unmatched records is merged with the original non-duplicate list and a second window scan is performed. Figure 2-2 shows how DE-SNM works.

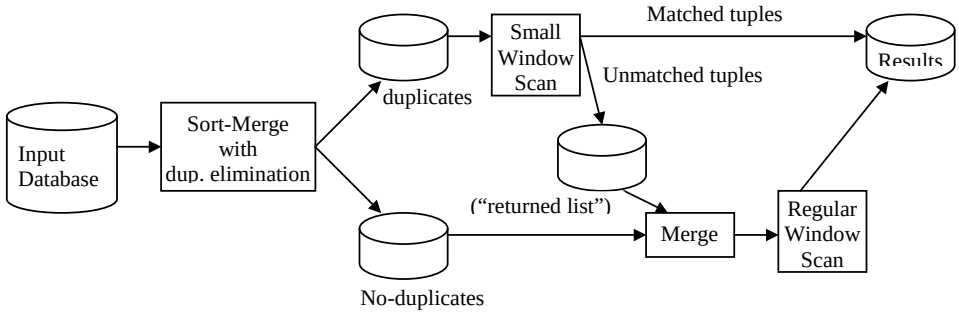

Figure 2-2: Duplication Elimination SNM.

DE-SNM does not contribute much on the improvement of accuracy of SNM. The benefit of DE-SNM is on that it runs faster than SNM under the same window size, especially for the databases that are heavily dirty. If the number of records in duplicate list is large, DE-SNM will run faster than SNM.

Priority Queue Method

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. Transitivity of the "is a duplicate of" relation is equivalent to reachability in the graph. There is a well-known data structure, *union-find data structure* [CLR90, HU73, Tar75], that efficiently solves the problem of determining and maintaining the connected components of undirected graph. 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.

More information on the union-find data structure can be found in [CLR90].

By using the union-find data structure, *Priority Queue method* is suggested in [ME97]. Priority Queue does two passes of sorting and scanning. Two passes are used to increase the accuracy over one pass as the reason is shown in multi-pass SNM. 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 but from right to left. Unlike previous algorithms, the sorting of the records in each pass is domain-independent. Thus the Priority Queue is a domain-independent detection method.

Priority Queue 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, it uses the *Find* operation. 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 [SW81], which is to find the lowest changes (mu-

tations, insertions, or deletions) that converts one string into another. 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. For practical reasons, the priority queue contains only a few number (e.g. 4) of sets of records (like the window size in SNM), and the sets in the priority queue represent the last few clusters detected.

Priority Queue using the union-find data structure to compute the transitive closure online, which may result in saving a lot of unnecessary comparisons. For example, for three duplicate records A_1, A_2 and A_3, there are three comparisons in SNM. However, in Priority Queue, if A_1 and A_2 have been compared and *Union*ed in a cluster, in which A_1 is the representative, then when A_3 is scanned, it only needs to compare with A_1 and one comparison is saved. Note that if the database is clean or slightly dirty, then each cluster in the priority queue most likely contains only one record (singleton set). Under this conditions, the Priority Queue is just the same as the Multi-pass SNM (2 passes) but with extra cost on the *Union* and *Find* operations. Thus for clean or slightly dirty databases, Priority Queue does not have any help, or even worse it takes more time due to the extra *Union* and *Find* operations before each comparison. However, surely, Priority Queue works better for heavily databases since clusters likely contain more than one record.

In Priority Queue, the size of the priority queue should be determined. Thus it still faces the same "window size" problem as SNM does. Further, as Priority Queue computes transitive closure online, it faces the transitive closure problem (discussed in Multi-pass SNM) as well. Moreover, representative records are chosen for each cluster and heuristics need to be developed for choosing the representative records, which will affect the results greatly.

We have introduced some detection methods and shown that each has its own tradeoff. Due to that pair-wisely comparing every record with every other record is infeasible for large databases, SNM is firstly proposed by providing an approximate solution. SNM includes three phases: Create Key, Sort Data, and Merge. The "Sorting" performs the first clustering on the database such that the similar records are close together. Then the "merging" performs clustering again on the sorted database to obtain the clustering result such that the records in each cluster represent the same entity and the records in different clusters represent different entities. The sorting and merging together is two-level clustering that the sorting is the first loose clustering, while the merging is the second strict clustering. In sorting, only the key value (normally one field) need to be compared, while in merging, all fields should be considered.

Clusterings (sorting and merging) are used to significantly reduce the (detection scope and comparison) time with achieving a reasonable accuracy. SNM generally cannot obtain high accuracy and also works for any database coherently. Other

approximate methods are further proposed to improve the performance on either efficiency or accuracy. Multi-pass SNM can largely increase the accuracy under the same time than SNM does. Since in Priority Queue, duplicate records are likely grouped into a set, and new records are compared only with the representative of the set, thus Priority Queue can save some unnecessary comparisons taken by SNM by computing the transitive closure online. Priority Queue may be faster than SNM but cannot improve the accuracy under the same conditions with SNM. In addition, the performance of Priority Queue depends on the properties of databases. For clean and slightly dirty databases, Priority Queue does not have any help for prevailing singleton sets. But for dirty databases, Priority Queue is much faster. The more dirty the database is, the more time it can save. Like Priority Queue, DE-SNM can also run faster than SNM for dirty databases, but DE-SNM will decrease the accuracy. Clustering SNM is an alternative method. As the name shows, Clustering SNM does one even looser clustering before applying SNM. The clustering SNM does three level clustering from looser to stricter. Clustering SNM is faster than SNM for very large databases but it may decrease the accuracy as well. Further, Clustering SNM is suitable for parallel implementation.

Given the trade-off of each method, a natural question is, under certain conditions, which method should be used. Table 2.2 gives some suggestions. Practically, among all these methods, multi-pass SNM is the most popular. Some data cleaner systems, such as IntelliClean [LLL00], DataCleanser DataBlade Module [Mod] etc., employ it as their underlying detection systems.

Condition and requirement	Suggestion
The database is quite small, or it is large but long time to execute is acceptable	Pair-wise comparisons
The database is very large, less false positives are more important than more correctness, and multiple processors are available	Clustering SNM
More correctness would be better, and some false positives are acceptable	Multi-pass SNM
The database is heavily dirty, and some false positives are acceptable	Priority Queue

Table 2.2: The methods would be used for different conditions.

2.3 Comparison Methods

As the detection methods determine which records need to be compared, pare-wise comparison methods are then to decide whether two records compared are duplicate. As we have indicated in Section 1.1 that the comparison methods can be distinguished as two different approaches, namely rule-based and similarity-based. The rule-based approach is using production rules based on domain-specific knowledge, and the similarity-based approach is by computing the degree of similarity of records, which is a value between 0.0 and 1.0.

Notice that the comparison of records is quite complicated, it needs to take more

information into consideration than the sorting does in the detection methods. Thus, the cost of comparisons is the dominate of the time taken by cleansing, which is proven by the performance studies in [HS95]. This further shows that the importance on avoiding unnecessary calls to the record comparison function by the detection system.

Further, all comparison methods (either rule-based or similarity-based) are only approximate methods. That is, none of them can guarantee to discover exactly correct result, which means that, given two duplicate records, the comparison method may not detect them as duplicate, or given two non-duplicate records, the comparison method may detect them as duplicate. The reason is that whether two records are duplicate is a semantical problem, but the solution to it is syntactical based.

2.3.1 Rule-based Methods

The rule-based approach uses a declarative rule language to specify the rules. A rule is generally of the form:

```
if <condition>
then <action>
```

The action part of the rule will be activated when the conditions are satisfied. Complex predicates and external function references may be contained in both the condition and action parts of the rule. The rules are derived naturally from the business domain. The business analyst with subject matter knowledge is able

```
Given two records, r1 and r2.
IF the last name of r1 equals the last name of r2,
   AND the first names differ slightly,
   AND the address of r1 equals the address of r2
THEN
   r1 is equivalent to r2.
```

Figure 2-3: A simplified rule of equational theory.

to fully understand the governing business logic and can develop the appropriate conditions and actions.

Equational Theory was proposed in [HS95] to compare records. Figure 2-3 presents a simplified rule that describes one axiom of equational theory. The implementation of "differ slightly" specified there is based upon the computation of a distance function applied to the first name fields of two records, and the comparison of its results to a threshold to capture obvious typographical errors that may occur in the data. The selection of a distance function and proper threshold is a knowledge intensive activity that demands experimental evaluation. An improperly chosen threshold will lead to either an increase in the number of false positives or to a decrease in the number of correct duplicate records.

In [HS95], rules are written in OPS5 [For81]. In [LLL01], rules are written in the Java Expert System Shell (JESS) [FH99]. JESS is a rule engine and scripting environment written in Sun's Java language and was inspired by the CLIPS [Ril02]

```
INPUT RECORDS: A, B

IF

    (A.currency == B.currency) AND

    (A.telephone == B.telephone AND

    (A.telephone != EMPTY_STRING) AND

    (SUBSTRING_ANY(A.code, B.code) == TRUE) AND

    (FIELDSIMILARITY(A.address, B.address) > 0.85)

THEN

    DUPLICATES(A, B)
```

Figure 2-4: A simplified rule written in JESS engine.

expert system shell. The data cleansing rules are represented as declarative rules in the JESS engine. Figure 2-4 shows one such rule (in pseudocode) written in JESS engine. For the rule to be activated, the corresponding currencies and telephone numbers must match. Telephone numbers must also not be empty, and one of the codes must be a substring of the other. The address must also be very similar.

The effectiveness of the rule-based comparison method is highly dependent on the rules developed. As well-developed rules are effective in identifying true duplicates and also strict enough to keep out false positives, the not well-developed rules will introduce even worse results. Therefore, the rules should be carefully developed and generally are tested repeatedly for the particular domain. As a result, the process of creating such (well-developed) rules can be time consuming.

Further, the rules must be continually updated whenever new data is added to the database that does not follow the patterns by which the rules were originally created. Moreover, the rule-based comparison methods are quite slow and do not clearly scale up for very large datasets. For example, in the experimental study in [HS95], all the rules are first written in OPS5 and then translated by hand into C since the OPS5 compiler is too slow.

To avoid these disadvantages exist in rule-based approach, similarity approach is an alternative. Although the similarity-based methods can resolve the disadvantages in rule-based method, they have their own disadvantages that we will show later. In the following, we discuss and analyze the similarity-based methods in details.

2.3.2 Similarity-based Methods

Similarity-based approach is to compute the degree of similarity for records by a similarity function Sim defined in Section 1.1, which returns a value between 0.0 and 1.0. Two records having large Sim value means that they are very similar. In the special values, 0.0 means absolute non-equivalence and 1.0 means absolute equivalence. Notice that the definition of Sim can be applied on any data type, such as strings and images etc. Therefore, how to view the content of records is important for the definition of similarity function. Of course, the similarity function for two strings is definitely different with the similarity function for two images.

Due to the string data is ubiquitous, currently the discussion is focused on this type of data. In the following discussion, we can assume that each field only contains string value. Thus, record comparison is basically an string matching algorithm and any of the approximate string matching algorithms [BM77, CL92, DC94, GG88, HD80, KJP77] can be used in place of the record comparison method in the detection system.

Edit Distance

Edit Distance [HD80, WF74] is a classic method in comparing two strings that has received much attention and has applications in many fields. It can also be employed in data cleansing and is a useful measure for similarity of two strings. Edit distance is defined as the minimum number of insertions, deletions, and substitutions needed to transform one string into another. For example, the edit distance between "intention" and "execution" is five. Figure 2-5 shows the operations taken by transforming "intention" to "execution". Edit distance is typically implemented using dynamic programming [Gus97], and run in $O(mn)$ time where m and n are the lengths of the two strings. Figure 2-6 shows the dynamic programming to compute the Edit Distance.

For two strings, Edit Distance returns an integer value. However, the lengths of the strings compared need to be taken into account. Although two strings of length 10 differing by 1 character have the same edit distance as two strings of length 2 differing by 1 character, we would most likely state that only the length

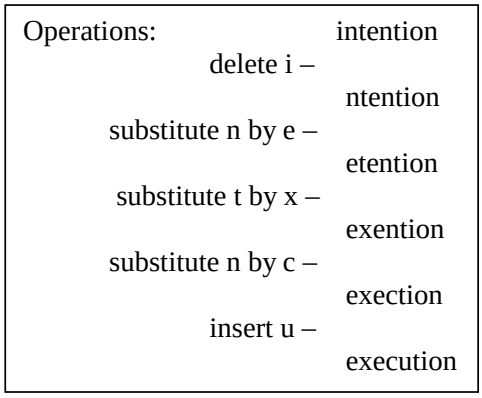

Figure 2-5: The operations taken by transforming "intention" to "execution".

10 strings are "almost equal". Thus edit distance need to be normalized. *Post-normalization* by the maximal length of the compared strings is quite popular. Since the edit distance of any two strings is between 0 and the maximal length of them, the post-normalization returns a value between 0.0 and 1.0. This value can be easily transfered to similarity as the difference of one and the normalized value. A variant normalization method, called *normalized edit distance* (NED), is proposed in [MV93, VMA95]. The NED between two strings is defined as the quotient between the number of the edit operations required to transform one into the other and the length of the editing path corresponding to these operations. NED is computed in $O(mn^2)$ time, where m and n are the lengths of the two strings, and $n \leq m$.

The post-normalized edit distance is to compare two strings instead of two sets of strings, thus it works on the field level instead of the record level (each field is a string, and each record is a set of strings). To make it work on the record

```
int edit(char* x, char* y) \* computation of edit distance *\
{
   int m = strlen(x), n = strlen(y);
   int EDIT[m][n], delta;
   for (i = 1; i <= m; i++) EDIT[i,0] = i;
   for (j = 1; j <= n; j++) EDIT[0,j] = j;
   for (i = 1; i <= m; i++) {
      for (j = 1; j <= n; j++) {
         if (x[i] == y[j]) delta = 0;
         else delta = 1;
         EDIT[i,j] = min(EDIT[i-1,j]+1, EDIT[i,j-1]+1,
                              EDIT[i-1,j-1]+delta);
      }
   }
   return EDIT[m,n]
}
```

Figure 2-6: The dynamic programming to compute edit distance.

level, in [ME97], each record is viewed just as one big string, and *Smith-Waterman algorithm* [SW81] is employed to compare records. The Smith-Waterman algorithm is a variant of edit distance and was originally developed for finding evolutionary relationships between biological protein and DNA sequences. Record Similarity, which is discussed as follows, uses another solution, which is to assign weightages for fields and then the similarity of records equals to the sum of the similarity of field times the field weightage. More information can be found later.

Longest Common Subsequence

The *longest common subsequence* [AG87, Hir77, Lar] is another classic method in comparing two strings. A *subsequence* [dic] of a given string is a string that can be obtained by deleting zero or more symbols from the given string. The Longest Common Subsequence is to find the maximum length of a common subsequence of two strings. We will show the formal description of Longest Common Subsequence in Section 3.3. The Longest Common Subsequence is closely related to the edit distance and it is also implemented using the dynamic programming (see Figure 2-7) with time complexity $O(nm)$ where m and n are the lengths of the two strings.

Longest Common Substring

The *Longest Common Substring* [dic] is to find the maximum length of a common substring of two strings. The longest Common Substring is different with the longest common subsequence. A substring is contiguous, while a subsequence need not be.

```
int lcs_length(char* A, char* B)
{
    int m = strlen(A), n = strlen(B);
    int L[m][n];
    for (i = m; i >= 0; i--) {
        for (j = n; j >= 0; j--) {
            if (A[i] == '\0' || B[j] == '\0') L[i,j] = 0;
            else if (A[i] == B[j]) L[i,j] = 1 + L[i+1, j+1];
            else L[i,j] = max(L[i+1, j], L[i, j+1]);
        }
    }
    return L[0,0];
}
```

Figure 2-7: The dynamic programming to compute Longest Common Subsequence.

The Longest Common Substring problem can be solved by using a data structure known as the suffix tree [McC76, Ukk92, Ukk95, Wei73] in $O(m + n)$ time where m and n are the lengths of the two strings. The longest common substring itself is not enough to describe the similarity of two strings. But it can be extended as a substring-based method [QSLS03].

Record Similarity

Record Similarity (RS) was introduced in [LLLK99]. In Record Similarity, record equivalence can be determined by viewing records at three levels: token, field and record. Since each field has different importance, field weightage is introduced on all the fields. The field weightage is decided through experimental tests and the sum of all field weightages equals to 1. The string value in each field is parsed as tokens by using a set of delimiters such as space and punctuations. Tokens can be viewed as meaningful components. For example, suppose the delimiter is space, then the string "Li Zhao" has tokens {"Li", "Zhao"}.

The process of computing the similarity between two records begins with comparing the sorted tokens of the corresponding fields. The following shows the details in computing the three levels similarities.

(1) Compute Token Similarity:

- If two tokens t_1 and t_2 are exactly matched, then their degree of similarity, $DoS_{(t_1,t_2)}$, is 1;
- Otherwise, if there is a total of x characters in the token t_1, then we deduct $1/x$ from the maximum degree of similarity of 1 for each character that is not found in the other token t_2.

In this definition, the similarity of tokens is not symmetric. That is, $DoS_{(t_i,t_j)} \neq DoS_{(t_j,t_i)}$. For example, if two tokens "cat" and "late" are compared, then the degree of similarity of comparing "cat" with "late", $DoS_{(cat,late)}$ = 1-1/3 = 0.67

since the character c in "cat" is not found in "late", and $DoS_{(late,cat)}$ = 1-2/4 = 0.5 since the characters l and e are not found in "cat".

(2) Compute Field Similarity:

- Suppose a field F in record X has tokens x_1, x_2, $\cdots$, x_n, and the field F in record Y has tokens y_1, y_2, $\cdots$, y_m;
- Each token x_i, $1 \leq i \leq n$, is compared with all the tokens y_j, $1 \leq j \leq m$;
- Let DoS_{x_1} be the maximum of the degree of similarities computed for tokens x_1 with y_1, y_2, $\cdots$, y_m respectively. That is,

  $DoS_{x_1} = max\{DoS_{(x_1,y_1)}, DoS_{(x_1,y_2)}, \cdots, DoS_{(x_1,y_m)}\}$.

  Similarly, we have DoS_{x_2},$\cdots$, DoS_{x_n}, DoS_{y_1}, DoS_{y_2},$\cdots$, DoS_{y_m};
- Field similarity for records X and Y on this field F is given by

$$Sim_F^{RS}(X,Y) = \frac{\sum_{i=1}^{n} DoS_{x_i} + \sum_{j=1}^{m} DoS_{y_j}}{n+m} \tag{2.1}$$

(3) Compute Record Similarity:

- Suppose the records have r fields, F_1, F_2, $\cdots$, F_r, and the field weightages are W_1, W_2, $\cdots$, W_r respectively, where $\sum_{i=1}^{r} W_i = 1$;
- Record Similarity for records X and Y is given by

$$Sim^{RS}(X,Y) = \sum_{i=1}^{r} (Sim_{F_i}^{RS}(X,Y) \times W_i) \tag{2.2}$$

Record	Field
X	ab ex
Y	ex ex ex ex ex ex ex ex ex ex

Table 2.3: Tokens repeat problem in Record Similarity.

We use superscript RS here to distinguish other similarity methods. If RS is understood, we then drop RS for simplicity.

Although the token similarity is not symmetric, the field similarity and the record similarity are symmetric. That is, for any field F and two records X and Y, we have $Sim_F(X, Y) = Sim_F(Y, X)$ and $Sim(X, Y) = Sim(Y, X)$. Two records are treated as a duplicate pair if their record similarity exceeds a certain threshold such as 0.8.

Notice that the method for computing the similarity of records is extensible. The same method can be used for transferring any field level similarity to record level similarity. That is, for any field similarity method Sim_F, such as Edit Distance, it can be transfered to similarity of records as $Sim = \sum_i (Sim_{F_i} \times W_i)$.

Record Similarity is employed in IntelliClean and generally shows good performance. However, theoretically, Record Similarity is not well defined. It surely guarantees that similar records have large value. However, it may also assign some (very) dissimilar records with large value. Especially, it does not address the repeated tokens well. For example, the two records (for simplicity, suppose that they have only one field) in Table 2.3 are very dissimilar, but the Record Similarity com-

puted for them is really large. Suppose the delimiter is space. For record X, the token similarities for "ab" and "ex" are $DoS_{ab} = 0$ and $DoS_{ex} = 1$ respectively. For record Y, there are 10 repeated tokens "ex" and each has $DoS_{ex} = 1$. Thus, $Sim(X, Y) = \frac{1+10}{2+10} = 0.92$, which is a quite large value, with which a false positive may be introduced.

Moving Contracting Window Patter Algorithm

In [QSLS03], the *Moving Contracting Window Patter Algorithm (MCWPA)* was proposed. The MCWPA is focused on the field level. Unlike Record Similarity which is a token base similarity method, MCWPA is a substring based method.

All characters as a whole within the window (string) constitute a *window pattern*. As an example, for the string "abcde", when the window is sliding from left to right with the window size being 3, the series of window patterns obtained are "abc", "bcd" and "cde". Notice that "window size" used here is different with that used in detection methods (e.g., SNM).

Given a Field F and two records X and Y, Let X_F denote the string value of field F of record X. Suppose that X_F and Y_F have m and n characters respectively (including blank space or comma). The field similarity for F of X and Y is given as

$$Sim_F^{MCWPA}(X, Y) = \frac{2\sqrt{SSNC}}{m + n}$$

The algorithm to calculate $SSNC$ is shown in Figure 2-8. $SSNC$ represents the Sum of the Square of the Number of the same characters between X_F and Y_F.

1. Assume $n \leq m$, that is $|X_F| \leq |Y_F|$; 2. $w = n$;
3. $SSNC = 0$;
4. window is placed on the leftmost position X_F;
5. while ((w != 0) and (still some characters in X_F are accessible))
6. {
7. while (window right border does not exceed the right border of the X_F)
8. {
9. if (the window pattern in X_F has the same pattern anywhere in Y_F)
10. {
11. $SSNC = SSNC + w^2$;
12. mark the pattern characters in X_F and Y_F as inaccessible characters to avoid revisiting;
13. }
14. move window rightward by 1 (if the window left border is on an inaccessible character, move window rightward by 2 and so on)
15. }
16. $w = w - 1$;
17. window is placed on the leftmost position where the window left border is on an accessible character;
18. }
19. return $SSNC$;

Figure 2-8: Calculate $SSNC$ in MCWPA algorithm.

$Sim_F(X, Y)$ reflects the ratio of the total number of the common characters in two fields to the total number of characters in two fields.

MCWPA does not have the repeated tokens problem existing in Record Similarity. For the two records in Table 2.3, since they only have a same length-three substring " ex", then the similarity computed by MCWPA for them is $Sim_F^{MCWPA}(X, Y) = \frac{2\sqrt{3^2}}{5+29} = \frac{6}{34} = 0.18$, while the similarity computed by Records Similarity is $Sim_F^{RS}(X, Y) = 0.92$. So MCWPA is more accurate to describe the two records in Table 2.3 and will not obtain them as duplicate.

As we can see, the similarity approach is simply to compute a similarity value, and it has the following advantages:

- *Easy implementation:* It can be easily implemented and embedded in any data cleansing system.
- *Uniform and stable:* For different databases or updated databases, only the field weightages and threshold need to be reset. Since the field weightages and threshold can be implemented as input parameters, the method itself does not need to be modified.
- *Fast and scalable:* It is much faster than rule-based approach and scales well for large databases.

However, it also has disadvantage. Since similarity methods only return a similarity value, they are hard to achieve the same accuracy as the well-developed

rules does. For example, for two non-duplicate records that are very similar, the similarity methods will return a value that is larger than the given threshold. Thus the non-duplicate records are detected as duplicate, which is a false-positive.

In general, it is difficult to achieve both in terms of efficiency and accuracy. The rule-based approach and similarity-based approach show the trade-off between time and accuracy.

2.4 Other Works

All the works we introduced above are on the algorithm-level of data cleansing. Other works related to data cleansing include proposing high level languages to express data transformation [Coh98, GFS+01a, GP99, LSS96], and introducing high level data cleansing frameworks [GFS+01a, LLL00, RH01].

In [GFS+01a], the authors presented five logical operators, namely mapping, view, matching, clustering, and merging, for expressing data cleansing transformations. These operators extend the data transformations expressible with SQL99 and can be composed to express all the data transformations from data cleansing in the research literature. Having a SQL-like language extension has the advantage of increased usability if users are familiar with SQL. More information about the five operators can be found there. Specially, a matching operator computes an approximate join between two relations. More specifically, it computes a distance value for each pair of tuples in the Cartesian product of the two input relations using an arbitrary distance function. However, a matching operation can be im-

plemented by different kinds of specialized algorithms and it also investigates the optimization techniques that can be used to optimize the matching process. One type of optimization is to use the multi-pass SNM to limit the number of records compared.

An interactive framework known as Potter's Wheel is proposed in [RH01]. It offers graphical specification of data transformations through a spreadsheet-like interface. The Potter's Wheel allows the user to try various transformations interactively, observe their effects on the data, and undo them if they are inappropriate.

In [LLL00], the authors presented a knowledge-based framework, IntelliClean, for intelligent data cleaning. This framework takes on a systematic approach and provides a complete strategy for standardizing, anomaly detection and removal, and duplicate elimination in dirty databases.

Notice that the algorithms are fundamental for all data cleansing. For instance, in [LLL00], multi-pass SNM is employed as its underlying detection system and Record Similarity is used in its rule based system to determine fields similarity. In [GFS+01a] edit distance is employed as its matching operator, and length filter [GIJ+01] and multi-pass SNM are used as its matching operator optimization. Therefore, the high level languages and data cleansing frameworks will benefit from the performance improvement of the algorithms.

Chapter 3

New Efficient Data Cleansing Methods

3.1 Introduction

The detection methods introduced in Chapter 2 are independent from the comparison methods. That is, any detection method can use any comparison method to compare records. This independence gives freedom for applications but will result in a loss of useful information, which can be used to save expensive comparisons.

Let Y+X denote a method in which Y is a detection method and X is a comparison method used in Y. Consider SNM+RS, if the size of the window is ω, then every new record entering the window is compared with the previous ω - 1 records with RS to find duplicate records. We note that when a new record entering the window, all the records in the window have been compared with each other and

should have achieved some knowledge on these comparisons. SNM+RS doesn't keep and make use of that comparison information and simply ignores it, which results in lots of unnecessary comparisons. For example, for records A, B and C, suppose that we have compared A with B, and B with C. SNM+RS will compare A with C in any case. However, intuitively, if A and B are very similar, and B and C are very similar/dissimilar, then A and C should be similar/dissimilar. If the comparison method can well describe this case such that $Sim(A, C)$, similarity of A and C, can be bounded by $Sim(A, B)$ and $Sim(B, C)$, and the previous two comparisons information ($Sim(A, B)$ and $Sim(B, C)$) are saved and then efficiently used, there would be a chance to know whether A and C are duplicate or not without actually comparing them. Since the comparisons are very expensive, reducing the number of comparisons is therefore important in reducing the whole execution time. Thus the problem now is how to save the comparison information and use this information efficiently.

To solve this problem, in this chapter, we first introduce a new comparison method, LCSS, based on the longest common subsequence. LCSS is efficient in detecting duplicate records and satisfies the following two useful properties:

- *Lower Bound Similarity Property (LP):*

  $Sim(A, C) \geq Sim(A, B) + Sim(B, C) - 1$

- *Upper Bound Similarity Property (UP):*

  $Sim(A, C) \leq 1 - |Sim(A, B) - Sim(B, C)|$

The proof for LCSS satisfying the LP and UP is given in Theorem 3.7. With

the above properties, for any three records A, B and C, when $Sim(A, B)$ and $Sim(B, C)$ are known, we can evaluate the lower bound and upper bound for $Sim(A, C)$ without actually comparing records A and C. Thus, these two properties provide a way to use previous comparison information (as similarity). To save that information, we set the *Anchor Record* in detection method, which is a particular record that keeps previous comparison information as a list of similarities. Detail description for how to choose anchor record is given in Section 3.4.2. The anchor record serves as an anchor (as the record B in LP and UP) for future comparisons.

Based on the properties and anchor record, we then propose two new methods, RAR1 and RAR2. They vary on the number of anchor records and their locations. RAR1 has one anchor record and it is in the window. RAR2 has two anchor records, one in the window and the other one outside the window.

The rest of the chapter is organized as follows. In the next section, we discuss the properties for similarity methods. In Section 3.3, we introduce the comparison method LCSS, and show that it satisfies the properties. In Section 3.4, we propose two new methods: RAR1 and RAR2. In Section 3.5, we discuss the transitive closure problem and introduce conditional transitive closure. We give the performance results in Section 3.6 and summary in Section 3.7.

3.2 Properties of Similarity

Intuitively, for any three records A, B and C, if A and B are very similar, and B and C are also very similar, then A and C should be similar. On the other hand, if A and B are very similar/dissimilar, and B and C are very dissimilar/similar, then A and C should be dissimilar.

Since *similarity* is used to describe the degree of similarity of records, similar records should have large similarity and dissimilar records should have small similarity. Thus a *well-defined similarity method* should be able to describe the above intuitive cases. We describe the intuition in similarity function, Sim, as follows.

1. If $Sim(A, B)$ and $Sim(B, C)$ are large, $Sim(A, C)$ should be relatively large.

2. If $Sim(A, B)$ is large/small and $Sim(B, C)$ is small/large, $Sim(A, C)$ should be relatively small.

Given two records A and B, with similarity $Sim(A, B)$, let $d(A, B) = 1 - Sim(A, B)$. If the measure d satisfies the triangle inequality, then for any records A, B and C, we have

- $d(A, C) \leq d(A, B) + d(B, C)$

  $\Leftrightarrow 1 - Sim(A, C) \leq 1 - Sim(A, B) + 1 - Sim(B, C)$

  $\Leftrightarrow Sim(A, C) \geq Sim(A, B) + Sim(B, C) - 1$

- $d(A, C) \geq d(A, B) - d(B, C)$

  $\Leftrightarrow 1 - Sim(A, C) \geq 1 - Sim(A, B) - (1 - Sim(B, C))$

  $\Leftrightarrow Sim(A, C) \leq 1 - (Sim(B, C) - Sim(A, B))$

Similarly, we can get

$Sim(A, C) \leq 1 - (Sim(A, B) - Sim(B, C))$.

Thus we have

$Sim(A, C) \leq 1 - |Sim(A, B) - Sim(B, C)|$.

Let

$$L_B(A, C) = Sim(A, B) + Sim(B, C) - 1, \text{ and}$$

$$U_B(A, C) = 1 - |Sim(A, B) - Sim(B, C)|.$$

We say a similarity method has *triangle inequality property* if for any three records A, B and C, we have

- *Lower Bound Similarity Property (LP):*

  $Sim(A, C) \geq L_B(A, C)$

- *Upper Bound Similarity Property (UP):*

  $Sim(A, C) \leq U_B(A, C)$

For brevity, we write L_B and U_B as L and U respectively if B is understood. The $L(A, C)$ and $U(A, C)$ are used to calculate the worst possible similarity and the best possible similarity between A and C respectively. They are calculated from the values $Sim(A, B)$ and $Sim(B, C)$ instead of from the comparison of A and C. One natural idea is that if there is a comparison method that satisfies the above two properties, we can employ them into the detection method, thus reducing the number of comparisons and saving execution time. In the next section, we propose an efficient comparison method that satisfies the two properties. Then in Section 3.4, we show how to employ the two properties efficiently.

3.3 LCSS

3.3.1 Longest Common Subsequence

The *Longest Common Subsequence* has been studied extensively and has many applications, such as text pattern matching, speech recognition etc. Here, we formally describe it and follow the notation used in [Lar].

Definition 3.1 *A* subsequence *of a string s is any string, which can be created from s by deleting some of the elements. Formally, if s is the string $s_1 s_2 \cdots s_k$ then $s_{i_1} s_{i_2} \cdots s_{i_p}$ is a subsequence of s if $\forall j \in \{1, \ldots, p\} : i_j \in \{1, \ldots k\}$ and $\forall j \in \{1, \ldots, p-1\} : i_j < i_{j+1}$.*

Definition 3.2 *The* longest common subsequences *of two strings $s = s_1 s_2 \cdots s_k$ and $t = t_1 t_2 \cdots t_m$ are the subsequences of both s and t with maximal length.*

The longest common subsequences of two strings are not unique but the length of all longest common subsequences are the same. We use $lcs(s, t)$ to denote the (unique) length of the longest common subsequences of strings s and t.

Lemma 3.3 *For any three strings x, y and z, we have*

$lcs(x, y) + lcs(y, z) \leq |y| + lcs(x, z)$.

Proof: Let S_{xy} be the longest common subsequence of strings x and y. Similar for S_{xz} and S_{yz}. Let $y' = y - S_{xy}$, that is, y' is the subsequence of y with removing S_{xy} from it. Hence, $|y'| = |y| - |S_{xy}|$. Thus,

$$lcs(x, y) + lcs(y, z) \leq |y| + lcs(x, z)$$

$$\Leftrightarrow |S_{xy}| + |S_{yz}| \leq |y| + |S_{xz}|$$

$$\Leftrightarrow |S_{yz}| \leq |y| - |S_{xy}| + |S_{xz}|$$

$$\Leftrightarrow |S_{yz}| \leq |y'| + |S_{xz}|.$$

Since $y' = y - S_{xy}$, we have $|S_{yz}| \leq |S_{y'z}| + |S_{S_{xy}z}|$. Since y' is a subsequence of y and S_{xy} is a subsequence of x, we then have $|S_{y'z}| \leq |y'|$ and $|S_{S_{xy}z}| \leq |S_{xz}|$. Hence $|S_{yz}| \leq |y'| + |S_{xz}|$.

Thus the theorem is proved. ∎

To get a similarity value between 0.0 and 1.0, *post-normalization* by the maximal length of the compared strings is quite popular. Let $lcs_p(s,t)$ denote the post-normalization of s and t, i.e., $lcs_p(s,t) = lcs(s,t)/max\{|s|,|t|\}$. If $|s| \geq |t|$, we have $lcs_p(s,t) = lcs(s,t)/|s|$.

Let $d_p(s,t) = 1 - lcs_p(s,t)$. If $|s| \geq |t|$, we have $d_p(s,t) = 1 - lcs(s,t)/|s|$.

Theorem 3.4 *For any three strings x, y and z, we have*

$d_p(x,z) \leq d_p(x,y) + d_p(y,z)$.

Proof: For simplicity, we write d_p as d in the proof. Without loss of generality, we assume $|x| \geq |z|$. There are only the following three cases for x, y and z:

(1) $|x| \geq |y| \geq |z|$

$$d(x,y) + d(y,z) = 1 - lcs(x,y)/|x| + 1 - lcs(y,z)/|y|$$

$$= 1 - lcs(x,y)/|x| + (|y| - lcs(y,z))/|y|$$

$$\geq 1 - lcs(x,y)/|x| + (|y| - lcs(y,z))/|x|$$

$$= 1 - (lcs(x,y) + lcs(y,z) - |y|)/|x|$$

$$\geq 1 - lcs(x,z)/|x| = d(x,z)$$

(2) $|x| \geq |z| \geq |y|$

$$d(x,y) + d(y,z) = 1 - lcs(x,y)/|x| + 1 - lcs(y,z)/|z|$$

$$= 1 - lcs(x,y)/|x| + (|z| - lcs(y,z))/|z|$$

$$\geq 1 - lcs(x,y)/|x| + (|y| - lcs(y,z))/|x|$$

$$= 1 - (lcs(x,y) + lcs(y,z) - |y|)/|x|$$

$$\geq 1 - lcs(x,z)/|x| = d(x,z)$$

(3) $|y| \geq |x| \geq |z|$

$$d(x,y) + d(y,z) = 1 - lcs(x,y)/|y| + 1 - lcs(y,z)/|y|$$

$$= 1 - (lcs(x,y) + lcs(y,z) - |y|)/|y|$$

$$\geq min\{1, 1 - (lcs(x,y) + lcs(y,z) - |y|)/|x|\}$$

$$\geq min\{1, 1 - lcs(x,z)/|x|\}$$

$$\geq 1 - lcs(x,z)/|x| = d(x,z)$$

Thus the theorem is proved. ■

3.3.2 LCSS and its Properties

Based on the normalized longest common subsequence, in the following, we propose the new comparison method, *LCSS*, and show that it satisfies the similarity properties.

To indicate the relative importance of fields, field weightages are introduced on all fields. The sum of all field weightages equals to 1. Field weightages are also used in RS.

Suppose a database has fields F_1, F_2, $\cdots$, F_n, and the field weightages are W_1, W_2, $\cdots$, W_n respectively, $\sum_{i=1}^{n} W_i = 1$. To compute the similarity of records, we first compute the similarity of the corresponding field.

Given a field F, the field similarity for records A and B is given as:

$$Sim_F^{LCSS}(A, B) = lcs_p(A_F, B_F) \tag{3.1}$$

where A_F and B_F are the strings in the filed F of A and B respectively.

Based on the field similarities, the similarity for records A and B is given as:

$$Sim^{LCSS}(A, B) = \sum_{i=1}^{n} (Sim_{F_i}^{LCSS}(A, B) \times W_i) \tag{3.2}$$

If LCSS is understood, we drop it from Sim^{LCSS} and Sim_F^{LCSS} for brevity. The similarity for any two records is between 0.0 and 1.0. Two records are treated as a duplicate pair if their similarity exceeds a certain threshold, denoted as σ (such as 0.8). The computation of threshold is a knowledge intensive activity and demands experimental evaluation.

Let $d(A, B) = 1 - Sim(A, B)$ and $d_F(A, B) = 1 - Sim_F(A, B)$. From the definitions of Sim_F and Theorem 3.4, we immediately have the following lemma.

Lemma 3.5 *For any field F and any three records A, B and C, we have*

$d_F(A, C) \leq d_F(A, B) + d_F(B, C)$. ∎

Theorem 3.6 *For any three records A, B and C, we have*

$d(A, C) \leq d(A, B) + d(B, C)$.

Proof: $d(A,B) + d(B,C)$

$= 1 - Sim(A,B) + 1 - Sim(B,C)$

$= \sum_{i=1}^{n} W_i - \sum_{i=1}^{n}(Sim_{F_i}(A,B) \times W_i) + \sum_{i=1}^{n} W_i - \sum_{i=1}^{n}(Sim_{F_i}(B,C) \times W_i)$

$= \sum_{i=1}^{n}((1 - Sim_{F_i}(A,B)) \times W_i) + \sum_{i=1}^{n}((1 - Sim_{F_i}(B,C)) \times W_i)$

$= \sum_{i=1}^{n}((d_{F_i}(A,B) + d_{F_i}(B,C)) \times W_i)$

$\geq \sum_{i=1}^{n}(d_{F_i}(A,C) \times W_i)$ by Lemma 3.5

$= \sum_{i=1}^{n}((1 - Sim_{F_i}(A,C)) \times W_i)$

$= \sum_{i=1}^{n} W_i - \sum_{i=1}^{n}(Sim_{F_i}(A,C) \times W_i)$

$= 1 - Sim(A,C)$

$= d(A,B)$ ∎

As shown in Section 3.2, LP and UP are derived from the triangle inequality of the measure d. Thus we have the following theorem immediately.

Theorem 3.7 *LCSS satisfies the properties LP and UP.* ∎

3.4 New Detection Methods

In this section, we propose two new detection methods, RAR1 and RAR2, by efficiently employing the properties LP and UP that the comparison method LCSS satisfies. Notice that existing detection methods, such as SNM, do not provide a way to use the properties.

We say that a record is an *anchor record* if the record has a list of similarities with all the other records in the current window. If the anchor record is in the

current window, we call it an *inAnchor* record. Otherwise, we call it an *outAnchor* record.

In this chapter, when we refer to the similarity of records, we always mean that the similarity is computed by LCSS.

3.4.1 Duplicate Rules

Suppose that the similarity threshold is σ, $0 \leq \sigma \leq 1$. Given a record B, for any two records A and C, we have

- *Duplicate Rule (D-rule):*

  If $L_B(A, C) \geq \sigma$, records A and C are duplicate.

- *Non-Duplicate Rule (ND-rule):*

  If $U_B(A, C) < \sigma$, records A and C are not duplicate.

The correctness of D-rule and ND-rule is from Theorem 3.7. Easily, given the similarity threshold σ, since LCSS satisfies the properties LP and UP, if $L(A, C) \geq \sigma$, by LP, we then have $Sim(A, C) \geq \sigma$, i.e., records A and C are detected as duplicate. Thus, D-rule is correct. Similarly, ND-rule is correct by UP.

D-rule/ND-rule can determine whether two records are duplicate/non-duplicate without actually comparing them. The following example (Example 3.1) shows one case that the two rules can save comparisons.

Example 3.1 *Suppose* $\sigma = 0.8$. *For three records A, B and C, if* $Sim(A, B) = 0.9$, *and* $Sim(B, C) = 0.9$, *then* $L(A, C) = 0.9 + 0.9 - 1 = 0.8$. *From D-rule, we*

know that records A and C are detected as a duplicate pair. If $Sim(A, B) = 0.9$, and $Sim(B, C) = 0.5$, then $U(A, C) = 1 + 0.5 - 0.9 = 0.6$. From ND-rule, we have that records A and C are not treated as a duplicate pair.

The following theorem shows that the two duplicate rules are consistent.

Theorem 3.8 *For any three records A, B and C, they cannot satisfy both D-rule and ND-rule.*

Proof: On the contrary we assume that there are three records A, B and C, which satisfy both D-rule and ND-rule. Without loss of generality, suppose that $Sim(A, B) \leq Sim(B, C)$, then from D-rule, we have

$$Sim(A, B) + Sim(B, C) - 1 \geq \sigma \tag{3.3}$$

and from ND-rule, we have

$$1 + Sim(A, B) - Sim(B, C) < \sigma \tag{3.4}$$

From Equations (3.3) and (3.4), we get

$$Sim(A, B) + Sim(B, C) - 1 > 1 + Sim(A, B) - Sim(B, C)$$

$$\Leftrightarrow Sim(B, C) > 1,$$

which is a contradiction with the fact that the similarity of any two records is less than or equal to 1. ∎

Corollary 3.9 *For any three records A, B and C, there are only three possibilities for them: 1) they satisfy D-rule but do not satisfy ND-rule, or 2) they satisfy ND-rule but do not satisfy D-rule, or 3) they do not satisfy D-rule and ND-rule.*

3.4.2 RAR1

The D-rule and ND-rule show how the properties LP and UP can be employed. Now, we propose the new method: *RAR1 (Reduction using one Anchor Record)*. The anchor record is an inAnchor record.

Like SNM, RAR1 can also be summarized in three phases: Create Key, Sort Data and Merge. The previous two phases are exactly the same with those in SNM. We show the Merge phase, which is divided into two stages, as follows. The algorithm is also shown in Figure 3-1.

1. *Initialization Stage:* Suppose the window size is ω. We first read the front ω records of the sorted dataset into the window and do pair-wise comparisons on the ω records. We then set the last record as the inAnchor record and compute the similarity list for it.

2. *Scan Stage:* This stage (see Figure 3-2) can be divided into two parts: comparison decision and anchor record choosing.

   (a) *Comparison decision:* For current window of records, there is an inAnchor record with a similarity list. We denote the inAnchor record as B. When a new record, say C, moves into the window, we first

compare C with B and get the similarity, $Sim(B, C)$. We also add $Sim(B, C)$ into the similarity list. After that, for each record in the current window, say A, with the $Sim(A, B)$ in the similarity list of B, if $L_B(A, C) = Sim(A, B) + Sim(B, C) - 1 \geq \sigma$, we treat C and A as duplicate and no comparison on them is required (D-rule). Else if $U_B(A, C) = 1 - |Sim(A, B) - Sim(B, C)| < \sigma$, we treat C and A as non-duplicate and no comparison on them is require either (ND-rule). Otherwise, we compare them directly to determine whether they are duplicate or not.

(b) *Anchor record choosing:* When the first record in the window slides out of the window and it is not the inAnchor record, we simply remove its similarity from the similarity list, since it is out-of-date. If the first record is the inAnchor record, with sliding it out of the window we choose the last record in the window as the new inAnchor record. We also compare the last record with all the other records in the current window and compute the similarity list for the new inAnchor record.

The main difference between SNM and RAR1 is as follows. While SNM compares the new record entering the current window with all previous records in the window, RAR1 will first check whether the new record with previous records are duplicate or not with D-rule and ND-rule. For those records having been determined as duplicate or non-duplicate with the new record by these two rules, there is no need to compare them with the new record. Thus, with D-rule and ND-rule,

Algorithm: merge phase of RAR1.

Input: Sorted database $\mathcal{D}$, window size ω, similarity threshold σ

Output: Duplicate records

Procedure:

1. Move the first ω records of $\mathcal{D}$ into the window, and pair-wise compare them;
2. Set the anchor record, denoted by B, as the last record in the window;
3. Set the SimilarityList, which keeps a list of similarities of B with the other records in the window;

4. **while** (not at the end of $\mathcal{D}$)
5. {
6. Slide the first record F in the window out of the window;
7. Read the next record C in $\mathcal{D}$ into the window;

8. **if** (B != F) //the anchor record isn't the sliding record
9. {
10. Remove the first similarity from the SimilarityList;
11. Compare C with B and get the similarity: $Sim(B, C)$;
12. **for** (each record A in the window with $Sim(A, B)$ in the SimilarityList)
13. {
14. $L_B(A, C) = Sim(A, B) + Sim(B, C) - 1$;
15. $U_B(A, C) = 1 - |Sim(A, B) - Sim(B, C)|$;

Continuing ...

Continued

```
16.          if (L_B(A, C) ≥ σ)
17.              records A and C are duplicate; //D-rule
18.          else if (U_B(A, C) < σ)
19.              records A and C are non-duplicate; //ND-rule
20.          else
21.               compare records A with C to decide whether they are duplicate;
22.      }
23.      Add the Sim(B, C) into the tail of SimilarityList;
24.  }
25.  else //the anchor record sliding out of the window
26.  {
27.      Set the anchor record B as C;
28.      Empty the SimilarityList;
29.      Compare B with all the other records in the window to detect duplicate
         records, and add the similarities into the SimilarityList as the order in
         the window;
30.  }
31. }
```

Figure 3-1: The algorithm of merge phase of RAR1.

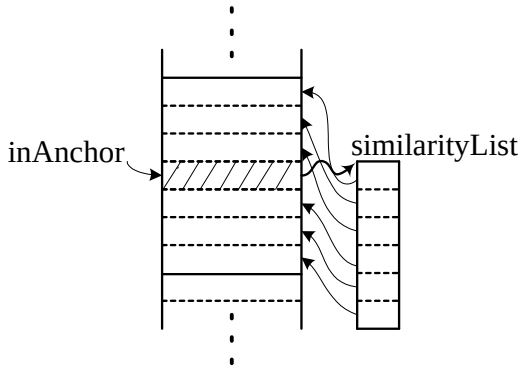

Figure 3-2: The merge phase of RAR1.

RAR1 will reduce a lot of record comparisons. Notice that we only use D-rule and ND-rule in RAR1, thus RAR1 can employ any comparison method that satisfies the properties LP and UP, such as LCSS.

Given three records A, B and C, and given $Sim(A, B)$ and $Sim(B, C)$, in SNM, a comparison on record A and record C is always required to determine whether they are duplicate or non-duplicate. However, in RAR1, we can calculate $L(A, C)$ and $U(A, C)$ for records A and C without comparing records A and C. When 1) $L(A, C) \geq \sigma$ or, 2) $U(A, C) < \sigma$, we treat records A and C as duplicate or non-duplicate and there is no comparison on them. In this case, RAR1 takes one less comparison than SNM does.

In the Example 3.2, we show how RAR1 works and how the LP and UP are used in RAR1 to save comparisons.

Example 3.2 *Suppose $\sigma = 0.8$. In Table 3.1, records A, B and C represent the*

same entity and record B has a small type error on the name field. Suppose that the field weightages for Name, Gender and Dept. are 0.5, 0.25 and 0.25 respectively, and that the window size is 5 and record A is chosen as the inAnchor record. Now we consider the following procedure in RAR1.

1. *Record B moves into the window. It is compared with record A and we get* $Sim(A, B) = 0.5 \times 7/8 + 0.25 \times 1 + 0.25 \times 1 = 0.94 > \sigma$. *Thus records A and B are duplicate.*

2. *Record C moves into the window. It is compared with record A and we get* $Sim(A, C) = 1$. *Thus records A and C are duplicate. Then we have* $Sim(B, C) \geq L(B, C) = Sim(A, B) + Sim(A, C) - 1 = 0.94 + 1 - 1 = 0.94 > \sigma$. *Hence, we obtain records B and C as a duplicate pair and no comparison on them is required.*

3. *Record D moves into the window. We get* $Sim(A, D) = 0.5 \times 0 + 0.25 \times 1 + 0.25 \times 1 = 0.5 < \sigma$. *Thus we know that records A and D are non-duplicate. Then we have* $Sim(B, D) \leq U(B, D) = 1 - |Sim(A, B) - Sim(A, D)| = 1 - |0.94 - 0.5| = 0.56 < \sigma$. *Hence, we know that records B and D are non-duplicate and no comparison on them is required. Similarly, we also know that records C and D are non-duplicate and no comparison on them is required either.*

From the above example, for the four records in Table 3.1, under SNM, there are 6 comparisons. However, in RAR1 only 3 comparisons are required. Hence

	Name	Gender	Dept.
A	li zhao	M	CS
B	li zhai	M	CS
C	li zhao	M	CS
D	sun peng	M	CS

Table 3.1: Four records in the same window.

RAR1 requires 3 comparisons less than SNM.

Let $<A, B>$ denote records A and B as a duplicate pair, DR(SNM) and DR(RAR1) be the duplicate results of SNM and RAR1 respectively. We have the following theorem.

Theorem 3.10 *If we run SNM and RAR1 at the same window size, and with LCSS as the comparison method, we have DR(RAR1) = DR(SNM).*

Proof: We first prove DR(RAR1) $\subseteq$ DR(SNM). For any duplicate pair $<A, C>$ in DR(RAR1), it can only be obtained in the following two ways:

1. If $<A, C>$ is obtained with $Sim(A, C) \geq \sigma$, then it is in DR(SNM) since we run SNM and RAR1 at the same window size.

2. If $<A, C>$ is obtained with the D-rule, then there is an anchor record B, such that $L_B(A, C) \geq \sigma$. Since $Sim(A, C) \geq L_B(A, C)$, we have $Sim(A, C) \geq \sigma$. So $<A, C>$ is in DR(SNM).

So DR(RAR1) $\subseteq$ DR(SNM). Now we prove DR(SNM) $\subseteq$ DR(RAR1). For any duplicate pair $<A, C>$ in DR(SNM), we have $Sim(A, C) \geq \sigma$. Then for any record B, $U_B(A, C) \geq Sim(A, C) \geq \sigma$, that is, ND-rule is not satisfied in RAR1. If D-rule is satisfied, we have $<A, C>$ in DR(RAR1). Otherwise, we will compare records A and C and get $Sim(A, C) \geq \sigma$. So $<A, C>$ is also in DR(RAR1). Thus DR(RAR1) $\subseteq$ DR(SNM). ■

3.4.3 RAR2

Now we propose another method: *RAR2 (Reduction using two Anchor Records)*. One anchor record is an inAnchor record and the other is an outAnchor record. The Merge phase is shown in the following.

1. *Initialization Stage:* Suppose the window size is ω. We first read the front ω records of the sorted dataset into the window and do pair-comparisons on the ω records. We then set the last record as the inAnchor record and compute the similarity list for it. Lastly, we set the outAnchor record as NULL and the similarity list for the outAnchor record as empty.

2. *Scan Stage:* Similar to RAR1, this stage (see Figure 3-3) can also be divided into two parts: comparison decision and anchor record choosing.

   (a) *Comparison decision:* For current window of records, there is an inAnchor record and an outAnchor record. We denote the inAnchor record and outAnchor record as IR and OR respectively. Each anchor record

keeps a list of similarity with all the other records in the current window. When a new record, say L, moves into the window, we first compare L with IR. We get the similarities, $Sim(L, IR)$, and add it into the similarity list of IR. If the record OR is not NULL, we also compare L with OR and add $Sim(L, OR)$ into the similarity list of OR. After that, for each record in the current window, say A, we first use the inAnchor record to determine whether L and A are duplicate or non-duplicate like RAR1 does. If it cannot be determined with the inAnchor record and the outAnchor is not NULL, we then try to use the outAnchor record. If it still cannot be determined, we compare the records directly to determine whether they are duplicate or not.

(b) *Anchor record choosing:* When the first record in the window slides out of the window and it is not the inAnchor record, we simply remove the first similarity from the similarity list of IR and OR. If the first record is the inAnchor record, we set it as the outAnchor record and slide it out of the window. The original outAnchor record is discarded. We then choose the last record in the window as the new inAnchor record. We also compare the last record with all the records in the current window and compute the similarity list for the new inAnchor record.

We have shown the two methods, RAR1 and RAR2. The main difference between them is on how many anchor records they have. RAR2 has two anchor records while RAR1 has only one anchor record. In RAR2, when a new record

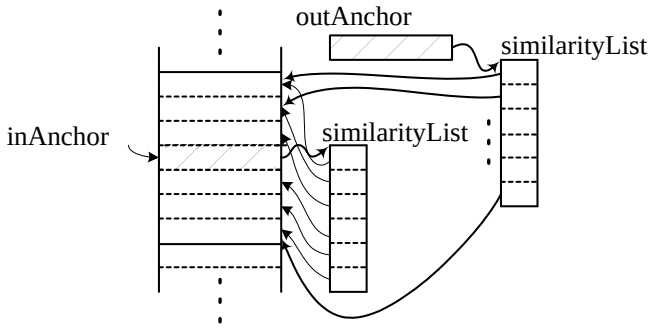

Figure 3-3: The merge phase of RAR2.

entering the current window, it is first compared with the two anchor records, which introduces one more comparison than RAR1 does. However, in RAR2, for each record in the current window, the chance of determining the new record as a duplicate record or not is increased, which produces more chance to reduce comparisons than RAR1 does. The performance result also proves this.

Since the outAnchor is outside the window and the last record in the window will compare with it, RAR2 will obtain a few more duplicate pairs than RAR1 does if both run at the same window size.

Let DR(RAR2) be the duplicate result of RAR2. We have the following theorem immediately.

Theorem 3.11 *If we run RAR1 and RAR2 at the same window size, and with LCSS as the comparison method, we have DR(RAR1)* $\subseteq$ *DR(RAR2).* ∎

As SNM is the core scheme of Clustering SNM, Multi-pass SNM and DE-SNM, similarly, we can propose Clustering RAR1, Multi-pass RAR1, and DE-RAR1 by RAR1, and Clustering RAR2, Multi-pass RAR2, and DE-RAR2 by RAR2.

3.4.4 Alternative Anchor Records Choosing Methods

In RAR1, when the inAnchor record slides out of the current window, the last record in the window (the record just entering the window) is chosen as the new inAnchor record and compared with the other records in the window to set its similarity list. In this section, we discuss alternative methods for choosing the anchor record.

Specifically, we have the following four methods to choose the inAnchor record.

1. *The last record*: choose the last record in the window as the new inAnchor record (the method RAR1 used).

2. *The first record*: choose the first record in the window as the new inAnchor record.

3. *A random record*: choose a record in the window randomly with given distributions, e.g., uniform distribution.

4. *The record with specific properties*: choose the record in the window with some specific properties. One such is to choose the record with the most number of comparisons with other records in the current window. This will be discussed with more details in the following.

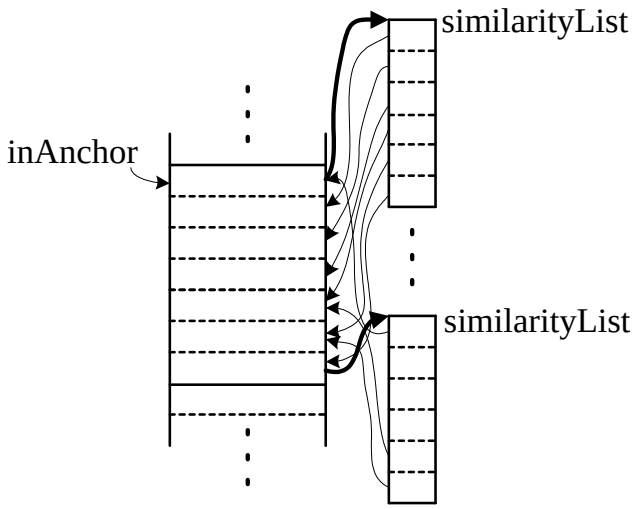

Figure 3-4: The most record method.

However, the second method (the first record) is obviously not a good solution. Since when the first record is chosen as the new inAnchor record, after one turn, it slides out of the window and a new inAnchor record should be chosen again. Thus "the first record" method will result in a lot of inAnchor record updates, which will affect the efficiency largely. Similarly, the third method (a random record) is not a good policy as well.

Consider the fourth method, the record with specific properties. We first explain this method in more details.

In this method, instead of only the anchor record having the similarity list in RAR1, each record in the window has an associated similarity list (see Figure 3-4). Differing with the similarity list of the inAnchor record, these similarity lists of

other records may not be full, i.e., for each record (not the inAnchor record), its list only contains the similarities of the record with other records compared. When the inAnchor record slides out of the window, we choose the record that has the most number of similarities in its list as the new inAnchor record. We call this method "the most record". To avoid updating the inAnchor record too often, user may set a parameter α ($1 \leq \alpha \leq \omega$) to limit the position of the new anchor record in the window, which means that the new inAnchor record must have position in the window larger than α.

The advantage of this method is that it has less comparison than that of "the last record" method to set the similarity list when the inAnchor record is updated. With the last record, we need to set the similarity list by comparing the last record with all the other records in the window. While with the most record, we only need to compare the record with the other records that are not compared previously. Since the record has the most number of similarities, generally, only a few new comparisons are required to set the similarity list.

The disadvantage of this method is that it is more complicated than "the last record" method. Thus it needs more time to implement it. Further, we need to update the similarity lists for all records in the window, while we only need to update the similarity list for the inAnchor record in "the last record" method. Moreover, more updates on the inAnchor record are required.

3.5 Transitive Closure

The computing of transitive closure is an option in some data cleaning methods, but an inherent requirement in some other data cleansing methods. The use of the transitive closure to merge "similar" pieces of information can also be found in the study of fuzzy set [Kre95]. Let $Equal(a, b)$ denote "a is identical to b", where a and b are any two objects. Traditional mathematical description of identity is formalized as follows:

- *Reflexive*: every object is identical to itself, i.e., $Equal(a, a)$;
- *Symmetric*: if a is identical to b, then b is identical to a, i.e., $Equal(a, b) \leftrightarrow Equal(b, a)$;
- *Transitive*: if a is identical to b, and b is identical to c, then a is identical to c, i.e., $Equal(a, b) \wedge Equal(b, c) \rightarrow Equal(a, c)$.

Notice that the semantics of equivalence of objects (records) satisfies the three conditions. But as stated in Section 1.1, the solution to the semantics problem is based on the syntax of records, which is only an approximate solution. Thus, the use of similarity-based and rule-based comparison methods does not guarantee all the three conditions will hold true. For similarity-based methods, by the definition (Definition 1.1) of the similarity function Sim, $Sim(A, A) = 1.0 \geq \sigma$ and $Sim(A, B) = Sim(B, A)$, the reflexive and symmetric conditions are always true. However, the transitive condition does not necessarily hold. It is plausible to find three records A, B, C such that $Equal(A, B)$ and $Equal(B, C)$ but

$\neg Equal(A, C)$. That is, $Sim(A, B) \geq \sigma$ and $Sim(B, C) \geq \sigma$ but $Sim(A, C) < \sigma$. For these cases, in reality, it could be either $Equal(A, C)$ or $\neg Equal(A, C)$. With transitive closure, we assume $\neg Equal(A, C)$ is incorrect and assert $Equal(A, C)$. Hence, transitive closure increases the number of correct duplicate pairs, and also (possibly) increases the number of false positives.

If the user determining transitive closure is not warranted for his application, he could ignore the transitive closure phase completely. However, this will exclude the increase of the correct duplicate pairs introduced by transitive closure. As an alternative, we introduce *conditional transitive closure with D-rule*, with which user may conduct. If $<A, B>$ and $<B, C>$ are obtained as duplicate pairs, then the transitive closure are conducted only when $Sim(A, B) + Sim(B, C) - 1 \geq \sigma$, i.e., records A and C are treated as duplicated. Otherwise, records A and C will not treated as duplicated. For the later case, there should be either false positives in $<A, B>$ and $<B, C>$ or correctness in $<A, C>$. To solve this issue, further pruning on these records can be employed. In Section 4.4, two pruning methods are discussed in details.

In the following, we show that the conditional transitive closure with D-rule is stronger than the normal transitive closure (the "is a duplicate of" relationship).

The D-rule, $L(A, C) \geq \sigma$, means that there is an anchor record B such that

$$Sim(A, B) + Sim(B, C) - 1 \geq \sigma$$

$$\Leftrightarrow Sim(A, B) + Sim(B, C) \geq 1 + \sigma$$

Since the similarity of records is less than or equal to 1, from the above formula,

we know that $Sim(A, B) \geq \sigma$ and $Sim(B, C) \geq \sigma$. Thus, $L(A, C) \geq \sigma$ implies that there is an anchor record B such that records A and B is obtained as a duplicate pair, and records B and C is also obtained as a duplicate pair. Under the transitive closure, records A and C will also be obtained as a duplicate pair. Thus, conditional transitive closure with D-rule is stronger than transitive closure.

With this conditional transitive closure, we can achieve the increase of the number of correct duplicate result and exclude extra false positives to some extends, thus improving the accuracy. In [LLL00], the authors also introduce a similar technique. Records are duplicate with certainty factor cf, $0 \leq cf \leq 1$, in its rule-based comparison system. Given duplicate pairs, $<A, B>$ and $<B, C>$, transitive closure is only conducted on them when the times of the certainty factors of both is larger than a given threshold.

In the following, we assume that the transitive closure is always computed. With this assumption, we indicate that RAR1 and RAR2 can work with any other comparison method to provide a tradeoff between correctness and efficiency, but will not introduce more false positives.

For simplicity, we assume that the comparison method is Record Similarity, which is efficient and has good performance in detecting duplicate records. However, it does not satisfy the properties LP and UP. Thus, the D-rule and ND-rule are not satisfied. The following example shows that three records do not satisfy the properties LP and UP with Record Similarity.

Example 3.3 *Consider the three records in Table 3.2. For brevity, we assume*

Record	Name
A	li li li
B	li zhao
C	zhao zhao zhao

Table 3.2: Three records that do not satisfy LP and UP with Record Similarity.

that the records only have a "Name" field. Easily, we have $Sim^{RS}(A, B) = 4/5$, $Sim^{RS}(B, C) = 4/5$, *and* $Sim^{RS}(A, C) = 0$. *However,* $L_B(A, C) = Sim^{RS}(A, B) + Sim^{RS}(B, C) - 1 = 3/5$. *Thus* $Sim^{RS}(A, C) < L_B(A, C)$. *So LP is not satisfied by Record Similarity. Similarly,* $U_A(B, C) = 1 - |Sim^{RS}(A, B) - Sim^{RS}(A, C)| = 1/5$. *Thus* $Sim^{RS}(B, C) > U_A(B, C)$. *So UP is not satisfied by Record Similarity as well.*

Hence, RAR1 and RAR2 with Record Similarity may introduce extra false positives and miss correct duplicate pairs as compared to SNM+RS. As we have shown that D-rule is stronger than transitive closure above, RAR1 introduces less false positives than transitive closure does.

Between the two duplicate rules, D-rule and ND-rule, only D-rule will introduce duplicate pairs. That is, for two duplicate records, they are only determined as duplicated either by D-rule or direct comparison.

Let TC(DR(SNM)) and TC(DR(RAR1)) be the transitive closure of DR(SNM) and DR(RAR1) respectively. We have the following theorem.

Theorem 3.12 *If we run SNM and RAR1 at the same window size with Record Similarity as the comparison method, we have DR(RAR1) $\subseteq$ TC(DR(SNM)).*

Proof: For any duplicate pair $<A, C>$ in DR(RAR1), it can only be obtained in the following two ways:

1. If $<A, C>$ is obtained with direct comparison, i.e., $Sim(A, C) \geq \sigma$, then it is in DR(SNM) since SNM and RAR1 are run at the same window size. So $<A, C>$ is in TC(DR(SNM)),

2. If $<A, C>$ is obtained with D-rule, i.e., $L(A, C) \geq \sigma$, then there is an inAnchor record B, such that $Sim(A, B) \geq \sigma$ and $Sim(B, C) \geq \sigma$. Thus $<A, B>$ and $<B, C>$ are in DR(SNM). So $<A, C>$ is in TC(DR(SNM)).

So we have DR(RAR1) $\subseteq$ TC(DR(SNM)). ∎

Since Record Similarity does not satisfy the properties LP and UP, with Record Similarity as comparison method, RAR1 cannot guarantee to detect the same result as SNM does. However, the above theorem says that under transitive closure, with Record Similarity, RAR1 will not introduce more false positives than SNM does.

Corollary 3.13 $TC(DR(RAR1)) \subseteq TC(DR(SNM))$. ∎

As shown in next section, RAR1 takes less comparisons than SNM does. Hence, if the comparison method Record Similarity is used, RAR1+RS will provide a tradeoff between accuracy and efficiency as compared to SNM+RS. Generally, RAR1+RS obtains less (correct and false) duplicate pairs and takes less time, while SNM+RS obtains more duplicate pairs and takes more time.

3.6 Experimental Results

3.6.1 Databases

We test the performance on a small real database, company; and a set of large synthetic databases, customers.

We get the company database from the authors of [LLL00]. The company database has 856 records and each record has 7 fields: company code, company name, first address, second address, currency used, telephone number and fax number.

All the customer databases are generated automatically by a database generator that allows us to perform controlled studies. This database generator provides a large number of parameters including, the size of the database, the percentage of duplicate records in the database, the amount of error to be introduced in the duplicated records in any of the attribute fields, and number of duplicates of per record. Each record generated consists of the following 13 fields: social security number, first name, last name, gender, marital status, race, nation, education, home phone, business name, business address, occupation and business phone. The error introduced in the duplicate records range from small typographical changes, to large changes of some fields. Each record may be duplicated more than once with Zipf distribution.

Zipf distribution [Li92, Lyn88, Ros98] is commonly used to represent highly skewed data. It gives high probability to small numbers of duplicates, but still give

non-trivial probability to large numbers of duplicates. It is also used in [ME97] to generate databases. A Zipf distribution has two parameters $0 \leq \theta \leq 1$ and $1 \leq M$. For $1 \leq i \leq M$ the probability of i duplicates is $ci^{\theta-1}$ where the normalization constant $c = 1/\sum_{i=1}^{M} i^{\theta-1}$. Having a maximum number of duplicates M is necessary because $\sum_{i=1}^{\infty} i^{\theta-1}$ diverges if $\theta \geq 0$.

To determine baseline accuracy and efficiency of our methods, we generate a base customer database, named customer_base, as follows. We first generate a clean database with 5,000,000 records. Then we add additional 2,390,000 duplicate records into the clean database.

3.6.2 Platform

All the databases are stored as relational table in Microsoft SQL server 7.0, which runs on windows 2000. The experiments on the databases are performed on a 500 MHz Pentium II machine with 256 MB of memory. The SQL server was connected with ODBC.

3.6.3 Performance

In this section, we first compare LCSS with Record Similarity. Then we compare RAR1, RAR2 with SNM. We choose Record Similarity and SNM respectively for our performance evaluation because of their efficiency and popularity.

Database	SNM+LCSS		SNM+RS		$DR_{LCSS} \cap DR_{RS}$	$\frac{DR_{LCSS} \cap DR_{RS}}{DR_{LCSS}}$
	Total	F. P.	Total	F. P.		
company	45	1	45	1	45	100%
customer_base	2315225	862	2315242	873	2315163	99.997%

Table 3.3: Duplicate result obtained by SNM+LCSS and SNM+RS at window size 10 on the company and customer_base databases.

Comparing LCSS with Record Similarity

To compare LCSS with Record Similarity, we choose SNM as the detection method. We run SNM+LCSS and SNM+RS on the company and customer_base databases to understand the efficiency and accuracy of the comparison method LCSS.

We first run both methods at window size 10. The duplicate result is shown in Table 3.3. "F. P." in the column under each method denotes the false positives of than method. The DR_{LCSS} and DR_{RS} denote the duplicate result obtained by the method SNM+LCSS and SNM+RS respectively. From this table we can see that the comparison method LCSS is as efficient as RS in capturing duplicate records.

- The duplicate results obtained by both methods on the company database are exactly the same. Both obtain exactly the same 45 duplicate pairs, among which one pair is false positive.

- The duplicate results obtained by both methods on the customer_base database are almost the same. SNM+LCSS and SNM+RS obtain 2,315,225 and

Window Size	SNM+LCSS	SNM+RS
5	4349	4610
10	8178	8751
15	12010	12855
20	15783	16893
25	19583	21002
30	23413	25051

Table 3.4: The time in seconds taken by SNM+LCSS and SNM+RS on the customer_base database.

2,315,242 duplicate pairs respectively, among which 2,315,163 duplicate pairs are the same in both methods. That is, more than 99.997% duplicate pairs obtained by both methods are the same. Further, LCSS gets a few less false positives than Record Similarity.

We then run both methods from window size 5 to 30 on the customer_base database to test the efficiency. The time result is shown in Table 3.4. From the table, we can see that LCSS is slightly faster than RS for all window sizes. Take the result at the window size 10 as an example:

- SNM+LCSS takes 8178 seconds and SNM+RS takes 8751 seconds. The time taken by SNM+LCSS is about 8% less than that taken by SNM+RS.

Thus, the results from Table 3.3 and Table 3.4 show that the comparison

method LCSS is as efficient as RS in capturing duplicate records and slightly faster than RS.

Comparing RAR1, RAR2 with SNM

We run the methods RAR1, RAR2 and SNM on the customer databases to understand the efficiency (the comparisons saved) of our methods. We test them on databases with different window sizes, duplicate ratios, Zipf distributions, and database sizes. We employ LCSS as the comparison method. Note that from Theorem 3.10 and Theorem 3.11, we know that SNM and RAR1 obtain exactly the same duplicate result, and RAR2 obtains slightly more duplicate pairs than RAR1 does. Thus, we only show the result on the efficiency of these methods.

Varying Window Sizes

We run all methods from window size 5 to 30 on the customer_base database. The results on comparisons are shown in Table 3.5, Figure 3-5 and Figure 3-6. Figure 3-5 shows the number of comparisons taken by all methods and Figure 3-6 shows the compassion saved (in percentage) by RAR1 and RAR2 compared with SNM.

From these table and figures, we can see that the comparisons taken by RAR1 and RAR2 are much less than that taken by SNM for all the window sizes, and the reduction in comparisons will increase when the window size increases. We see that RAR1 saves comparisons from 31.48% to 38.72% and RAR2 saves comparisons from 26.98% to 55.88% when the window size increases from 5 to 30. RAR2

Win. Size	SNM	RAR1	RAR2	$\frac{SNM-RAR1}{SNM}$	$\frac{SNM-RAR2}{SNM}$
5	36988430	25343994	27007133	31.48%	26.98%
10	73976835	47254889	41854977	36.12%	43.42%
15	110965215	69394158	56083595	37.46%	49.46%
20	147953570	91557896	70072186	38.12%	52.64%
25	184941900	113799572	84049999	38.47%	54.55%
30	221930205	135996998	97918994	38.72%	55.88%

Table 3.5: On the customer_base database: the number of comparisons taken by SNM, RAR1 and RAR2.

takes more comparisons than RAR1 does when the window size is 5, but takes far less comparisons for window size 30. It is reasonable since there are two anchor records in RAR2 and each new record entering the window will compare with the two anchor records first, while in RAR1 there is only one comparison with the inAnchor record. For small window size, the two additional comparisons in RAR2 may not save more comparisons than one additional comparison in RAR1. While for large window size, the two additional comparisons can save more comparisons than one additional comparison. As shown in Figure 3-6, when the window size increases, RAR2 saves more and more comparisons than RAR1 does.

Varying Duplicate Ratios

To understand whether or how different duplicate ratios will affect the efficiency of our methods, we run RAR1, RAR2 and SNM on 5 databases with different duplicate ratios, 1%, 2%, 5%, 10% and 20%, respectively. The *duplicate ratio* is the ratio of the number of records which have duplicate records in the database to the total number of records of the database. Each database has totally 1,000,000 records. We run all methods at the window size of 10.

Figure 3-7 gives the result. It clearly shows that both RAR1 and RAR2 are much more efficiency than SNM for all duplicate ratios. The dirtier the database is, the more comparisons both RAR1 and RAR2 save. This is because the database is dirtier, the D-rule and ND-rule are mostly likely to arise. However, the duplicate ratio affects the result little.

Varying Number of Duplicates Per Record

This experiment uses databases where the records are duplicated according to the Zipf distribution with different θ values, 0.1, 0.2, 0.4, and 0.8, respectively. The maximum number of duplicates per record is kept constant at 10. Each database has 1,000,000 records, and we run all methods at the window size of 10.

The result of the this experiment is shown in Figure 3-8 and the analysis is similar to the above test.

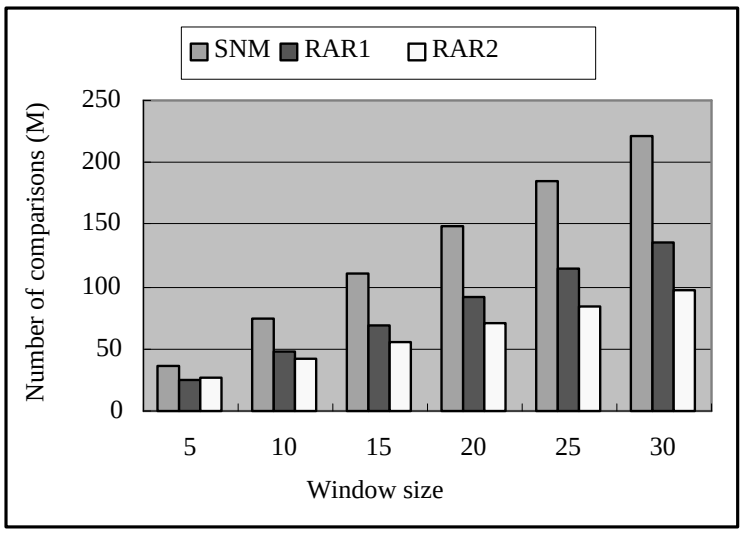

Figure 3-5: Varying window sizes: the number of comparisons taken by SNM, RAR1 and RAR2.

Scalability: Varying Database Sizes

At last, we run RAR1 RAR2 and SNM on 4 databases with different number of records, 1, 2, 5 and 10×10^6 records respectively, to test the scalability. Each database has duplicate ratio 0.2. We run all methods at window size of 10.

The results are given in Figure 3-9. All methods show linear scalability with the number of records from 1M to 10M. However, our methods are much more scalable.

3.6.4 Number of Anchor Records

As the performance shown in last subsection, for large window size, RAR2 saves more comparisons than RAR1 does, thus the RAR2 is superior to RAR1. One

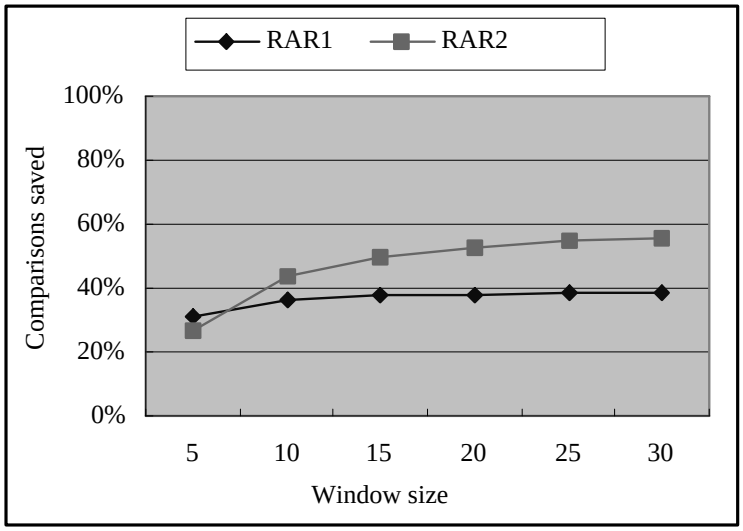

Figure 3-6: Varying window sizes: the comparison saved by RAR1 and RAR2 as compared to SNM.

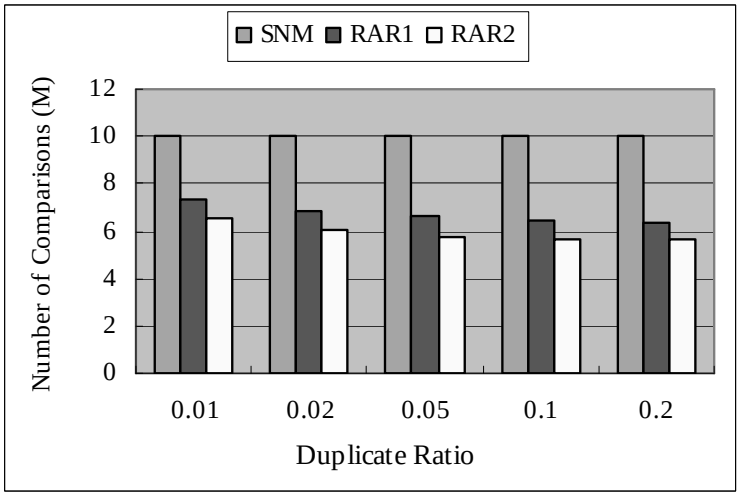

Figure 3-7: Varying duplicate ratios: the number of comparisons taken by SNM, RAR1 and RAR2.

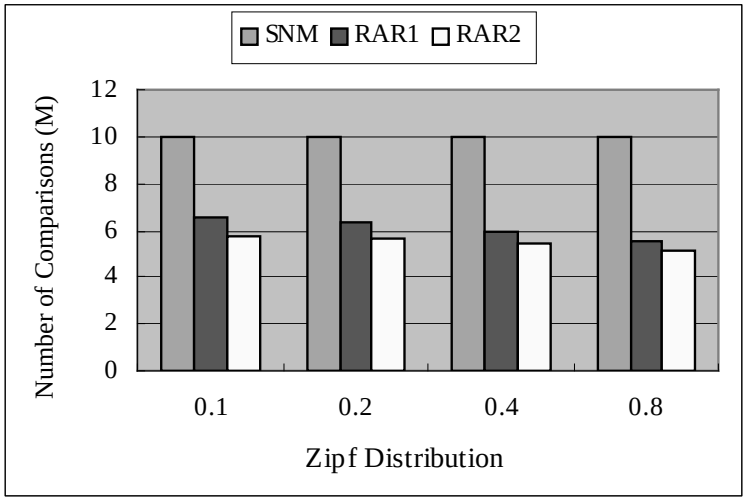

Figure 3-8: Varying number of duplicates per record: the number of comparisons taken by SNM, RAR1 and RAR2.

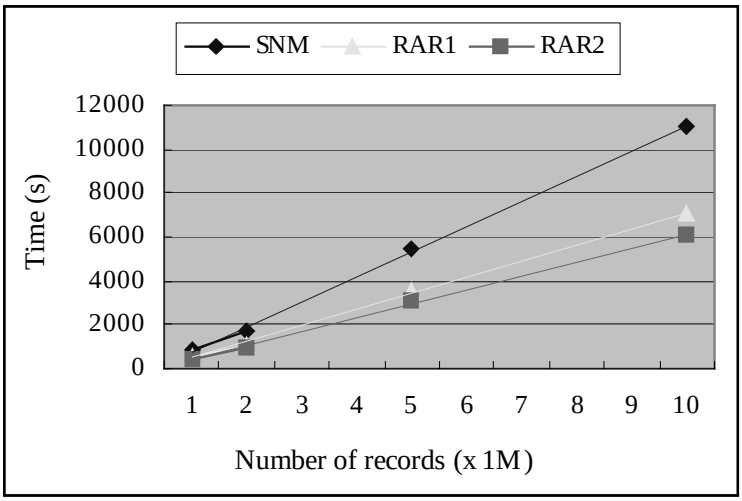

Figure 3-9: Varying database size: the scalability of RAR1 and RAR2.

natural question to pose is that could we use more anchor records for large window size. To answer this question, in the following, we first give a theoretical analysis on the relationship between window size and the number of anchor records.

For easy discussion, we assume that all anchor records are inAnchor records (similar for outAnchor records). Consider RAR1, for each record in the window, suppose that no comparison required with the new record entering the window has average probability of p. That is, for the new record entering the window and each record in the window, the average probability of that they are compared is $(1-p)$. For the N records in D, there are $\lfloor \frac{N}{\omega+1} \rfloor$ anchor records and $N - \lfloor \frac{N}{\omega+1} \rfloor$ non-anchor records. For each record entering the window, if it is chosen as the anchor record, there are ω comparisons. Otherwise, there are $1 + (1-p)(\omega-1)$ comparisons. Thus the total number of comparisons taken by RAR1 is: $\omega \lfloor \frac{N}{\omega+1} \rfloor + (1 + (1-p)(\omega-1))(N - \lfloor \frac{N}{\omega+1} \rfloor)$, which approximately equals to

$$\frac{\omega N}{\omega+1} + (1 + (1-p)(\omega-1))\frac{\omega N}{\omega+1} \tag{3.5}$$

Similarly, the total number of comparisons taken by method (the last record) with k inAnchor records is:

$$c_\omega(k) = \frac{\omega k N}{\omega+1} + (k + (1-p)^k(\omega-k))\frac{(\omega+1-k)N}{\omega+1} \tag{3.6}$$

Obviously, when $k = 0$ and $k = \omega$, $c_\omega(k)$ takes the maximal value, $c_\omega(0) = c_\omega(\omega) = \omega N$,. Thus, given D, $|D| = N$, and ω, the problem of saving the most

Window Size	p
5	0.4722
10	0.4415
15	0.4281
20	0.4213
25	0.4168
30	0.4139

Table 3.6: The value of p relative to different window sizes.

comparisons is to find the k such that $c_\omega(k)$ minimizes. However, this k is not easy to obtain since p is not a constant value and is affected by ω. Theoretically, given ω, it's hard to get the value of p. But performance study shows that the value p ranges from 0.4 to 0.5. Table 3.6 shows the value of p relative to different window size obtained from performance tests. From this table, we can see that p decreases when ω increases. Example 3.4 shows an example to compute p from ω from the performance result in Table 3.5.

Example 3.4 *Suppose* $\omega = 10$. *SNM takes* $10N$ *comparisons. For* $k = 1$, *Equation (3.6) is* $\frac{10N}{11} + (1 + (1-p) \times 9)\frac{10N}{11} = \frac{10(11-9p)}{11}N$. *Let* c_{SNM} *and* c_{RAR1} *denote the number of comparisons taken by SNM and RAR1 respectively. Then* $\frac{c_{RAR1}}{c_{SNM}} = \frac{11-9p}{11}$. *From the performance tests (Table 3.5 and Figure 3-6) on window size 10, we see that RAR1 takes about 63.88% comparisons of SNM. So* $\frac{11-9p}{11} = 0.6388$. *Thus, we get* $p = 0.45$.

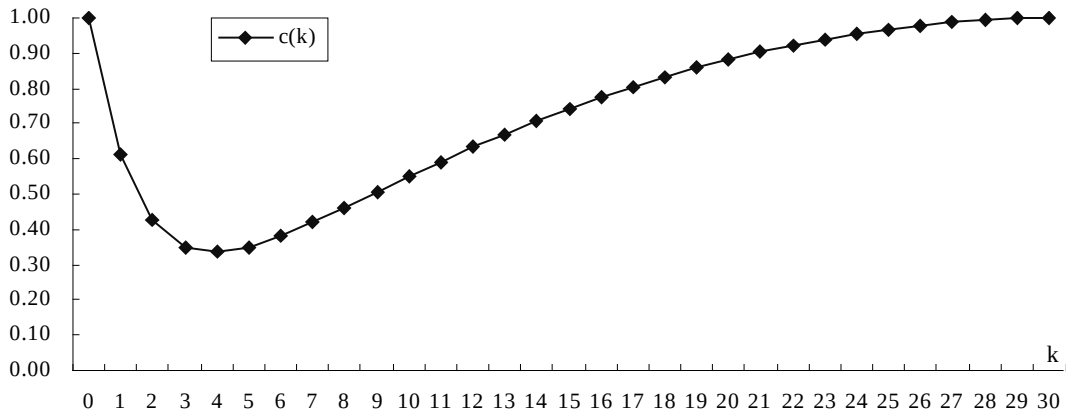

Figure 3-10: The values of $c_\omega(k)$ over ωN for different k with $\omega = 30$.

Figure 3-10 shows the values of $c_\omega(k)$ for different k with $\omega = 30$. we can see that $c_\omega(k)$ take the minimal value when $k = 4$, which suggests that, with 4 inAnchor records at window size 30, we can achieve the best performance result on time.

3.7 Summary

In this chapter, we first show two similarity properties, LP and UP, which are derived from the triangle inequality of the distance measure d, the difference of one and the similarity. We then propose a new comparison method, LCSS, based on the longest common subsequence, and discover that it satisfies these two properties LP and UP. Take advantage of these properties, we indicate two duplicate rules D-rule and ND-rule directly from the properties LP and UP respectively. With D-rule and ND-rule, we propose two new data cleansing methods, RAR1 and RAR2. These

two methods can largely reduce unnecessary comparisons by efficiently using the properties satisfied by the comparison method, while existing cleansing methods, such as SNM, cannot. The performance study on real and synthetic datasets shows that both RAR1 and RAR2 save comparisons significantly without impairing accuracy. Further, both RAR1 and RAR2 are much more efficient than SNM for databases with different parameters. In addition, for large window size, RAR2 is more efficient than RAR1 further. Theoretical analysis on the relationship between the number of anchor records and the window size is presented.

Chapter 4

A Fast Filtering Scheme

4.1 Introduction

Large proportion of time in data cleansing is spent on the comparisons of records. Reducing the time on comparisons (the number of comparisons and the time for each comparison) will be critical for reducing the whole cleansing time. We have explained how to reduce the number of comparisons with RAR1+LCSS and RAR2+LCSS in Chapter 3. In this chapter, we show how to reduce the time for each comparison.

Consider SNM+RS, if the window size is ω and the average time for each comparison is T, then there will be ωN comparisons and the time taken will be ωNT. However, among the ωN comparisons, generally only a few records compared will be detected as duplicate ones, while the other comparisons do not contribute duplicate records, which results in wasting a great amount of time since

the existing comparison methods are quite expensive. We call these comparisons that do not contribute duplicate records as *uncontributive comparisons*. Hence, the question is that whether we can get the same duplicate result as the existing methods while taking much less time to run.

One intuitive idea is to perform a scan on the database first to quickly identify possible duplicate records, that is, to fast filter out the uncontributive comparisons (non-duplicate records). Then we apply any existing method on the duplicate result to further eliminate false positives that are generated by the first scan. Since the duplicate result is much smaller than the initial database, it takes far less time to process the records in the result. The two advantages of this approach are: 1) it is fast, and 2) it can be combined with any existing method.

How to carry out the first scan therefore is rather important. Before cleansing, there is a pre-processing on records, such as in [HS95, LLL00], which deals with data type checks, format standardization, and inconsistent abbreviations etc. After the pre-processing, intuitively, for any two duplicate records, the corresponding fields in them should have almost the same characters. In other words, for two records, whose characters of the corresponding fields have large difference, they cannot be duplicate records.

Based on this intuition, in this paper, we first propose a simple and fast comparison method, *TI-Similarity*. Similar to RS and LCSS, TI-Similarity also computes the degree of similarity for records and two records with similarity exceeding a certain threshold are treated as a duplicate pair. In TI-Similarity, each field is

simply treated as a set of characters. The field similarity is defined as the number of characters in the intersection of the corresponding fields divides by the larger number of characters in the two fields. For a field in two records with m and n characters respectively, the time complexity of TI-Similarity is $O(m+n)$, while it is $O(mn)$ for edit distance, Record Similarity and LCSS. Thus TI-Similarity is much faster.

Furthermore, we can show that the distance on TI-Similarity, the difference of one and the TI-Similarity, satisfies the triangle inequality proposed in Section 3.2. Thus, TI-Similarity also satisfies the two properties, LP and UP. That is, For any three records A, B and C, let $Sim^{TI}(A,C)$ be the TI-Similarity of records A and C, etc. TI-Similarity satisfies

- *Lower Bound Similarity Property (LP):*

  $Sim^{TI}(A,C) \geq Sim^{TI}(A,B) + Sim^{TI}(B,C) - 1$

- *Upper Bound Similarity Property (UP):*

  $Sim^{TI}(A,C) \leq 1 - |Sim^{TI}(A,B) - Sim^{TI}(B,C)|$

Therefore, when TI-Similarity is chosen as comparison method, we can also get the benefits that RAR1 and RAR2 provide.

With TI-Similarity and RAR1 (we discuss only on RAR1, but it also applies to RAR2), our new approach can be outlined as follows (see Figure 4-1): We first perform a fast scan on the whole database. We call this *filtering process*. The filtering process is carried out by RAR1+TI. This process takes far less time than existing methods (e.g., SNM+RS) and can detect possible duplicate records but

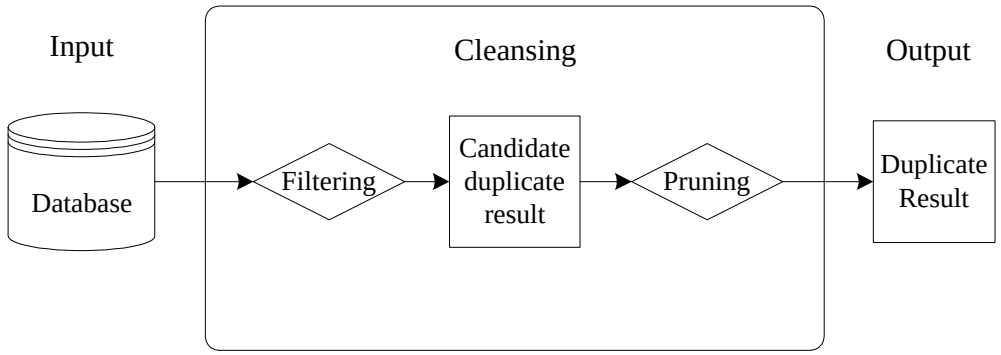

Figure 4-1: The filtering and pruning processes.

may get some extra false positives. The duplicate result obtained by the filtering process is also called as *candidate duplicate result*. Then a pruning on the candidate duplicate result is performed with a more trustworthy comparison method, such as Equational Theory, edit distance, Record Similarity and LCSS etc. We call this *pruning process*. The pruning process can eliminate the false positives obtained in the filtering process. Since the duplicate result is generally far less than the database, the time taken by the pruning process is only a small portion of the whole data cleansing processing time. We term our approach as (RAR1+TI, X), where X is the comparison method used in the pruning process. X is independent of the filtering process and can be chosen as any comparison method.

The rest of the chapter is organized as follows. In the next section, we introduce the simple comparison method, TI-Similarity. In Section 4.3, we describe the filtering scheme with RAR1 (RAR2) and TI-Similarity. In Section 4.4, we show

how to perform the pruning process on the duplicate result obtained by the filtering process. We give the performance results in section 4.5 and summary in Section 4.6.

4.2 A Simple and Fast Comparison Method: TI-Similarity

We have shown in Section 2.1, a pre-processing on the records in the database will be conducted before the cleansing process, like in [HS95, LLL00]. After the pre-processing stage, intuitively, for two records representing the same entity, the corresponding fields should have almost the same characters. In other words, for two records, if the characters of the corresponding fields in them have large difference, they cannot be duplicate records. This intuitive idea is similar to that in counting filter [JTU96] used for approximate string matching, which is to find all segments of a long text T, $|T| = m$, whose edit distance to a short pattern P is at most k, where $0 < k < |P|$.

Based on this intuition, we propose the *TI-Similarity* as follows.

Similar to RS and LCSS, we also set field weightages and compute field similarity. The field weightages indicate the relative importance of fields and are decided by experimental tests as well.

The values in fields are simply treated as sets of characters. Each character has an associated number that identifies the serial number of the character appearing in the field. For example, for a string "ababc", it is transferred to the character

set {a1, b1, a2, b2, c1}. Notice that transferring a character to the character with associated number is for simply discussion. In real implementation, we do not need to do this transfer.

Field Similarity: Suppose a field F in record A has the character set $A_F = \{x_1, x_2, \cdots, x_n\}$ and the corresponding field in record B has the character set $B_F = \{y_1, y_2, \cdots, y_m\}$, where x_i, $1 \leq i \leq n$, and y_j, $1 \leq j \leq m$, are characters with associate numbers. We have the field similarity of field F for A and B as:

$$Sim_F^{TI}(A, B) = min\{\frac{|A_F \cap B_F|}{|A_F|}, \frac{|A_F \cap B_F|}{|B_F|}\} \tag{4.1}$$

If F is understood, for simplicity, we write A_F and B_F as A and B respectively. For example, suppose the field F of A is "abccddde" and B is "aaccddf", then A = {a1, b1, c1, c2, d1, d2, d3, e1} and B = {a1, a2, c1, c2, d1, d2, f1}. Also $A \cap B$ = {a1, c1, c2, d1, d2}, so we have $|A| = 8$, $|B| = 7$, and $|A \cap B| = 5$. Thus, $Sim_F^{TI}(A, B) = |A \cap B|/|A| = 5/8$.

However, due to its simplicity, TI-Similarity does not consider the order of characters in the field. Therefore it is possible to introduce some false positives. E.g., if A = "abcd" and B = "dcba", we will get $Sim(A, B) = 1$. Thus we get records A and B as duplicate but they may not represent the same entity. So a pruning on duplicate result with more trustworthy records comparison methods is performed. In Section 4.4, we will show how to carry out the pruning on duplicate result in details.

Based on field similarity, we can compute the similarity for records. Suppose

```
float get_TI-Similarity(char* s1, char* s2)
{
    int number[256] = {0};
    int i = 0, total = 0;
    int len1 = strlen(s1), len2 = strlen(s2);
    for (i=0; i<len1; i++)
        number[s1[i]]++;
    for (i=0; i<len2; i++) {
        if (number[s2[i]]-- >= 0)
            total++;
    }
    return (float)total/max(len1, len2);
}
```

Figure 4-2: The fast algorithm to compute field similarity.

that a database has fields $F_1, F_2, \cdots, F_n$ with field weightages $W_1, W_2, \cdots, W_n$ respectively, where $\sum_{i=1}^{n} W_i = 1$. Given two records A and B, let $Sim_{F_1}^{TI}(A, B)$, $Sim_{F_2}^{TI}(A, B)$, $\cdots$, $Sim_{F_n}^{TI}(A, B)$ be the field similarities computed. The TI-Similarity of the two records is given by the expression:

$$Sim^{TI}(A, B) = \sum_{i=1}^{n} (Sim_{F_i}^{TI}(A, B) \times W_i) \tag{4.2}$$

We use superscript TI to distinguish the TI-Similarity with the other similarity methods such as Record Similarity and LCSS. If without ambiguous, we just write Sim^{TI} as Sim for simplicity.

The similarity of records computed by TI-Similarity is always between 0.0 and 1.0. Two records are treated as a duplicate pair if the similarity of them exceeds a certain threshold such as 0.8.

Note that the similarity of records depends on the similarity of fields. In Figure 4-2, we give a very fast algorithm to calculate the field similarity. The algorithm shown here is based on the algorithm proposed in [Nav97] with a few modifications for our case. It is easy to see that if two strings' length are m and n respectively, the time complexity of TI-Similarity is $O(m+n)$, while the time complexities of edit distance, Record Similarity and LCSS are $O(mn)$. Thus TI-Similarity is much faster than existing comparison methods.

Let $d(A,B) = 1 - Sim(A,B)$ and $d_F(A,B) = 1 - Sim_F(A,B)$, suppose $|A_F| \geq |B_F|$, then $d_F(A,B) = 1 - |A_F \cap B_F|/|A_F| = (|A_F| - |A_F \cap B_F|)/|A_F| = |A_F - B_F|/|A_F|$.

we have the following Lemma, which is similar to Lemma 3.5, for TI-Similarity.

Lemma 4.1 *Given a filed F, for any three records A, B and C, we have $d_F(A,B) + d_F(B,C) \geq d_F(A,C)$.*

Proof: For simplicity, we drop F from A_F, B_F, C_F and d_F. Without loss of generality, suppose $|A| \geq |C|$.

(1) $|A| \geq |B| \geq |C|$.

$$d(A,B)+d(B,C)=(1-\tfrac{|A\cap B|}{|A|})+(1-\tfrac{|B\cap C|}{|B|})$$
$$=\tfrac{|A|-|A\cap B|}{|A|}+\tfrac{|B|-|B\cap C|}{|B|}\geq\tfrac{|A|-|A\cap B|}{|A|}+\tfrac{|B|-|B\cap C|}{|A|}$$
$$=\tfrac{|A-B|}{|A|}+\tfrac{|B-C|}{|A|}\geq\tfrac{|A-C|}{|A|}=d(A,C).$$

(2) $|A|\geq|C|\geq|B|$.

$$d(A,B)+d(B,C)=(1-\tfrac{|A\cap B|}{|A|})+(1-\tfrac{|B\cap C|}{|C|})$$
$$\geq(1-\tfrac{|A\cap B|}{|A|})+(1-\tfrac{|B\cap C|}{|B|})$$
$$=\tfrac{|A|-|A\cap B|}{|A|}+\tfrac{|B|-|B\cap C|}{|B|}$$
$$\geq\tfrac{|A|-|A\cap B|}{|A|}+\tfrac{|B|-|B\cap C|}{|A|}\geq d(A,C).$$

(3) $|B|\geq|A|\geq|C|$.

$$d(A,B)+d(B,C)=(1-\tfrac{|A\cap B|}{|B|})+(1-\tfrac{|B\cap C|}{|B|})$$
$$=1+\tfrac{|B|-|A\cap B|-|B\cap C|}{|B|}$$

If $(|B|-|A\cap B|-|B\cap C|)\geq 0$, then

$$d(A,B)+d(B,C)\geq 1\geq d(A,C).$$

If $(|B|-|A\cap B|-|B\cap C|)<0$, then

$$1+\tfrac{|B|-|A\cap B|-|B\cap C|}{|B|}\geq 1+\tfrac{|B|-|A\cap B|-|B\cap C|}{|A|}$$
$$=\tfrac{|A|-|A\cap B|}{|A|}+\tfrac{|B|-|B\cap C|}{|A|}\geq d(A,C).$$

Thus we complete our proof. ∎

Theorem 4.2 *For any records A, B and C, we have*

$$d(A,B)+d(B,C)\geq d(A,C).$$

Proof: The proof is the same with the proof of Theorem 3.7. ∎

Similarly, we immediately have the following theorem.

Theorem 4.3 *TI-Similarity satisfies the properties LP and UP.*

4.3 Filtering Scheme

When we refer to the similarity of records, we always mean that the similarity is computed by TI-Similarity.

We have shown the comparison method TI-Similarity and that it satisfies the properties LP and UP. Hence, the D-rule and ND-rule are correct and consistent with TI-Similarity. Thus RAR1 and RAR2 works correctly with TI-Similarity. This means that RAR1 and RAR2 will not introduce extra false positives or missing some correct duplicate pairs. However, TI-Similarity definitely will introduce extra false positives as it is quite simple and does not consider the order of characters. Thus we could not run RAR1+TI and RAR2+TI as an sole cleansing method.

To get the advantage (very fast) of RAR1+TI but exclude its disadvantage (lots of extra false positives), further consideration is needed.

From observing real world scenarios [Her96], the size of the duplicate pairs obtained is at least one order of magnitude smaller than the corresponding number of comparisons taken. Thus, intuitively, we could run RAR1+TI as a fast filter and then prune the result with existing more trustworthy method. We call this *filtering scheme* and it is shown in Figure 4-1.

Take SNM+RS as an example. For window size ω, there will be ωNT total time, where T is the average comparison time. Now consider RAR1+TI and with pruning by Record Similarity. The filtering time is $p\omega NT'$, where $0.0 \leq p \leq 1.0$

is the ratio that the number of comparisons taken by RAR1 to the number of comparisons taken by SNM, and T' is the average time taken by TI-Similarity for each comparison. The pruning time is dT, where d is the number of duplicate pairs obtained by RAR1+TI. Thus the total time for the filtering scheme is $p\omega NT' + dT$. Since $T' \ll T$ and $d \ll \omega N$, then $p\omega NT' + dT \ll \omega NT$.

Let DR(RAR1+TI) and DR(RAR1+TI, LCSS) be the duplicate result obtained by RAR1+TI and (RAR1+TI, LCSS) respectively. Obviously DR(RAR1+TI, LCSS) $\subseteq$ DR(RAR1+TI).

Theorem 4.4 *Under the same similarity threshold σ and the same field weightages, and running at the same window size, the filtering scheme (RAR1+TI, LCSS) obtain the same duplicate result with RAR1+LCSS. That is DR(RAR1+TI, LCSS) = DR(RAR1+LCSS).*

Proof: To prove DR(RAR1+TI, LCSS) = DR(RAR1+LCSS), we only need to show that TI-Similarity does not filter the duplicate pairs obtained by LCSS under the above conditions. That is, to prove DR(RAR1+TI) $\subseteq$ DR(RAR1+LCSS).

For any two records A and B and any Field F, we have $Sim_F^{TI}(A, B) = \frac{|A_F \cup B_F|}{|A_F|}$, and $Sim_F^{LCSS}(A, B) = \frac{lcs(A_F, B_F)}{|A_F|}$. Obviously, $lcs(A_F, B_F) \leq |A_F \cup B_F|$, thus $Sim_F^{LCSS}(A, B) \leq Sim_F^{TI}(A, B)$. Since with the same field weightages, thus $Sim^{LCSS}(A, B) \leq Sim^{TI}(A, B)$. So DR(RAR1+TI) $\subseteq$ DR(RAR1+LCSS). ∎

Thus, through combining the RAR1 with TI-Similarity, we can achieve the benefit of both. RAR1 reduces unnecessary comparisons and TI-Similarity reduces the time for each comparison.

4.4 Pruning on Duplicate Result

After the filtering process, a lot of non-duplicate records have been filtered out and we obtain the candidate duplicate result. Since TI-Similarity used in the filtering process is quite simple and the order of characters is not considered, the result contains correct duplicate records and false positives as well. Thus a pruning with more trustworthy comparison methods, such as Equational Theory, RS etc., on the duplicate result is required.

One simple way is to re-compare each duplicate pair in the duplicate result with more trustworthy comparison method. We call this as *direct pruning*. Obviously, the direct pruning can be easily integrated into the filtering process, i.e., when a duplicate pair is obtained in the filtering process, we do the pruning immediately. Especially, the direct pruning is useful when the candidate duplicate results is large and cannot be kept in memory.

When correct duplicate records are important, an alternative method, called *transitive closure pruning*, could be used. The transitive closure pruning is to compute transitive closure on candidate duplicate result first, then do prune on each equivalent class.

Formally, let CDR be the duplicate result obtained by the filtering process. We define the equivalent relation among the records, $A \sim B$, if B is a duplicate record of A under transitive closure. The transitive closure is executed on pairs of record id's, each at most 30 bits, and fast solutions to compute transitive closure exist [AJ88, ME97]. Let $X_A = \{B|A \sim B\}$. Then $\{X_A\}$ are equivalent classes

under this equivalent relation. Thus for any two records A and B, we have either $X_A = X_B$ or $X_A \cap X_B = \emptyset$.

The transitive closure pruning on the candidate duplicate result is to perform pair-wise comparison for the records in each X_A. The transitive closure pruning will introduce more correct duplicate records and eliminate the false positives introduced by transitive closure. However, when the candidate duplicate records cannot be kept in memory, the transitive closure pruning will take some more time than direct pruning. Thus, there is a trade-off between correct duplicate records and time. Example 4.2 shows how the pruning is performed.

Example 4.1 *If CDR = {<A, B>, <B, C>, <D, E>}, then* $X_A = \{A, B, C\}$ *and* $X_D = \{D, E\}$. *In the direct pruning, we just compare the three pairs in CDR. In the transitive closure pruning process, we pair-wisely compare records A, B and C, and pair-wisely compare records D and E.*

For both pruning methods, we will employ some other more trustworthy comparison methods, such as Equational Theory, edit distance, RS and LCSS etc., instead of TI-Similarity used in the filtering process. The pruning is not limited to only one comparison method. In some cases, having false positives is much worse than missing some correct duplicate records. The accuracy of the result (maximizing the number of correct duplicate records while minimizing the number of false positives) is therefore of paramount importance. Since no comparison method is completely trustworthy, we can perform the pruning with two or more trustworthy comparison methods and obtain the result determined as duplicate in

all the methods. This works because that duplicate result obtained by the filtering process is far less than the database and the number of records in X_A is also small. From observing real world scenarios [HS98], the size of the data set over which the equivalent classes is computed is at least one order of magnitude smaller than the corresponding database of records. Thus, the number of comparisons taken on the pruning process is far smaller than that taken on the filtering process. This is still true even for a database that consists of many duplicate records. The following gives such an analysis example.

Example 4.2 *Suppose that a database has N records. For easy comparison, we assume that the detection method used in the filtering process is SNM with a window size of 10 instead of RAR1. We obtain 20% of the records in the database as duplicate records and the average number of X_A is 5. Then the filtering process requires $10N$ comparisons and the pruning process requires $\frac{20\% \times N}{5} \times C_5^2 = 2N/5$ comparisons. Thus the number of comparisons in the pruning process is only $1/25$ of that in the filtering process.*

Therefore, the time taken by the pruning process is only a small portion in the whole data cleansing processing time.

4.5 Performance Study

4.5.1 Performance

We test the performance on synthetic databases generated by the database generator used in Section 3.6.3.

We compare the filtering approach (RAR2+TI, LCSS) with RAR2+LCSS(the similar results apply to RAR1). To understand the efficiency of this approach, similar to the performance tests done in Section 3.6.3, we test both methods on a set of databases with variant window sizes, database duplicate ratios, and database sizes. In all tests, the time taken in our approach includes the filtering process time and the pruning process time. We adopt the direct pruning and integrate it into the filtering process. We get the pruning time by first running filtering process without pruning, then running the filtering process with immediately pruning, and the pruning time is their difference.

Varying Window Sizes

We first run (RAR2+TI, LCSS) and RAR2+LCSS on the customer_base database.

Figure 4-3 shows the time required for each method. We can see that (RAR2+TI, LCSS) is much faster than RAR2+LCSS for all the window sizes. Time saving effect becomes much more significant as the window size increases. Thus (RAR2+TI, LCSS) further scales much better than RAR2+LCSS with the increase of window size.

Figure 4-4 shows the efficiency of the filtering process. For all the window sizes,

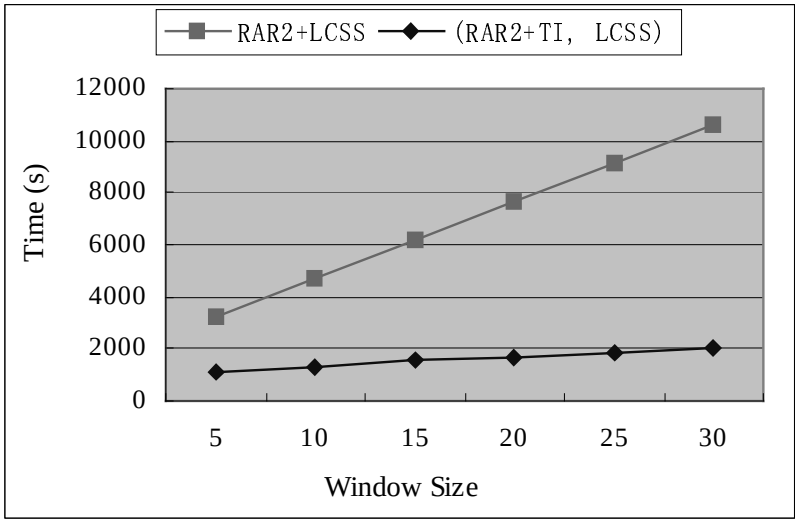

Figure 4-3: Varying window size: time taken.

the number of CDR is only a few more than that of DR. That is, only a few extra false positives (less than 5%) is introduced by the filtering process, which shows that the filtering process is efficient in filtering out uncontributive comparisons.

Figure 4-5 shows the time taken by filtering process and pruning process. We can see that the time taken by filtering process increases as the window size increases, while the time taken by pruning process is almost the same. From Figure 4-4, we know that the candidate duplicate result increases very slowly as window size increases. Since pruning is only performed on the candidate duplicate result, thus window sizes do not affect the pruning time much.

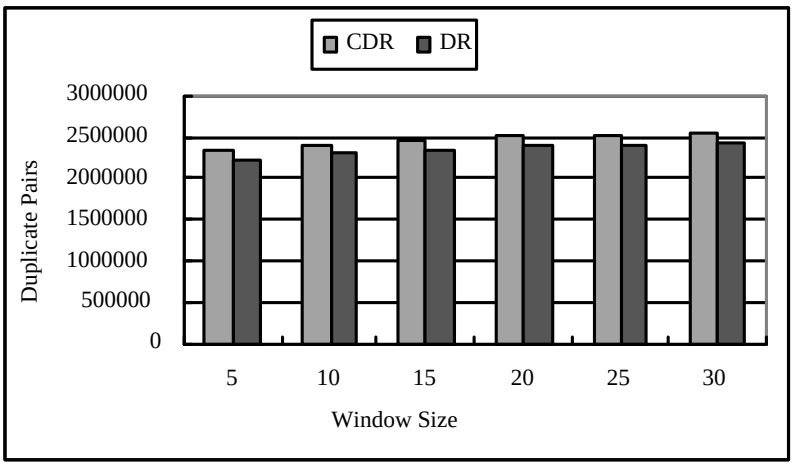

Figure 4-4: Varying window size: result obtained.

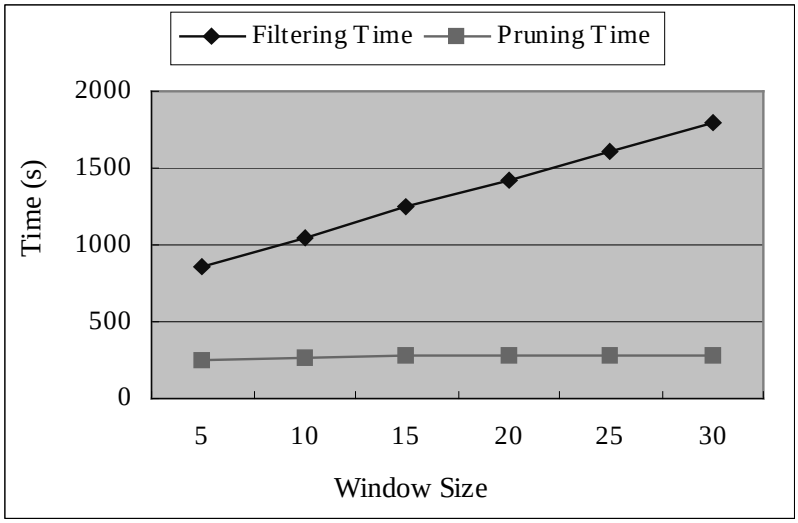

Figure 4-5: Varying window size: filtering time and pruning time.

Varying Duplicate Ratios

We then run (RAR2+TI, LCSS) and RAR2+LCSS on the same 5 databases with different duplicate ratios used in Section 3.6.3. We run both methods at the window size of 10. The time taken by each method is shown in Figure 4-6, from which we see that the time taken by (RAR2+TI, LCSS) increases as the duplicate ratio increases while RAR2+LCSS decreases. This is because when the duplicate ratio increases, the candidate duplicate result will increase. Thus the pruning process time increases. However, the time increased with the duplicate ratio is quite slowly compared with the time taken by RAR2+LCSS. Even the database is heavy duplicated, the time taken by (RAR2+TI, LCSS) is still much less than that taken by RAR2+LCSS. Hence, the duplicate ratio has little effect on the efficiency. For different duplicate ratios, (RAR1+TI, LCSS) always saves a great amount of time.

Varying Database Sizes

At last, we run (RAR2+TI, LCSS) and RAR2+LCSS on the 4 databases with different number of records (1, 2, 5 and 10×10^6 records). We run both methods at window size of 10. The results are shown in Figure 4-7. Both methods show linear scalability with the number of records from 1M to 10M. However, (RAR2+TI, LCSS) is much more scalable. As the number of records grows up, the difference between the two methods becomes larger and larger. Combined with the scalability result shown in Section 3.6.3, overall, our approach is about an order of magnitude

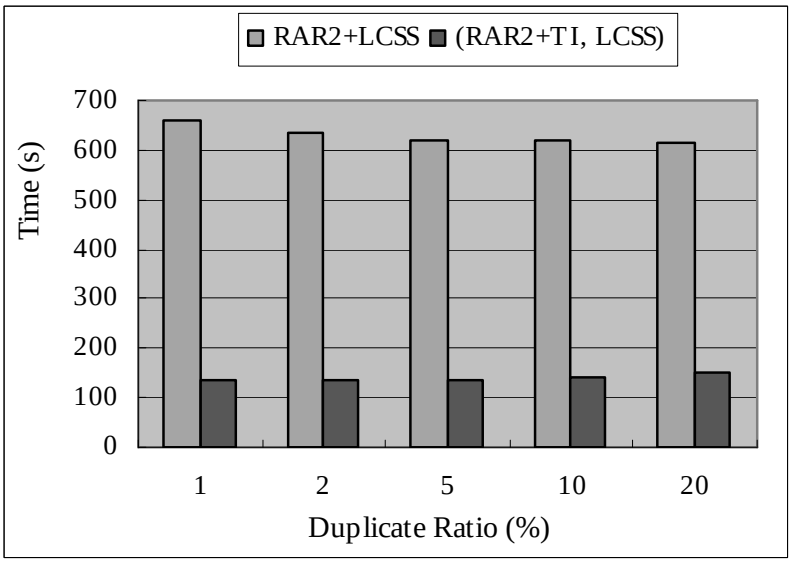

Figure 4-6: Varying duplicate ratio: time taken.

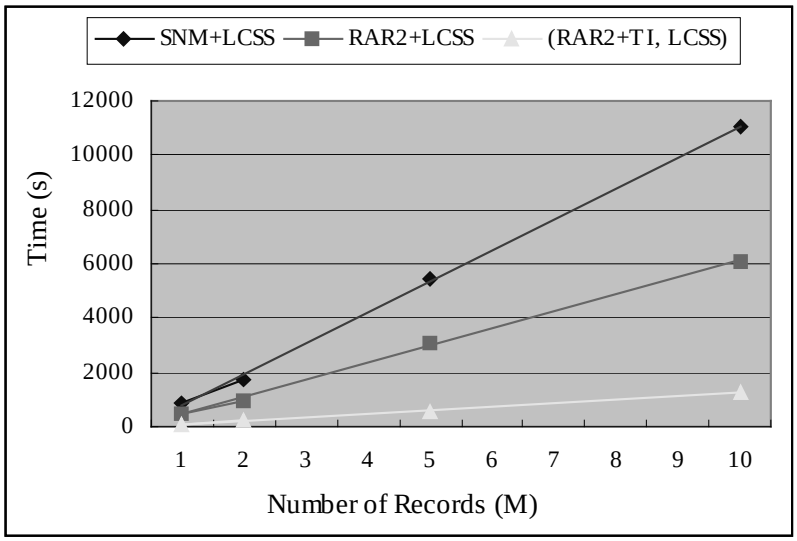

Figure 4-7: Varying database size: scalability with the number of records.

faster than existing methods for large databases.

4.6 Summary

In this chapter, we first propose a simple yet efficient comparison method, TI-Similarity, to reduce the time taken on each comparison. We then prove that TI-Similarity satisfies the triangle inequality property. Thus, both RAR1 and RAR2 also work with TI-Similarity. We run RAR1+TI (RAR2+TI) as a filter to quickly filter out a lot of uncontributive comparisons. We then perform a pruning on the candidate duplicate result obtained by the filter with more trustworthy comparison methods to eliminate false positives. Performance study shows that the filtering scheme can save time significantly, and the results from different databases and different window sizes show that this approach has good scalability.

Chapter 5

Dynamic Similarity for Fields with NULL values

5.1 Introduction

The comparison methods prove to have good performances in capturing duplicate records. However, they all have a common drawback, that is, they implicitly assume that the values in all fields are known, and NULL values on fields are simply treated as empty strings. However, in practice, databases to be cleansed very likely have records with NULL values. Treating the NULL values as empty strings is definitely not a good method and will result in a loss of correct duplicate records. Table 5.1 gives an example showing that while treating NULL values as empty strings in Record Similarity, a correct duplicate pair is lost. More analysis on the table is in Section 5.2. Thus, the fields with NULL values need to be specially

treated. In this paper, we propose a simple yet efficient method, called *Dynamic Similarity*, which solves the "NULL field problem" by dynamically adjusting the similarity for field with NULL value. For each field, there are a set of dependent fields associated with it. For any field with NULL value, the dependent fields will be used to determine its similarity. To determine the dependent field, one option is that the domain expert (database designer) can provide this information. another one is using the approximate functional dependence methods.

To test our method, we compare it with Record Similarity. The performance result shows that Dynamic Similarity can get more correct duplicate records and does not introduce new false positives as compared with Record Similarity.

The rest of this chapter is organized as follows. In next section, we propose the Dynamic Similarity. In Section 5.3, we give the performance results. We summary in Section 5.4.

5.2 Dynamic Similarity

In this section, we propose the *Dynamic Similarity*, which is an extension scheme for existing comparison methods. For easy discussion, we focus on the Record Similarity. Similarly, it can be applied to other comparison methods, such as LCSS and Equational Theory etc.

Suppose that a database has fields F_1, F_2, $\cdots$, F_n with field weightages W_1, W_2, $\cdots$, W_n respectively. Let $\mathcal{F} = \{F_1, F_2, \cdots, F_n\}$. Given two records X and Y, let X_{F_1} and Y_{F_1} be the values of field F_1 of X and Y respectively. The fields F_2,

$\cdots$, F_n are similarly defined.

Intuitively, for field with NULL value, there could be some fields that affect its similarity. These fields are called *dependent fields*. Furthermore, for each field, its dependent fields may have different importance. Thus weightages are assigned to dependent fields. To distinguish from the field weightages defined previously, these weightages are called *dependent weightages*. A formal definition is given as follows.

Definition 5.1 *Formally, a* weighted dependent function *on* $\mathcal{F}$ *is a function* Φ*:* $\mathcal{F} \mapsto 2^{\mathcal{F}\times[0,1]}$ *such that,* $\forall F_i \in \mathcal{F}$*,*

- $\forall (F_j, v_j) \in \Phi(F_i) \Rightarrow F_j \neq F_i$*;*
- $\sum_{(F_j,v_j)\in\Phi(F_i)} v_j = 1$.

The weighted dependent function defines a dependent relation on fields. It may not be symmetric, i.e., $(F_j, -) \in \Phi(F_i)$ does not imply $(F_i, -) \in \Phi(F_j)$. The $-$ in $(F_i, -)$ stands for the value that we don't care. For example, the same value in "name" field in two records imply the "age" field in the two records having the same value with large chance. However, the reverse could not be true. The dependent relation is highly domain dependent and need experimental tests to decide. Experts on the database to be cleansed will lead to a better dependent relation.

One special case of weighted dependent function is that all dependent fields have the same weightage. For simplicity, we call the special case as *dependent function*, which can be formally defined as follows with equal weightage.

Definition 5.2 *Formally, a* dependent function *on $\mathcal{F}$ is a function Φ: $\mathcal{F} \mapsto 2^{\mathcal{F}}$ such that $\forall F_i \in \mathcal{F}, F_i \notin \Phi(F_i)$.*

One simple and commonly used dependent function is: $\Phi(F_i) = \mathcal{F} - \{F_i\}$, that is, each field depends on all the other fields. Generally, this function works well. For some databases, a careful defined weighted dependent function may improve the performance in getting correct duplicate records.

With the weighted dependent function or dependent function, we can then compute the similarity for records with NULL values. Similar to Record Similarity, each field is identified as tokens by using a set of delimiters and tokens comparison is also the same. The difference is on how to compute the similarity for field with NULL value. In Record Similarity, similarity for NULL values is 0, while in Dynamic Similarity, similarity for NULL values is dynamically adjusted with its dependent fields and dependent weightages.

Field Similarity: Given a field F and two records X and Y, If $X_F \neq NULL \wedge Y_F \neq NULL$, then $Sim_F^{DS}(X,Y) = Sim_F^{RS}(X,Y)$. Otherwise, if Φ is a weighted dependent function, let

$$F' = \{(F_i, v_i) | (F_i, v_i) \in \Phi(F) \wedge X_{F_i} \neq NULL \wedge Y_{F_i} \neq NULL\}.$$

We have

$$Sim_F^{DS}(X,Y) = \sum_{(F_i,v_i)\in F'} (Sim_{F_i}^{DS}(X,Y) \times v_i) \tag{5.1}$$

If Φ is dependent function, i.e., $v_i = \frac{1}{|\Phi(F)|}$, then Formula (5.1) can be written as

$$Sim_F^{DS}(X,Y) = \frac{\sum_{(F_i,v_i)\in F'} Sim_{F_i}^{DS}(X,Y)}{|\Phi(F)|} \tag{5.2}$$

The field similarity is computed as follows. If the fields in two records are NOT NULL, we use the same method in Record Similarity to compute the field similarity. Otherwise, we compute it dynamically. In this case, the field similarity for these two records is computed from all the dependent fields that do not have NULL values, and the corresponding dependent weightages.

We have defined the (weighted) dependent function and shown how to use it to compute the similarity for NULL field. The idea behind it is from Functional Dependency (FD) in relational database normalization theory. Given a clean database and an FD $\mathcal{F}_1 \rightarrow \mathcal{F}_2$, where $\mathcal{F}_1 \subseteq \mathcal{F}$ and $\mathcal{F}_2 \subseteq \mathcal{F}$. For any records X and Y, if $X.\mathcal{F}_1 = Y.\mathcal{F}_1$, we have $X.\mathcal{F}_2 = Y.\mathcal{F}_2$, where $X.\mathcal{F}_1$ denotes the projection of record X onto the fields in $\mathcal{F}_1$. That is, the FD $\mathcal{F}_1 \rightarrow \mathcal{F}_2$ says that if two records agree on the values in fields $\mathcal{F}_1$, they must also agree on the values in fields $\mathcal{F}_2$. Thus, for a clean database, from FDs, we can determine the values of some fields from the values of some other fields. However, in this chapter, our discussion is on dirty databases and the values in some fields are likely having errors or missing. Therefore, we propose the (weighted) dependent function and field similarity, which says that if all fields in $\Phi(F_i)$ have large similarities, the similarity of F_i is to be large. From Formula (5.1), we can see that if $\forall F_j \in \Phi(F_i)$, $Sim_{F_j}(X,Y) = 1$, then $Sim_{F_i}(X,Y) = 1$. That is, if $X.\Phi(F_i) = Y.\Phi(F_i)$, then $X.\{F_i\} = Y.\{F_i\}$. Thus

(weighted) dependent function and field similarity are an extension of FDs with similarity.

When the similarities of all fields are computed, the Dynamic Similarity of records is obtained by:

$$Sim^{DS}(X,Y) = \sum_{i=1}^{n}(Sim_{F_i}^{DS}(X,Y) \times W_i) \tag{5.3}$$

Obviously, we have $0 \leq Sim^{DS}(X,Y) \leq 1$.

We adopt Record Similarity as the base of Dynamic Similarity because Record Similarity is an efficient comparison method. However, Dynamic Similarity can be extended to any other similarity-based comparison methods easily.

We have shown the method to compute the Dynamic Similarity, which can well deal with the fields with NULL values. Table 5.1 gives an example, which shows that correct duplicate records are obtained in Dynamic Similarity but missed in Record Similarity. The records are from a real database which is used in the performance study. Some fields' values are shortened to fit in one line. The field weightages are 0.05, 0.3, 0.15, 0.25, 0.05, 0.1, 0.1 respectively and the threshold is 0.73. These values are obtained by experimental tests on the database. In Dynamic Similarity, we use $\Phi(F) = \mathcal{F} - \{F\}$ as the dependent function.

In Table 5.1, the record X and record Y are duplicate and each has fields with NULL values. The similarity computed by Record Similarity is 0.7, which is less than the threshold 0.73. Thus they are missed by Record Similarity. However, with Dynamic Similarity, we have $Sim_{Name}^{DS}(X,Y) = Sim_{2nd}^{DS}(X,Y) =$

	Code	Name	1st Addr.	2nd Addr.	Cur.	Tel	Fax
X		JVC Elec.	79 AYE road	Ayer rajah ind., SG 139890	USD	7764711	
Y	JVC	JVC Elec.		Ayer rajah ind., SG 139890	USD	7764711	

Table 5.1: Correct duplicate records in Dynamic Similarity but not in RS.

$Sim^{DS}_{Cur}(X,Y) = Sim^{DS}_{Tel}(X,Y) = 1.0$. From Formula (5.2), we have $Sim^{DS}_{Code}(X,Y) = \frac{Sim^{DS}_{Name}(X,Y)+Sim^{DS}_{2nd}(X,Y)+Sim^{DS}_{Cur}(X,Y)+Sim^{DS}_{Tel}(X,Y)}{6} = \frac{4}{6} = 0.67$. Similarly, we have $Sim^{DS}_{1st}(X,Y) = Sim^{DS}_{Fax}(X,Y) = 0.67$. Thus from Formula (5.3), we can have $Sim^{DS}(X,Y) = 0.67 \times 0.05 + 1.0 \times 0.3 + 0.67 \times 0.15 + 1.0 \times 0.25 + 1.0 \times 0.05 + 1.0 \times 0.1 + 0.67 \times 0.1 = 0.9 > 0.73$. Then they are correctly obtained as duplicate records. An alternative solution is to decrease the threshold. For example, if the threshold is decreased to 0.7 for the above example, the two records can also be detected as duplicate by Record Similarity. However, this is far from a good solution since decreasing threshold obtained by experimental tests will largely increase the number of false positives. Another alternative solution is to treat two NULL values as equal, that is, $Sim_F(NULL, NULL) = 1$. Then the two records in Table 5.1 have similarity of $0.8 > 0.73$. They are obtained as duplicate records. However, treating two fields with NULL values as equal will also increase the number of false positives. Table 5.2 gives such an example. In Table 5.2, the two records are non-duplicate and each has fields with NULL values. If two fields with NULL values are

Code	Name	1st addr.	2nd addr.	Cur.	Tel	Fax
OMNI	Omni Elec.		#07-01/03, woodlands ave 5, SG	SGD		
OMNI-LD	Omni Elec.		lower delta road, #01-12/16, SG	SGD		

Table 5.2: False positives obtained if treating two NULL values as equal.

treated as equal, then the similarity computed by Record Similarity is 0.84, which is larger than the threshold 0.73. Thus they will be falsely obtained as duplicate with Record Similarity. With Dynamic Similarity, the similarity is $0.67 < 0.73$. They can be correctly detected as non-duplicate with Dynamic Similarity.

Dynamic Similarity depends on the dependent function. Therefore, we need to decide the dependent fields for each field. One option is that the domain expert (e.g., database designer) can provide this information. Another one is using the approximate functional dependence [HKPT98, KM95, KP96]. An approximate functional dependency is a functional dependency that almost holds. The approximate dependencies arise in many databases when there is natural dependency between attributes, but some rows contain errors (e.g., type errors, missing values etc.) or represent exceptions to the rule. Thus, it can be used by dynamical similarity to determine the dependent fields. The approximate dependency problem has been widely studied and [HKPT98] gives an efficient method.

5.3 Experimental Results

We test the performance on the real database, company; and four new synthetic databases, customers. The synthetic databases are generated by the database generator used in Section 3.6.3 but with an additional parameter, the number of field with NULL values.

We generate the customers as follows. We first generate a clean database with 100000 records. Each record consists of 8 fields: name, gender, marital status, race, nation, education, phone and occupation. Then we add additional 50000 duplicate records into the clean database. The changes in duplicate records range from small typographical difference in some fields to loss of values in some fields (NULL values). Basing on how many fields with NULL values, we generate four databases, customer-0, customer-1, customer-2 and customer-3, with average 0, 1, 2 and 3 fields with NULL values respectively.

We compare the performance of Record Similarity and Dynamic Similarity on the company database and the customer databases. The performance results are shown in Table 5.3 and Figure 5-1. The cleansing method we used is SNM and all results are obtained at the window size of 10. In Table 5.3, The "C" column under each method is the correct duplicate records obtained by that method. The "F. P." column under each method is the false positives obtained by that method. C_{DS} and C_{RS} denote the correct duplicate records obtained by Dynamic Similarity and Record Similarity respectively.

The results from all databases clearly show that Dynamic Similarity can ob-

Database	Record Similarity			Dynamic Similarity			
	Total	C	F. P.	Total	C	F. P.	$C_{DS} - C_{RS}$
company	52	51	1	57	56	1	5
customer-0	62192	62155	37	62192	62155	37	0
customer-1	61918	61881	37	62145	62118	37	237
customer-2	60630	60593	37	61940	61903	37	1310
customer-3	56732	56685	37	61389	61352	37	4667

Table 5.3: Duplicate pairs obtained by Record Similarity and Dynamic Similarity.

tain more correct duplicate records and does not introduce more false positives as compared with Record Similarity. For instance, in the company database, Record Similarity gets 51 correct duplicate records and introduces 1 false positive, while Dynamic Similarity gets 56 correct duplicate records and also introduces 1 false positive. There is 5 correct duplicate records increased. In the customer-3 database, Record Similarity gets 56685 correct duplicate records and introduces 37 false positive, while Dynamic Similarity gets 61352 correct duplicate records and also introduces 37 false positive. There is 4667 correct duplicate records increased. Furthermore, the results on customers show that when the average number of NULL fields increases, the Dynamic Similarity can get more correct duplicate records than Record Similarity does. As we can see, in customer-1, Dynamic Similarity get 237 more correct duplicate records, while in customer-3, Dynamic Similarity get 4667 more correct duplicate records.

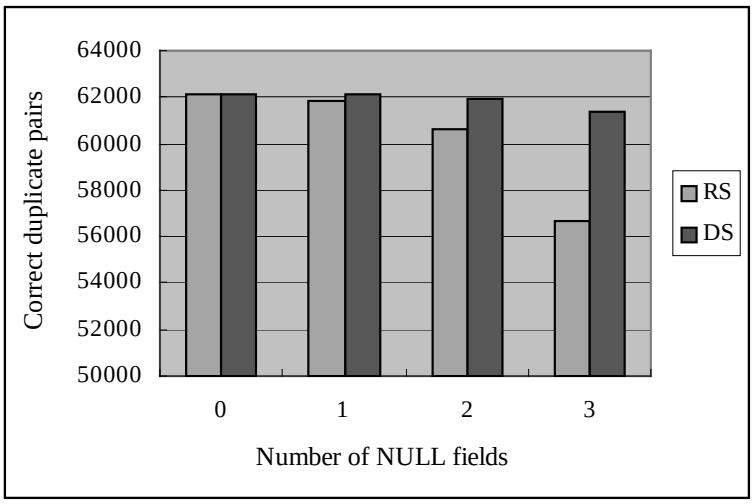

Figure 5-1: The number of Duplicates Per Record.

5.4 Summary

Existing comparison methods do not address the field with NULL value well, which lead to a decrease in the number of correct duplicate records. In this paper, we propose a simple yet efficient comparison method, Dynamic Similarity, which deals with fields with NULL values. As Dynamic Similarity is discussed on Record Similarity, it can be easily extended to any other comparison methods. Performance results on real and synthetic datasets show that Dynamic Similarity can get more correct duplicate records and does not introduce more false positives as compared with Record Similarity. Furthermore, the percentage of correct duplicate records obtained by Dynamic Similarity but not obtained by Record Similarity will increase if the number of fields with NULL values increases.

Chapter 6

Conclusion

6.1 Summary of the Thesis Work

In this thesis, we have studied several problems in data cleansing.

In Chapter 2, we first describes the research work that has been done in the data cleansing field. We focus our discussions on the data cleansing algorithms which are fundamental in all data cleansing. We also introduce other high level works, e.g., data cleansing language and data cleansing framework.

In Chapter 3, we propose two new efficient data cleansing methods, RAR1 and RAR2. We first discover two similarity rules, and show that a similarity method, LCSS, satisfies these rules. By employing these two rules efficiently, we propose these two methods which are much faster than existing methods.

In Chapter 4, we present a filtering scheme that further improves the result in Chapter 3. Since similarity methods are generally very costly, we then propose a

filtering scheme which runs very fast. Furthermore, the filter proposed satisfies the two similarity rules proposed in Chapter 3. Thus the new data cleansing methods proposed in Chapter 3 can be employed in our filtering scheme. However, the filter may produces some extra false positives. We introduce pruning with more trustworthy methods on the result obtained by the filter.

In Chapter 5, we propose a dynamic similarity method, which is an extension scheme for existing comparison methods. As existing comparison methods do not address fields with NULL value well, we then extend them by dynamically adjusting the similarity for field with NULL value. The idea behind dynamic similarity is from (approximate) functional dependencies.

6.2 Future Works

Some future research works are presented as follows.

All the existing detection methods and our methods proposed are "sorting and then merging" based. Although some methods differ on how the sorting is performed, the basic idea is the same. The "sorting and then merging" method is widely acceptable since that the merging phase is much more expensive than thee sorting (and clustering) phase. As shown in [HS95], any time advantage gained the sorting phase becomes small with respect to the overall time. However, since we have largely decrease the time (about one order of magnitude) on the merging phase, the time taken by sorting phase then cannot be ignored and may be worthwhile to decrease as well. Especially for very large databases that cannot

keep into memory, sorting results in $LogN$ scans on the databases, which may take longer time than the merging phase and then becomes the bottleneck. Thus, techniques that do not depend on the sorting (or partially depend on sorting) are worth to be addressed. One possible solution is to partition the whole database into small clusters (like clustering SNM). But this solution has its own drawbacks. First, the clustering itself may be costly, and secondly, clustering may largely decrease the number of duplicate pairs found.

Another issue need to addressed is on incremental cleansing. An incremental cleansing procedure is of practical importance since many commercial organizations periodically receive increments of data that need to be merged with previously processed data. Existing data cleansing methods assume the entire database is used for cleansing, and they do not attempt to use any previously gathered results in subsequent executions of the procedure, even if the procedure is run over data that has already been processed. Two strategies, called *Basic Incremental Merge/Purge Procedure (BIMP)* and *Increment Sampling Incremental Merge/Purge Procedure (ISIMP)*, have been proposed and evaluated in [Wal98]. These two strategies work better than normal data cleansing methods for this incremental cleansing problem, but they are not very efficient and there are still rooms for further improvement. Therefore, much better strategy would be discussed. Currently, we are considering this problem and a multi-level partition strategy is under development.

Bibliography

[AG87] A. Apostolico and C. Gueera. The longest common subsequence problem revisited. *Algorithmica*, pages 315–336, 1987.

[AJ88] R. Agrawal and H. V. Jagadish. Multiprocessor transitive closure algorithms. In *Proc. Int'l Symp. On Databases in Parallel and Distributed Systems*, pages 56–66, December 1988.

[BD83] D. Bitton and D. J. DeWitt. Duplicate record elimination in large data files. *ACM Transactions on Database Systems*, pages 8(2):255–265, 1983.

[Bic87] M. A. Bickel. Automatic correction to misspelled names: a fourth-generation language approach. *Communications of the ACM*, pages 30(3):224–228, 1987.

[BLN86] C. Batini, M. Lenzerini, and S. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.

[BM77] R. S. Boyer and J. S. Moore. A fast string-searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[CD97] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. In *ACM SIGMOD Record*, page 26 (1), 1997.

[CL92] W. I. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *CPM: 3rd Symposium on Combinatorial Pattern Matching*, pages 175–184, 1992.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[Coh98] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of the ACM SIGMOD International Conference on Managemnet of Data*, pages 201–212, 1998.

[DC94] M. W. Du and S. C. Chang. Approach to designing very fast approximate string matching algorithms. *IEEE Transactions on Knowledge and Data Engineering*, pages 6:620–633, 1994.

[dic] Dictionary of algorithms and data structures. `http://www.nist.gov/dads/`.

[DNS91] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation

of non-equijoin algorithms. In *Proc. 17th Int'l. Conf. on Very Large Databases*, pages 443–452, Barcelona, Spain, December 1991.

[FH99] E. J. Friedman-Hill. Jess, the java expert system shell, 1999. Available from `http://herzberg.ca.sandia.gov/jess`.

[For81] C. L. Forgy. Ops5 user's manual. Technical Report CMU-CS-81-135, Carnegie Mellon University, July 1981.

[GFS+01a] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. A. Saita. Declarative data cleaning: Language, mode, and algorithms. In *Proc. 27th Int'l. Conf. on Very Large Databases*, pages 371–380, Roma, Italy, 2001.

[GFS+01b] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. A. Saita. Improving data cleaning quality using a data lineage facility. In *Workshop on Design and Management of Data Warehouses (DMDW)*, Interlaken, Switzerland, June 2001.

[GFSS00] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. Ajax: An extensible data cleaning tool. In *Proceedings of the ACM SIGMOD International Conference on Managemnet of Data*, page 290, 2000.

[GG88] Z. Galil and R. Giancarlo. Data structures and algorithms for approximate string matching. *Journal of Complexity*, 4:33–72, 1988.

[GIJ+01] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukr-

ishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *Proc. 27th Int'l. Conf. on Very Large Databases*, pages 491–500, Roma, Italy, 2001.

[GP99] P. Gulutzan and T. Pelzer. *SQL-99 Complete, Really.* R&D Books, 1999.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[HD80] P. A. V. Hall and G. R. Dowling. Approximate string matching. *ACM Computing Surveys*, 12(4):381–402, 1980.

[Her96] M. Hernandez. *A generalization of band joins and the merge/purge problem.* PhD thesis, Columbia University, 1996.

[Hir77] D. S. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of the ACM*, 24:664–675, 1977.

[HKPT98] Yka Huhtala, Juha Karkkainen, Pasi Porkka, and Hannu Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *Proceedings of 14th International Conference on Data Engineering (ICDE)*, pages 392–401, Orlando, FL, 1998.

[HS95] M. Hernandez and S. Stolfo. The merge/purge problem for large

databases. In *Proceedings of the ACM SIGMOD International Conference on Managemnet of Data*, pages 127–138, May 1995.

[HS98] M. Hernandez and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, Vol. 2, No. 1:9–37, 1998.

[HU73] J. E. Hopcroft and J. D. Ullman. Set merging algorithms. *SIAM Jouanal on Computing*, 2(4):292–303, 1973.

[JTU96] P. Jokinen, J. Tarhio, and E. Ukkonen. A comparison of approximate string matching algorithms. *Software Practice and Expererience*, 26(12):1439–1458, 1996.

[JVV00] M. L. Jarke, M. Vassiliou, and P. Vassiliadis. *Fundamentals of data warehouses*. Springer, 2000.

[KCGS93] W. Kim, I. Choi, S. Gala, and M. Scheevel. On resolving schematic heterogeneity in multidatabase systems. *Distributed and Parallel Databases*, 1(3):251–279, 1993.

[Kim96] R. Kimball. Dealing with dirty data. *DBMS online*, September 1996. Available from `http://www.dbmsmag.com/9609d14.html`.

[KJP77] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[KM95] J. Kivinen and H. Mannila. Approximate dependency inference from relations. *Theoretical Computer Science*, pages 149(1):129–149, 1995.

[KP96] S. Kramer and B. Pfahringer. Efficient search of strong partial determinations. In *Proceedinngs of the Second International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 371–378, Portland, OR, August 1996.

[Kre95] V. Kreinovich. Strongly transitive fuzzy relations: An alternative way to describe similarity. *International Journal of Intelligent Systems*, 10:1061–1076, 1995.

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

[Lar] K. S. Larsen. Length of maximal common subsequences. Available from `http://www.daimi.au.dk/PB/426/PB-426.pdf`.

[Li92] W. Li. Random texts exhibit zipf's-law-like word frequency distribution. *IEEE Transactions on Information Theory*, 38:1842–1845, November 1992.

[Lim98] Infoshare Limited. Best value guide to data standardizing. *InfoDB*, July 1998. Available from `http://www.infoshare.ltd.uk`.

[LLL00] M. L. Lee, T. W. Ling, and W. L. Low. Intelliclean: A knowledge-based intelligent data cleaner. In *Proceedings of the sixth ACM SIGKDD in-*

ternational conference on Knowledge discovery and data mining, pages 290–294, 2000.

[LLL01] M. L. Lee, T. W. Ling, and W. L. Low. A knowledge-based framework for intelligent data cleansing. *Information System Journal - Special Issue on Data Extraction, Cleaning, and Reconciliation*, 26(8), 2001.

[LLLK99] M. L. Lee, H. J. Lu, T. W. Ling, and Y. T. Ko. Cleansing data for mining and warehousing. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications (DEXA)*, pages 751–760, 1999.

[LSQS02] Zhao Li, Sam Y. Sung, Xiao Y. Qi, and Peng Sun. Dynamic similarity for fields with null values. In *4th International Conference on Data Warehousing and Knowledge Discovery (DaWaK)*, pages 161–169, Aix-en-Provence, France, 2002.

[LSS96] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL - a language for interoperability in relational multi-database systems. In *Proc. 22nd Int'l. Conf. on Very Large Databases*, pages 239–250, Mumbai, 1996.

[LSSL02] Zhao Li, Sam Y. Sung, Peng Sun, and Tok W. Ling. A new efficient data cleansing method. In *13th International Conference on Database and Expert Systems Applications (DEXA)*, pages 484–493, Aix-en-Provence, France, 2002.

[Lyn88] C. A. Lynch. Selectivity estimation and query optimization in large databases with highly skewed distributions of column values. In *Proceedings of the 14th International Conference on Very Large DataBases (VLDB)*, pages 240–251, August 1988.

[MAZ96] M. Madhavaram, D. L. Ali, and Ming Zhou. Integrating heterogeneous distributed database systems. *Computers & Industrial Engineering*, 31(1-2):315–318, 1996.

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Jorunal of Algorithms*, 23(2):262–272, 1976.

[ME96] A. E. Monge and C. 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, 1996.

[ME97] A. E. Monge and C. P. Elkan. An efficient domain-independent algorithm for detecting approximately duplicate database records. In *Proceeding of the ACM-SIGMOD Workshop on Research Issues on Knowledge Discovery and Data Mining*, pages 23–29, Tucson, AZ, 1997.

[Mod] DataCleanser DataBlade Module. `http://www.informix.com/informix/products/options/udo/datablade/dbmodule/edd1.htm`.

[Mon97] A. E. Monge. *Adaptive detection of approximately duplicate database records and teh database integration approach to information discovery.* PhD thesis, Department of Computer Science and Engineering, University of California, San Diego, 1997.

[Mon00] A. E. Monge. Matching algorithm within a duplicate detection system. In *IEEE Data Engineering Bulletin*, volume 23(4), December 2000.

[Mon01] A. E. Monge. An adaptive and efficient algorithm for detecting approximately duplicate database records. *Information System Journal - Special Issue on Data Extraction, Cleaning, and Reconciliation*, 2001.

[Mos98] L. Moss. Data cleansing: A dichotomy of data warehousing? *DM Review*, February 1998. Available from `http://www.dmreview.com/editorial/dmreview/print_action.cfm?EdID=828`.

[MV93] A. Marzal and E. Vidal. Computation of normalized edit distances and applications. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 15(9):926–932, 1993.

[Nav97] G. Navarro. Multiple approximate string matching by counting. In *Proceedings of the 4th South American Workshop on String Processing*, 1997.

[QSLS03] Xiao Y. Qi, Sam Y. Sung, Zhao Li, and Peng Sun. Fast algo-

rithm of string comparison. *Pattern Analysis and Applications (PAA)*, 6(2):122–133, 2003.

[RD00] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. In *IEEE Data Engineering Bulletin*, volume 23(4), December 2000.

[RH01] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In *Proc. 27th Int'l. Conf. on Very Large Databases*, pages 381–390, Rome, 2001.

[Ril02] G. Riley. A tool for building expert systems, 2002. Available from `http://www.ghg.net/clips/CLIPS.html`.

[Ros98] S. Ross. *A First Course in Probability*. Prentice-Hall International, Inc., 1998.

[SJB96] W. W. Song, P. Johannesson, and J. A. Bubebko. Semantic similarity relations and computation in schema integration. *Data & Knowledge Engineering*, 19(1):65–97, 1996.

[SL02] Sam Y. Sung and Zhao Li. *Information Retrivial and Clustering*, chapter Clustering Techniques for Large Database Cleansing. Kluwer Academic Publishers, 2002.

[SLS02] Sam Y. Sung, Zhao Li, and Peng Sun. A fast filtering scheme for large database cleansing. In *Eleventh International Conference on Informa-*

tion and Knowledge Management (CIKM'02), McLean, VA, November 2002.

[SLTN03] Sam Y. Sung, Zhao Li, Chew L. Tan, and Peter A. Ng. Forecasting association rules using existing datasets. *IEEE Transaction on Knowledge and Data Engineering (TKDE)*, 15(6):1448–1459, 2003.

[SSLT02] Sam Y. Sung, Peng Sun, Zhao Li, and Chew L. Tan. Virtual-join: A query execution technique. In *21st IEEE International Performance, Computing, and Communication conference (IPCCC)*, Phoenix, Arizona, 2002.

[SSU96] A. Silberschatz, M. StoneBraker, and J. Ullman. Database research: Achievements and opportunities into the 21st century. In *SIGMOD Record (ACM Special Interest Group on Management of Data)*, page 25(1):52, 1996.

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

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

[Ukk92] E. Ukkonen. Constructing suffix trees on-line in linear time. *In Algorithms, Software, Architecture*, 1:484–492, 1992.

[Ukk95] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[VMA95] E. Vidal, A. Marzal, and P. Aibar. Fast computation of normalized edit distances. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 17(9):899–902, September 1995.

[Wal98] M. J. Waller. A comparison of two incremental merge/purge strategies. Master's thesis, University of Illinois, 1998.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Annual Symp. on Switching and Automata Theory*, pages 1–11, 1973.

[WF74] R. Wagner and M. Fisher. The string to string correction problem. *Jouanal of the ACM*, 21(1):168–173, 1974.

[WRK95] R. Y. Wang, M. P. Reddy, and H. B. Kon. Towards quality data: An attribute-based approach. *Decision Support Systems*, 13, 1995.

Appendix A Abbreviations

CDR	Candidate Duplicate Result
D-rule	If $L(A, C) \geq \sigma$, records A and C are duplicate
DR	Duplicate Result
ED	Edit Distance
L	$L_B(A, C) = Sim(A, B) + Sim(B, C) - 1$
LCS	Longest Common Subsequence
LCSS	Similarity method based on Longest Common Subsequence
LP	Lower Bound Similarity Property: $Sim(A, C) \geq L_B(A, C)$
ND-rule	If $L(A, C) < \sigma$, records A and C are not duplicate
RAR1	Reducing with one Anchor Record, a new detection method
RAR2	Reducing with two Anchor Records, a new detection method
RS	Record Similarity
SNM	Sorted Neighborhood Method
TC	Transitive Closure
TI	TI-Similarity, a simple yet fast similarity method
U	$U_B(A, C) = 1 - \|Sim(A, B) - Sim(B, C)\|$
UP	Upper Bound Similarity Property: $Sim(A, C) \leq U_B(A, C)$