# Cleansing Data for Mining and Warehousing*

Mong Li Lee    Hongjun Lu    Tok Wang Ling    Yee Teng Ko

School of Computing

National University of Singapore

{leeml, luhj, lingtw}@comp.nus.edu.sg

**Abstract**

Given the rapid growth of data, it is increasingly important to extract, mine and discover useful information from databases and data warehouses. The process of data cleansing or data scrubbing is crucial because of the "garbage in, garbage out" principle. However, "dirty" data files are prevalent as a result of incorrect or missing data values due to data entry errors or typographical errors, inconsistent value naming conventions due to different field entry format and use of abbreviations, and incomplete information due to unavailability of data. Hence, we may have multiple records refering to the same real world entity. This not only contributes to the problem of handling ever-increasing amount of data, but also leads to the mining of inconsistent or inaccurate information which is obviously undesirable. In this paper, we examine the problem of detecting and removing duplicating records. We present several efficient techniques to pre-process the records before sorting them so that potentially matching records will be brought to a close neighbourhood subsequently. These techniques include scrubbing data fields using external source files to remove typographical errors and the use of abbreviations, tokenizing data fields and then sorting the tokens in the data fields. We also propose a method to determine the similarity between two records. Based on these techniques, we implement a data cleansing system which is able to detect and remove more duplicate records than existing methods.

**Keywords:** Duplicates elimination, data mining, pattern matching, similarity measures

*A shorter version of this paper will appear in the 10th International Conference on Database and Expert Systems Applications (DEXA99), Florence, Italy, August 1999

# 1 Introduction

Organizations today are confronted with the challenge of handling an ever-increasing amount of data. In order to respond quickly to changes and make logical decisions, the management needs rapid access to information in order to research the past and identify relevant trends. These information is usually kept in very large operational databases and the easiest way to gain access to this data and facilitate strategic decision making is to set up a data warehouse. Data mining techniques can then be used to find "optimal" clusterings, or interesting irregularities in the data warehouse because these techniques are able to zoom in on interesting sub-parts of the warehouse.

Prior to the process of mining information in a data warehouse, **data cleaning** or **data scrubbing** is crucial because of the "garbage in, garbage out" principle. Therefore, one important task in data cleaning is to **de-duplicate** records. 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 or typing mistakes, (2) inconsistent value naming conventions because of different entry formats and the use of abbreviations such as 'ONE' vs '1', (3) incomplete information because certain data is not captured or available, (4) clients moving from one place to another with notifying change of address, etc, and (5) clients mis-spell their names or give false address (incorrect information about themselves).

As a result, we encounter situations where several records may refer to the same real world entity while not being syntactically equivalent. We can treat a set of records that refer to the same entity in two ways. We can view one of the records as correct and the rest of the records as duplicates containing erroneous information. Then the objective is to cleanse the database of the duplicate records [SSU95, HS95]. Alternatively, we can view each matching record as a partial source of information. Then the objective is to merge the duplicate records to obtain one record with more complete information.

In this paper we hold the latter view when we examine the problem of detecting and removing duplicating records. We present several novel techniques to pre-process the records before sorting them so that potentially matching records will be brought to a close neighbourhood subsequently. This will enable more matching records to be detected and removed. The pre-processing techniques include scrubbing data fields using external source files to remove typographical errors and the use of abbreviations, tokenizing data fields and then sorting the tokens in the data fields to solve the different field entry format problem which always exists in dirty data files but has been neglected by existing methods. We also introduce the use of field weightage to compute similarity among records. Accuracy is further improved with the help of external source files. Based on these techniques, we implement a data cleansing system which is able to detect and remove duplicate records than existing methods.

The rest of the paper is organized as follows. Section 2 gives a motivating example and surveys related works. Section 3 describes our proposed data cleansing methodology. Section 4 discusses the implementation and time complexity of our system, and finally we conclude in Section 5.

## 2 Motivation

In order to remove duplicated records from a dataset, the main consideration is how to decide that two records are duplicate? This involves comparing two records to determine their degree of similarity. which also implies that the corresponding fields in the two records has to be compared. The comparison of fields to determine whether or not two syntactic values are alternative representations of the same semantic entity is also known as the **field matching problem** [ME96].

| Record | EmpNo | Name | Address |
|---|---|---|---|
| 1 | 142625M | Liu Hang Xiang | 1020 Jalan Bandar Lamma, Industrial Park 3, West Malaysia |
| 2 | 142725M | Mr. Liu H.X. | Ind Park 3, 1020 Jalan Bandar Lama, Malaysia |

Table 1: Example of two duplicate records.

Table 1 shows two records, Record 1 and Record 2. At first glance, all the field values in both records look different. On closer examination, we note that the EmpNo of Record

1 and Record 2 are very similar except for one digit difference. We also observe that "Liu" is common in the Name field of Record 1 and Record 2. Furthermore, "H.X." in Record 2 seems to be an abbreviation of "Hang Xiang" in Record 1. If we reorganize the address of Record 2 as {*1020 Jalan Bandar Lamma, Ind Park 3, Malaysia*} we find that the addresses of Record 1 and Record 2 are actually the same except for the typographical error {*Lamma*} in Record 1 and the missing word {*West*} in Record 2. In addition, the abbreviation {*Ind*} has been used in Record 2 instead of {*Industrial*}.

Since the EmpNo, Name and Address field values of Record 1 and 2 are very similar to each other, we may conclude that Record 1 and Record 2 are most likely to be duplicates and they refer to the same employee in the real world. The differences in the Name and Address field values in Record 1 and 2 are typical of **different field entry format** problem.

There has been little previous research on the field matching problem although it has been recognized as important in the industry. Published work deals with special cases of the field matching problem such as the Smith-Waterman algorithm for comparing DNA and protein sequences [SW81], and variant entries in a lexicon [JR94]. [HS95] use domain specific equational axioms to determine if two tuples are equivalent. The closest work to ours is in [ME96] which gives a basic field matching algorithm based on matching strings and a recursive algorithm to handle abbreviations. However, the former algorithm does not handle abbreviation while the latter has quadratic time complexity.

The standard method of detecting **exact** duplicates in a database is to sort the database and then check if the neighbouring records are identical [BD93]. In order to detect approximate duplicates in a database, the most reliable way is to compare every record with every other record in the database. However, this is a very slow process which requires N(N-1)/2 record comparisions, where N is the number of records in the database. [HS95] proposed a **Sorted Neigbourhood Method (SNM)** to detect approximate duplicates by first sorting the database on a chosen application-specific key such as {Name, Address}. This key is a sequence of a subset of attributes, or substrings within the attributes, which has sufficient discriminating power in identifying likely candidates for matching. There is no rule specifying how the key should be designed. We can design a key such that it is a concatenation of the first 3 digits in EmpNo and the first 5 consonants in Name. As a result, "potentially matching" records are brought to within a close neighbourhood. Next, pairwise comparisons of nearby records are made by sliding a window of fixed size over the sorted database. Suppose the size of the window is w records, then every new record entering the window is compared with the previous w-1 records to find "matching records". The first record in the window slides out of the window (Figure 1).

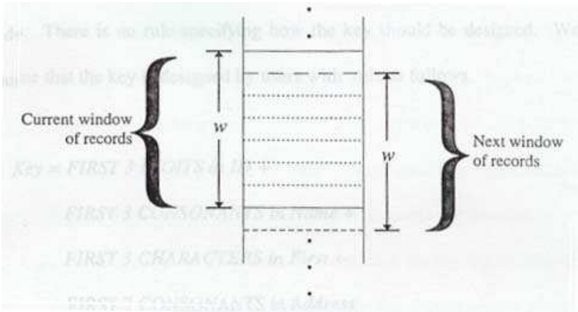


Figure 1: Window scan during data cleansing

This SNM is obviously faster since it requires only wN comparisons. However, the effectiveness of this approach depends on the quality of the chosen keys which may fail to bring possible duplicate records near to each other for subsequent comparison. For example, suppose we choose the Address field in Table 1 to be the key to sort the database. Then Record 1 and Record 2 will be very far apart after sorting because the address field value of Record 1 starts with "1020" while that of Record 2 starts with "Ind". On the other hand, if we choose the Name field to be the sort key, then Record 1 and Record 2 will be very close after sorting since both their name field values start with "Liu".

The **Duplication Elimination SNM (DE-SNM)** [H95] improves the results of SNM by first sorting the records on a chosen key and then dividing the sorted records into two lists: a duplicate list and a no-duplicate list. The duplicate list contains all records with exact duplicate keys. All the other records are put into the no-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 no-duplicate list and a second window scan is performed. These steps are summarized in Figure 2. However, the drawback of the SNM still persists in this DE-SNM.

An alternative to sorting the data is the **clustering method** which first partitions the database into independent clusters of approximately duplicate records. The SNM is then applied to each cluster independently and in parallel [H95]. Both the SNM and the

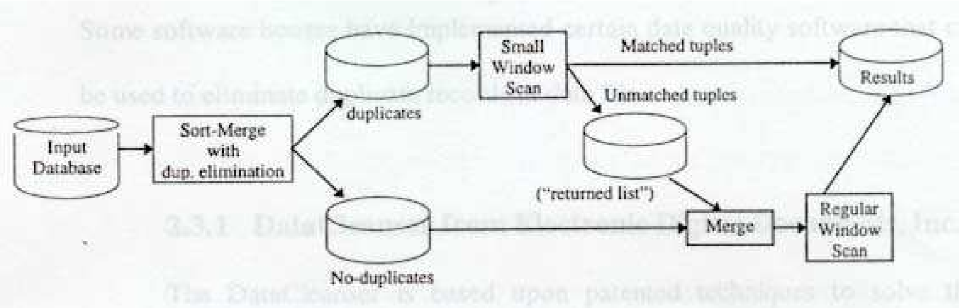


Figure 2: Duplicate Elimination Sorted-Neighbourhood Method

clustering method cannot achieve high accuracy without examining large neighbourhoods. In general, the duplicates elimination problem is difficult to handle both in scale and accuracy. Our proposed approach aims to increase the accuracy of the results by first pre-processing the records so that subsequent sorting will bring potentially matching records to a close neighbourhood. In this way, the window size can be reduced, thus improving the processing time.

Before we leave this section, we note that while there are a few data cleansing software in the industry, most companies do not disclose the details of how it's done.

# 3 Proposed Cleansing Methodology

Our approach to cleansing a database comprises of several steps.

1. **Scrub dirty data fields.** This step attempts to remove typographical errors and the use of abbreviations in the data fields. This will increase the probability that "potentially matching" records will be brought to within a close neighbourhood after sorting since sorting of the records uses keys which are extracted directly from these data fields.

2. **Sort tokens in data fields.** The characters in a string can be grouped into meaningful pieces. For example, the characters in a sentence can be grouped into meaningful words delimited by space. Therefore, string values in data fields such as Name and Address are split into meaningful groups, called **tokens**, which are then sorted. The objective

of this step is also to increase the likelihood of bringing matching records nearer during the subsequent record sorting.

3. **Sort records.**

4. **Comparison of records.** A window of fixed size is moved through the sorted records to limit the comparisons for matching records. We introduce the use of field weightage to compute the degree of similarity between two records.

5. **Merge matching records.** We treat each matching record as a partial source of information and merge them to obtain one record with more complete information.

Steps 1 and 2 are not found in existing cleansing methods. These additional steps enhance the possibility that matching records will be brought closer during the sorting. We will elaborate steps 1, 2 and 4 in the following subsections.

## 3.1 Scrubbing Dirty Data Fields

Existing data cleansing techniques such as the SNM and the DE-SNM are highly dependent on the key chosen to sort the database. Since the data is dirty and the keys are extracted directly from the data, then the keys for sorting will also be dirty. Therefore, the process of sorting the records to bring matching records together will not as effective. A substantial number of matching records may not be detected in the subsequent window scan.

Data in records are "dirtied" in various ways. It is common to find data entry errors or typing mistakes in name and address fields. Such **typographical errors** causes the data to be incorrect or contain missing values. In addition, these fields may have different **entry format** as illustrated in Table 1. **Abbreviations** are often used to speed up data entry. Therefore, the effectiveness of any approach to find duplicate records is to first remove such dirty data in these record fields.

Suppose we have a record with entry *ACER TECHNOOLGY PTE LTD* in its Company Name Field. There may be some typographical error in this field. Some typographical errors cannot be corrected by a spelling checker because special names such as the name of a person or a company cannot be found in any dictionaries. For example, *ACER* is not spelled wrongly because it is a company name but *TECHNOOLGY* has a typographical error. Abbreviations such as *TECH.* for *TECHNOLOGY* may also be used in another matching record. In order to ensure the correctness of data in the database, we use **external source files** to validate data and resolve any data conflicts. The external source files contain information in record format, that is, each record will have several fields as shown in Table 2. Such external source files can be obtained from National Registries such as the Registry of Birth, Registry of Companies etc, which would contain more accurate and complete information on a person or company.

| SSNO | Name | Age | Sex |
|---|---|---|---|
| 0273632T | Koh Yiak Heng | 43 | M |
| 3635290Y | Tan Kah Seng | 16 | M |
| 5927356K | Vivian Chua | 25 | F |

Table 2: Example of an external source file.

In Table 2, a particular person's information is contained in only one record. This external source file can be used to format and correct the information in a "dirty" database. We observe that there exists a functional dependency $SSNO \rightarrow Name, Age, Sex$ in our example external source file. $SSNO$ is unique and is called the **key** field. This feature in the external source file may be used to enforce any functional dependencies between the fields in the database. Note that fields in the source files should correspond to fields in the database and this correspondence have to be provided by users. Formatting of the fields in the "dirty" database will be carried out according to key field in the external source file. Table 3 shows an example "dirty" record in the database. During the scrubbing process, the system will find the SSNO of this record in the external source file (Table 2). It will then change the values of the Name and Age fields of of the "dirty" record (Table 3) to the corresponding field values of the equivalent record in the external source file (Table 2). Table 4 shows the cleansed record with the Name field value re-formatted and the Age value corrected. With this step, we can guarantee the correctness of data as well as standardize the entry format in the database.

If there exists some errors in the SSNO of the dirty database, then there are two possible

| SSNO | Name | Age | Sex |
|---|---|---|---|
| 0273632T | Koh Y.H. | 42 | M |

Table 3: "Dirty" record in the database.

| SSNO | Name | Age | Sex |
|---|---|---|---|
| 0273632T | Koh Yiak Heng | 43 | M |

Table 4: "Cleaned" record in the database.

scenarios:

1. The wrong SSNO does not exist in external source file.
   In this case, the system would inform the user of the error.

2. The SSNO is the SSNO of another person.
   In this case, the system should first calculate the similarity between the record in the database and those in the external source file. We develop a method to compute the similarity between two records by using field weightage. This method, which will be explained in section 3.3, can be used to calculate the similarity between a record in the database and a matching record in the external file. The field values in the database record will only be re-formatted or corrected if the computed similarity exceed certain value. Otherwise, the system would prompt the user whether or not to format the record in the database.

## 3.2 Tokenizing and Sorting Data Fields

We have seen in Section 2 how the key chosen for sorting the database records in the SNM plays an important role in bringing potentially matching records to within a window. This key can also cause the matching records to become further apart and hence reduce the effectiveness of the subsequent comparison phase. Table 5 shows three records in a database. Suppose we choose the Address field in Table 5 to be the key to sort the database. Then

Record 1 and Record 2 will be very far apart after sorting because the address field value of Record 1 starts with a numeric string "1020" while that of Record 2 starts with "Industrial".

We observe that the characters in a string can be grouped into meaningful pieces. For example, the characters in a Name or Address field can be grouped into individual words or numbers. We can often identify separate, possibly important, components within the field. Therefore, we can break a string up into pieces, called **tokens**, by using a set of delimiters such as space and punctuations (commas, full-stops, etc). In our approach, we first tokenise fields which has been chosen to be keys for sorting the database. The tokens within these fields are then sorted. For example, we can tokenise the Name and Address fields of the records in Table 5. So, for Record 1, we get the tokens {Liu Kok Hong} in the Name field. After sorting these tokens, we will obtain {Hong Kok Liu} for the Name field in Record 1. Table 6 shows the resulting database after tokenizing and sorting the tokens.

| Record | Name | Address | Sex |
|---|---|---|---|
| 1 | Liu Kok Hong | 1020 Jalan Bandar Lama, Industrial Park 3, Malaysia | M |
| 2 | Liu K.H. | Industrial Park 3, 1020 Jalan Bandar Lama, Selangor Darul Ehsan, Malaysia | M |
| 3 | Yap Kooi Shan | Blk 33 Marsiling Ind. Estate, #07-03, Singapore 130037 | F |

Table 5: Unsorted database

Sorting of records will now be based on the sorted tokens in the selected field. If the user chooses to use the Address field to sort the database, then the order of the records in the database will be 3, 2, 1. However, if the user selects the Name field to sort the database, then the order of the records in the database will be 2, 1, 3. Users can also choose to use {Name, Address} to sort the database. In this case, the system will make two pass on the database. It will first sort the records according to the tokenised and sorted Name field and remove any duplicate records. Then it will sort the database according to the tokenised and sorted Address field and remove any duplicate records. Note that information in the duplicate records are merged to obtain a record with more complete information.

| Record | Name | Address | Sex |
|---|---|---|---|
| 1 | Hong Kok Liu | 3 1020 Bandar Industrial Jalan Lama Malaysia Park | M |
| 2 | H K Liu | 3 1020 Bandar Darul Ehsan Ind. Jalan Lama Selangor | M |
| 3 | Kooi Shan Yap | 03 07 33 130037 Blk Estate Ind. Marsiling Singapore | F |

Table 6: Database with fields tokenised and sorted

We observe that in order for the sorting to bring possible matching records close to each other, users should choose fields which contains representative information of the record. For example, if users choose the Sex field to sort the database, then it is obvious that the sort will not be able to bring matching records close to each other since there are a lot of records containing same value in this field.

Note that there is a drawback in using the sorted tokens of one field to sort the database. For example, if the leading token of the Address field in Record 1 is missing in the database, then the leading token in the field will be 3 instead of 1020. If we only sort the records according to the Address field, then there is a high chance that Record 1 and Record 2 will be far away from each other since 3 and 1020 are not similar at all. In this situation, we will fail to detect that Record 1 and Record 2 are duplicates. This problem can be easily resolved by separating the numeric tokens from the string tokens. An alternative solution is to use sort fields so that if the missing leading token problem occurs in one sort field but not the other, the system will still be able to detect the duplicates.

## 3.3 Comparing Records

After the records in the database has been sorted, a window of fixed size $w$ is moved through the records to limit the comparisons of potentially matching records to those records in the window. Every new record entering the window is compared with the previous $w-1$ records to find matching records. The first record in the window slides out of the window.

The process of detecting duplicate records is very complicated. An efficient method is required to compare two records to determine their degree of similarity. In order to calculate the **degree of similarity** between two records, it is crucial to know that the importance of each field. Therefore, we introduce the concept of **field weightage**. For example, the Name field will definitely have a higher weightage than Sex field because name is obviously more representative of a record than sex. The field weightage of each field has to be provided by users and the sum of all field weightages should be equal to 1. For example, if the user want to eliminate duplicate records based on the Name and Address fields equally, then they should assign a weightage of 0.5 to each of these two fields and 0 for the other fields in the record. This indicates that records with same Name field and Address field will be matched. Alternatively, if the user want a database that contains unique names, then they should assign a weightage of 100 to the Name field and 0 for all other fields. This implies that matching of records will be based on the Name field only. That is, if two records have the same names, then they are considered duplicates regardless of the values in the other fields.

The process of computing the similarity between two records begins with comparing the sorted tokens of the corresponding fields. The tokens are compared using exact string matching, single-error matching, abbreviation matching and prefix matching. Based on the field token comparison results, the similarity between the entire field is computed. Finally, the record similarity can be computed from the fields similarity and the fields weightage. This is given in the following two propositions.

**Proposition: Field Similarity**

Suppose a field in Record X has tokens $t_{x_1}, t_{x_2}, ..., t_{x_n}$ and the corresponding field in Record Y has tokens $t_{y_1}, t_{y_2}, ..., t_{y_m}$. Each token $t_{x_i}, 1 \leq i \leq n$ is compared with tokens $t_{y_j}, 1 \leq j \leq m$. Let $DoS_{x_1}, DoS_{x_2}, ..., DoS_{x_n}, DoS_{y_1}, DoS_{y_2}, ..., DoS_{y_m}$ be the maximum of the degree of similarities computed for tokens $t_{x_1}, t_{x_2}, ..., t_{x_n}$, $t_{y_1}, t_{y_2}, ..., t_{y_m}$ respectively. Then the field similarity for Record X and Y $Sim_F(X, Y)$ is given by the expression $\frac{\sum_{i=1}^{n} t_{x_i} + \sum_{i=1}^{m} t_{y_i}}{n+m}$ .

**Proposition: Record Similarity**

Suppose a database has data fields $F_1, F_2, ..., F_n$ with field weightages $W_1, W_2, ..., W_n$ respectively. Given two records X and Y, let $Sim_{F_1}(X, Y), Sim_{F_2}(X, Y), ..., Sim_{F_n}(X, Y)$ be the field similarities computed. Then the record similarity for Record X and Y is given by the expression $\sum_{i=1}^{n} Sim_{F_i}(X,Y) * W_i$

We can have a rule that two records with record similarity exceeding a certain threshold such as 0.8 are duplicates and therefore, should be merged. While it is straightforward to check whether two tokens are exactly the same, it is not sufficient because of the existence of typographical errors, use of abbreviations etc. Therefore, we need to consider single-error matching, abbreviation matching and substring matching when comparing tokens to calculate the degree of similarity. If two tokens are an exact match, then they have a degree of similarity of 1. Otherwise, if there is a total of $x$ characters in the token, then we deduct $\frac{1}{x}$ from the maximum degree of similarity of 1 for each character that is not found in the other token. For example, if we are comparing tokens "cat" and "late", then $DoS_{cat} = 1 - \frac{1}{3} = 0.67$ since the character $c$ in "cat" is not found in "late" and $DoS_{late} = 1 - \frac{2}{3} = 0.33$ since the characters $l$ and $e$ are not found in "cat". We shall now elaborate on the various types of matching techniques and how the degree of similarity of the tokens are obtained.

1. **Exact string matching**

   This can be done using the standard *strcmp()* function. If two tokens are exactly the same, return 1, else return 0.

2. **Single-error matching**

   If two tokens do not match exactly, we will carry out single-error checking which includes checking for additional characters, missing characters, substituted characters and transposition of adjacent characters. Table 7 shows resulting degree of similarities when we compare the tokens "COMPUPTER", "COMPTER", "COMPUTOR", "COMPUTRE" to the token "COMPUTER". Note that the former tokens either have an additional character ("COMPUPTER"), a missing character ("COMPTER"), a different character ("COMPUTOR"), or a transposed character ("COMPUTRE") when compared to the latter.

3. **Abbreviation matching**

   An external source file containing the abbreviations of words is required. Table 8 shows an example abbreviation file. We observe that a token A is a possible abbreviation of token B only if all the characters in A are contained in B and these characters in A appear in the same order as in B. If a token is found to be an abbreviation of another,

| Token 1 | Token 2 | $DoS_{Token1}$ | $DoS_{Token2}$ |
|---|---|---|---|
| COMPUTER | COMPUPTER | 1.0 | 0.89 |
| COMPUTER | COMPTER | 0.88 | 1.0 |
| COMPUTER | COMPUTOR | 0.88 | 0.88 |
| COMPUTER | COMPUTRE | 1.0 | 1.0 |

Table 7: Single-error matching

then they have a similarity of degree 1. That is, this is an exact match. Note that if the abbreviation file is domain dependent and may not be complete in the early stage of database applications, then we may want to generate a list of ¡abbreviation, word¿ candidates from the given data set and ask the user to confirm the abbreviations.

| Abbreviation | Word |
|---|---|
| SVCS | Services |
| PTE | Private |
| LTD | Limited |

Table 8: Example of an abbreviation file

4. **Prefix substring matching**

   Here, we look for two similar tokens where one is a leading substring of the other. For example, "Tech." and "Technology", or "Int." and "International". Note that $DoS_{Tech} = 1$ since all the characters in "Tech." are found in "Technology" while $DoS_{Technology} = 0.4$ since there are 6 characters in "Technology" that are not found in "Tech". Note that if the substring does not occur at the beginning of a token, then the two token may not be too similar. For example, "national" and "international". In this case, we will give assign a similarity of degree of 0.0 for both these tokens.

We shall conclude this section by illustrating how we can compute the Address field similarity of Record 1(R1) and Record 2(R2) in Table 6.

1. Compare token *3* in R1 with token *3* in R2.

Degree of similarity for both these tokens is 1. Note that the degree of similarity for token *3* when compared with the rest of the tokens in R2 is 0. Similarly, the degree of similarity for token *3* when compared with the rest of the tokens in R1 is also 0. Therefore, $DoS_{3_{R1}} = 1$ and $DoS_{3_{R2}} = 1$.

2. Compare next token *1020* in R1 with token *1020* in R2.
   $DoS_{1020_{R1}} = 1$ and $DoS_{1020_{R2}} = 1$.

3. Compare *Bandar* in R1 with *Bandar* in R2.
   $DoS_{Bandar_{R1}} = 1$ and $DoS_{Bandar_{R2}} = 1$.

4. Compare *Industrial* in R1 with *Darul* in R2.
   Degree of similarity for both tokens *Industrial* and *Darul* is 0.

5. Compare *Industrial* in R1 with *Ehsan* in R2.
   Again, the degree of similarity for both tokens *Industrial* and *Ehsan* is also 0.

6. Compare *Industrial* in R1 with *Ind* in R2.
   *Ind* is a prefix substring of *Industrial*. Degree of similarity for *Ind* is 1 while the degree of similarity for *Industrial* is 0.3. When we compare *Industrial* with the rest of the tokens in R2, the degree of similarity for *Industrial* are all 0. Similarly, when we compare *Ind* with the rest of the tokens in R1, the degree of similarity for *Ind* are also all 0. After all these token comparison, we will take the maximum of the degree of similarities for *Industrial* and *Ind* as follows:
   $DoS_{Industrial_{R1}} = 0.3$ and $DoS_{Ind_{R2}} = 1$.

7. In the same way, we will obtain the following degree of similarities for the remaining tokens in the Address field of R1 and R2:
   $DoS_{Jalan_{R1}} = 1$; $DoS_{Jalan_{R2}} = 1$
   $DoS_{Lama_{R1}} = 1$; $DoS_{Lama_{R2}} = 1$
   $DoS_{Malaysia_{R1}} = 1$; $DoS_{Malaysia_{R2}} = 1$
   $DoS_{Park_{R1}} = 1$; $DoS_{Park_{R2}} = 1$
   $DoS_{Selangor_{R2}} = 0$

8. The Address field similarity for R1 and R2 is computed by adding up the degree of similarity *DoS* of **all** the tokens and then dividing by the total number of tokens. That

is, we have $F_{Address} = \frac{15.3}{19} = 0.81$

# 4 Data Cleansing System - Implementation and Performance

Based on the ideas and techniques described in the previous section, we implemented a data cleansing system in C on the UNIX. We tested our system with an actual company dataset of 856 records. Each record contains seven fields, namely, Company Code, Company Name, First Address, Second Address, Currency Used, Telephone Number and Fax Number. Manual inspection of the dataset reveals 40 duplicate records. Typical problems encountered in this dataset include records with empty Company Code field or Address field, matching records with different Company Code, records with names frequently mis-spelled and abbreviated. There are also different entry formats for the Address field and the separation of address into First Address and Second Address will cause a lot of matching problems. Table 9 lists the numbers of records in the dataset containing the various errors.

| Type of Error | Number of Records |
|---|---|
| Empty Company Code | 74 |
| Empty First Address | 436 |
| Empty Telephone Number | 477 |
| Empty Fax Number | 505 |
| Different entry format for the the address fields when combined | 422 |
| Typographical/Spelling | 389 |
| Abbreviations | 655 |

Table 9: Number of records with various errors in dataset

We note that the fields which contains representative information of a record and are most likely able to distinguish the records are Company Name, First Address and Second Address. These are the fields where typographical errors and use of abbreviations abounds. Therefore, the use of an abbreviation file to standardize the data will increase the degree of similarity between matching records. We observe that the fields First Address and Second Address should be merged since almost half the number of records have empty First Address. This will also overcome the problem of different entry formats [1] and facilitate matching of records when the address fields are used.

When we run our system, it is possible that duplicate records are not detected. It is also possible that similar records which do not represent the same real world entity are treated as duplicates. We call these incorrectly paired records **false-positives**. We obtain the following results when we run our system on the company dataset with a window size of 10:

1. **Misses.** The system failed to detect 5 individual records. That is, it has 12.5 % misses or 87.5 % true-positives.
2. **False-positives.** The system incorrectly matched 1 record. That is, it has 0.12 % false-positives.

The results show that our system is able to detect and remove the majority of the duplicate records with minimal false-positives. The additional pre-processing steps of scrubbing the data fields using external source files, tokenizing and sorting the data fields enables the subsequent sorting step to bring more potentially matching records to a close neighbourhood. In contrast to existing methods such as the SNM and DE-SNM, our method is not dependent on the key chosen to sort the records.

We conclude this section with a mathematical analysis of the time complexity of our system which shows that although these pre-processing steps may take extra time, they are not exponential.

Suppose we have $N$ records in a dataset and each record is of length $m$. Let $F$ be the number of fields in a record and $flen$ be the average length of a field.

1. Complexity for scrubbing the data fields using external source file.
   Let E be the number of records in the external source file. If we build an AVL tree to index the records in this file, then this scrubbing process will take $O(N * logE)$.

[1] The usual format for an address field is *building number, road name, unit number, postal code*

2. Complexity for tokenizing data fields and sorting the tokens.
   The complexity of tokenizing the fields in a record is $O(F * flen)$. If we use quicksort to sort the tokens, then the complexity of sorting the tokens is $O(F * t)$ where t is the average number of tokens in a field. Therefore, the total time complexity for tokenizing the data fields and sorting the tokens for all the records is $O(N*(O(F*flen)+O(F*t)))$. Since $t \ll flen$, the total complexity for this step is $O(N * F * flen) \approx O(N * m)$.

3. Complexity for sorting the records.
   Using quicksort, this step will take $O(N * logN)$.

4. Complexity for comparing records and removing duplicates.
   Let A be the number of records in the abbreviation file. Again, if we build an AVL tree to index the records in this file, then the process of checking for abbreviations in a record will take will take $O(F * t * logA)$ where t is the number of tokens in a field. The complexity of to merge duplicate records is $O(d * N * F * t * logA)$, where d is the maximum number of duplicates of one record.

From the above, we can see that the total time complexity of the whole system is $O(NlogN)$ if $(d * F * tlogA)$ is less than $logN$, and $O(d * N * F * tlogA)$ otherwise.

# 5 Conclusion

In this paper, we have examined the problem of detecting and removing duplicating records. We presented several efficient techniques to pre-process the records before sorting them so that potentially matching records will be brought to a close neighbourhood subsequently. These techniques include scrubbing data fields using external source files to remove typographical errors and the use of abbreviations, tokenizing data fields and then sorting the tokens in the data fields. We have shown that these pre-processing steps, which have been neglected by existing methods, are necessary if we want to detect and remove more duplicate records. We have also proposed a method to determine the degree of similarity between two records by using field weightage. Based on these various techniques, we implemented a data cleansing system to remove duplicate records with high accuracy. While the preliminary results obtained by our system has been encouraging, ongoing work involves testing the system's scalability and accuracy with real-world large data set.

# References


[BD93] D. Bitton and D.J. DeWitt. Duplicate record elimination in large data files. *ACM Transactions on Database Systems*, 1995.

[HS95] M. Hernandez and S. Stolfo. The merge/purge problem for large databases. *Proc. of ACM SIGMOD Int. Conference on Management of Data* pages 127-138, 1995.

[H95] M. Hernandez. A generation of band joins and the merge/purge problem. *Technical report CUCS-005-1995*, Department of Computer Science, Columbia University, 1995.

[JR94] C. Jacquemin and J. Royaute. Retrieving terms and their variants in a lexicalized unification-based framework. *Proc. of the ACM-SIGIR Conference on Research and Development in Information Retrieval* pages 132-141, 1994.

[ME96] A.E. Monge and C.P. Elkan. The field matching problem: Algorithms and applications. *Proc. of the 2nd Int. Conference on Knowledge Discovery and Data Mining* pages 267-270, 1996.

[SSU95] A. Siberschatz, M. Stonebraker, and J.D. Ullman. Database research: achievements and opportunities into the 21st century. A report of an NSF workshop on the future of database research *SIGMOD RECORD*, March 1996.

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