University of Windsor
Scholarship at UWindsor

Electronic Theses and Dissertations | Theses, Dissertations, and Major Papers

2002

Domain-independent de-duplication in data warehouse cleaning.

Ajumobi Okwuchukwu. Udechukwu
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Udechukwu, Ajumobi Okwuchukwu., "Domain-independent de-duplication in data warehouse cleaning." (2002). *Electronic Theses and Dissertations*. 1777.
https://scholar.uwindsor.ca/etd/1777

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

DOMAIN-INDEPENDENT DE-DUPLICATION IN DATA WAREHOUSE CLEANING

by

Ajumobi O. Udechukwu

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada

2002

National Library of Canada

Bibliothèque nationale du Canada

Acquisitions and Bibliographic Services

Acquisitions et services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-75854-0

Canada

Ajumobi O. Udechukwu 2002

© All Rights Reserved

ABSTRACT

Many organizations collect large amounts of data to support their business and decision-making processes. The data collected originate from a variety of sources that may have inherent data quality problems. These problems become more pronounced when heterogeneous data sources are integrated to build data warehouses. Data warehouses integrating huge amounts of data from a number of heterogeneous data sources, are used to support decision-making and on-line analytical processing. The integrated databases inherit the data quality problems that were present in the source databases, and also have data quality problems arising from the integration process. The data in the integrated systems (especially data warehouses) need to be cleaned for reliable decision support querying.

A major problem that arises from integrating different databases is the existence of duplicates. The challenge of de-duplication is identifying "equivalent" records within the database. Most published research in de-duplication propose techniques that rely heavily on domain knowledge. A few others propose solutions that are partially domain-independent. This thesis identifies two levels of domain-independence in de-duplication namely: domain-independence at the attribute level, and domain-independence at the record level. The thesis then proposes a positional algorithm that achieves domain-independent de-duplication at the attribute level. The thesis also proposes a technique for field weighting by data profiling, which, when used with the positional algorithm, achieves domain-independent de-duplication at the record level. Experiments show that the positional algorithm achieves more accurate de-duplication than the existing algorithms. Experiments also show that the data profiling technique for field weighting effectively assigns field weights for de-duplication purposes.

[Keywords: Data-warehousing, Heterogeneous databases, Multi-database systems, Data cleaning, Data cleansing, Data quality, De-duplication, Dirty data, Sorted neighborhood method, Merge/purge, Field matching, Object identity, Record linkage, Edit distance, Data integration]

To the memory of my father,
Albert Tagbo Udechuku, LLB (London), BL.
(1925 – 2000)

To my family with lots of love

ACKNOWLEDGEMENTS

Many thanks to my supervisor, Dr Christie Ezeife for the opportunity she gave me to study for this Masters degree. I also appreciate her direction of this thesis, and the generous research assistantships I received from her throughout my stay at Windsor. I wish to thank Drs Alioune Ngom, Ihsan Al-Aasm, and A. K. Aggarwal for serving on my thesis committee. I also wish to thank my brother, A. T. Udechuku, for his financial contributions to my study. I started my Masters program just after the death of my father, and my brother has been a father to the family since then.

My love goes to my mum, Mrs. Uzor Udechukwu, for the emotional cushion and love I've always received from her, and also to my other siblings Kene, Chinwe, Onyii, and Ikem. I'm glad I have a place I call home. I appreciate the friendship of Bunmi Fagbamiye, a most intriguing young lady who has added a lot of flavor to my life. I also wish to appreciate my cousins Okenwa Okoli and Nneka Ifeka for all their phone calls that gave me a sense of family in North America. Also, to all my friends (especially Malik Agyemang, Emmanuel Olusakin, Timothy Ohanekwu, Bola Falobi, and Samuel Igbokwe) and colleagues who made Windsor livable for me, I say thank you.

Finally, to Him who is able to do immeasurably more than I can ever ask or see, to the only wise God, and to my savior Jesus Christ, I give all the glory and praise.

TABLE OF CONTENTS

ABSTRACT		iv
DEDICATION		v
ACKNOWLEDGEMENTS		vi
LIST OF TABLES		ix
LIST OF FIGURES		x
1	INTRODUCTION	1
	1.1 An Overview of Data Warehousing	1
	1.2 Introduction to Data Cleaning	5
	1.3 Approaches to Data Cleaning	8
	1.4 Thesis Problem and Contributions	10
	1.5 Outline of the Thesis	11
2	PREVIOUS / RELATED WORK	12
	2.1 Duplicate Elimination	12
	2.1.1 Standard Duplicate Elimination Techniques	13
	2.1.2 Sorted Neighborhood Technique	13
	2.1.3 Duplicate Elimination Sorted-Neighborhood Method (DE-SNM)	15
	2.1.4 Duplicate Elimination with Pre-processing	19
	2.1.5 Adaptive Duplicate Detection Approach	20
	2.2 Field Matching and "Differ Slightly"	25
	2.2.1 The Basic Field Matching Algorithm	26
	2.2.2 The Recursive Field Matching Algorithm	27
	2.2.3 The Smith-Waterman Algorithm	30

2.2.4 The Field-Weightage Approach 32

2.2.5 String Matching with n-grams 33

2.3 Frameworks for Data Cleaning 34

2.3.1 ARKTOS: A Tool for Data Cleaning and Transformation 34

2.3.2 AJAX: An Extensible Tool for Data Cleaning 35

2.3.3 Potters Wheel: An Interactive Framework for Data Transformation and Cleaning 36

3 PROPOSED TECHNIQUE FOR DOMAIN-INDEPENDENT DATA CLEANING 38

3.1 The De-duplication Scheme 39

3.2 Positional Algorithm 41

3.3 Establishing the Match Threshold 48

3.4 Comparison of the Recursive Field-Matching Algorithm with the Positional Algorithm 50

3.5 Field Weighting with Data Profiling 51

4. EXPERIMENTATION AND EVALUATION 58

4.1 Experiments for Accuracy 58

4.2 Experiments for Response Time 63

5. CONCLUSIONS AND FUTURE WORK 65

REFERENCES 66

VITA AUCTORIS 72

LIST OF TABLES

1.1	An Example of Multiple-Source Data Quality Problem	9
2.1	A sample table of unclean data	17
2.2	A sample duplicates table in DE – SNM	17
2.3	A sample no-duplicates table in DE –SNM	18
2.4	A sample returned table in DE – SNM	18
2.5	A sample sorted table of unclean data	23
2.6	Example of basic field matching	27
3.1	Scheme for Scoring Field Weights	53
3.2a	A sample table of customer addresses in Ontario	56
3.2b	A sample table of customer addresses in Windsor, Ontario	56
4.1	Experimental Results on Test data 1	60
4.2	Experimental Results on Test data 2	62
4.3	Experimental Results on Response Time	64

LIST OF FIGURES

1.1	A simple data warehouse architecture	4
1.2	Classification of data quality problems in data sources	8
2.1	A sample unclean undirected graph in adaptive duplicate elimination	22
2.2	A sample cleaned undirected graph in adaptive duplicate elimination	25
3.1	Algorithm Main_De-duplicate	40
3.2	Algorithm Match_Records	41
3.3	Algorithm Match_Fields	45
3.4	Algorithm Match_Words	46
3.5	Algorithm Compute_Field_Weights	54
4.1	Percentage Precision versus Threshold on Test data 1	60
4.2	Percentage Recall versus Threshold on Test data 1	61
4.3	Percentage Precision versus Threshold on Test data 2	63
4.4	Percentage Recall versus Threshold on Test data 2	63
4.5	Response Time versus Number of Records	64

1. INTRODUCTION

The methods for collecting, storing, and processing data have changed drastically over the last two decades. The changes are driven by a number of factors, including gains in hardware processing power, reduction in relative hardware costs, and the emergence of the Internet. Many applications today need information from diverse sources, in which related data may be represented differently. This development also brings a need for the integration of different information sources into a unified schema, and the elimination of errors, duplications and redundancies. The application scenarios could be data warehouses, web-based global information systems, or federated information systems. However, data warehouses are of particular interest in this thesis. A data warehousing system is a single data store that integrates data from heterogeneous data sources like relational databases, multimedia databases, HTML documents, object–oriented databases, and others [Wi95a]. The integration effort usually requires that data from various sources be extracted, the source schemas transformed to the desired schemas, and the resulting database cleaned. Each of these complex processes has received considerable attention in the research community. However, this thesis focuses on data cleaning, proposing domain-independent approaches for removing duplicates in data-warehouse tables. The rest of this chapter will give an overview of data warehousing, introduce data cleaning, and present the various data cleaning problems.

1.1 An Overview of Data Warehousing

Data warehousing brings together selected data from component data sources, cleans it (e.g., by removing duplicated data), and transforms it so that it is suitable for end-user querying and analysis (e.g. for Online Analytical Processing – OLAP, Decision Support Systems – DSS, or Data mining). Formally, a data-warehouse is a subject-oriented, integrated, historical, and non-volatile collection of data used to support business decision-making [Ez01]. Data warehouses are subject-oriented because they are developed around major subjects (entities) of an enterprise (rather than around transactional functions), e.g., in a purchasing data warehouse for a retail company, the subjects would be: vendors, products, departments, regions, etc. Data warehouses store

historical data over a long period of time (typically up to ten years). They are also not updated frequently (i.e. non-volatile). Data warehousing provides a way for integrating various source databases originally stored as relational databases, flat files, HTML documents and knowledge bases, for decision support querying [Wi95a]. The front-ends to data warehousing systems (OLAP, DSS, Data mining) require complex volumes of data, making query response time, maintenance cost, and disk space utilization important warehousing issues [Ez01]. For example, a retailer (e.g., Staples Inc) that has chains of stores may want to keep an "Order" data warehouse to aid its ordering process and its interaction with its numerous vendors. The data warehouse in this example should be accessible to all the vendors (e.g., through the internet), and should also be accessible to the different departments of Staples Inc that may want to place purchase orders. Note that within Staples Inc, an item may be represented differently in the various departmental databases. For instance, Staples Inc sells personal computers. The Sales department may represent personal computers as Item No 001. Staples Inc also uses personal computers, therefore the Admin department will also have a way of representing personal computers in their fixed assets register (e.g., FA04). There is a marked difference in the accounting and reporting standards of a sale item and a fixed asset. The International Accounting Standards (IAS) requires that these two groups of items must not be grouped together in the company's records. In this scenario, there is a possibility that both the Sales and Admin departments may want to order personal computers from the same vendor at the same time. A way is needed to consolidate all the order information into a single data repository to enhance decision taking on vendor selection. This need is met by the "orders" data-warehouse.

For example, the management of a retail chain may be interested in viewing the number of times its suppliers (vendors) failed to deliver orders on time, or delivered faulty goods. This view could be for each vendor, for each department, each product, each region, quarterly. The view of interest could also be the total number for each vendor, annually, irrespective of the department, product, or region. Several "views" of the data stored in the data-warehouse may be viewed by management to aid their decision process. In this case, such decision could be related to vendor selection. The total possible number of

views of data stored in a data warehouse is determined by the design (schema) of the data warehouse. If the star schema is used, the possible number of views is determined as 2^n, where n is the number of attributes in the warehouse fact table. The star schema is a common relational data warehouse schema. In the star schema, the main integrated data is housed in a main warehouse table called the fact table. All attributes in the fact table, apart from the aggregation attributes of interest, are foreign keys. The star schema also includes a number of dimension tables that define values for the dimension attributes (i.e. the attributes represented by foreign keys in the fact table). In the example scenario we are considering, given that the database schemas for the Admin department (source database 1) and Sales department (source database 2) are:

Admin department: suppliers (sid, supplier-name, address)
assets (aid, asset-name, asset-category, asset-location, sid)

Sales department: vendors (vid, vendor-name, address, industry, rid)
products (pid, product-name, lead-time, vid)
region (rid, region-name)

An example corresponding fact table will be:

Fact (vid, pid, did, rid, time-d, delayed_goods, faulty_goods)

In the data warehouse fact table schema given above, the attributes of interest to management are quantities of delayed_goods (goods that were delivered later than the agreed date), and faulty_goods (goods that had to be returned to the vendor due to deficiency in quality). All the other attributes are foreign keys to the dimension tables. The dimension tables for our example will be:

vendors (*vid*, vendor-name, address, industry)
products (*pid*, product-name, lead-time)
department (*did*, department-name, location)
region (*rid*, region-name)
time (*time-d*, day, week, month, quarter, year)

The collection of aggregated views that are of interest to the organization is stored in the corporate data warehouse. The corporate data store can be complemented by an operational data store (ODS) which groups the base data collected and integrated from the sources. This hierarchy of data stores is a logical way to represent the data flow

between the sources and the data marts. In practice, all the intermediate states between the sources and the data marts can be represented in the same database. Summarily, data warehouses store historical, integrated, "subject-oriented" and summarized data of an establishment that aid online analytical processing (OLAP), decision support systems (DSS), and data mining. Conventional databases on the other hand store volatile, operational, "function-oriented" data that aid online transaction processing (OLTP). A simple data warehouse system architecture is presented in Figure 1.1.

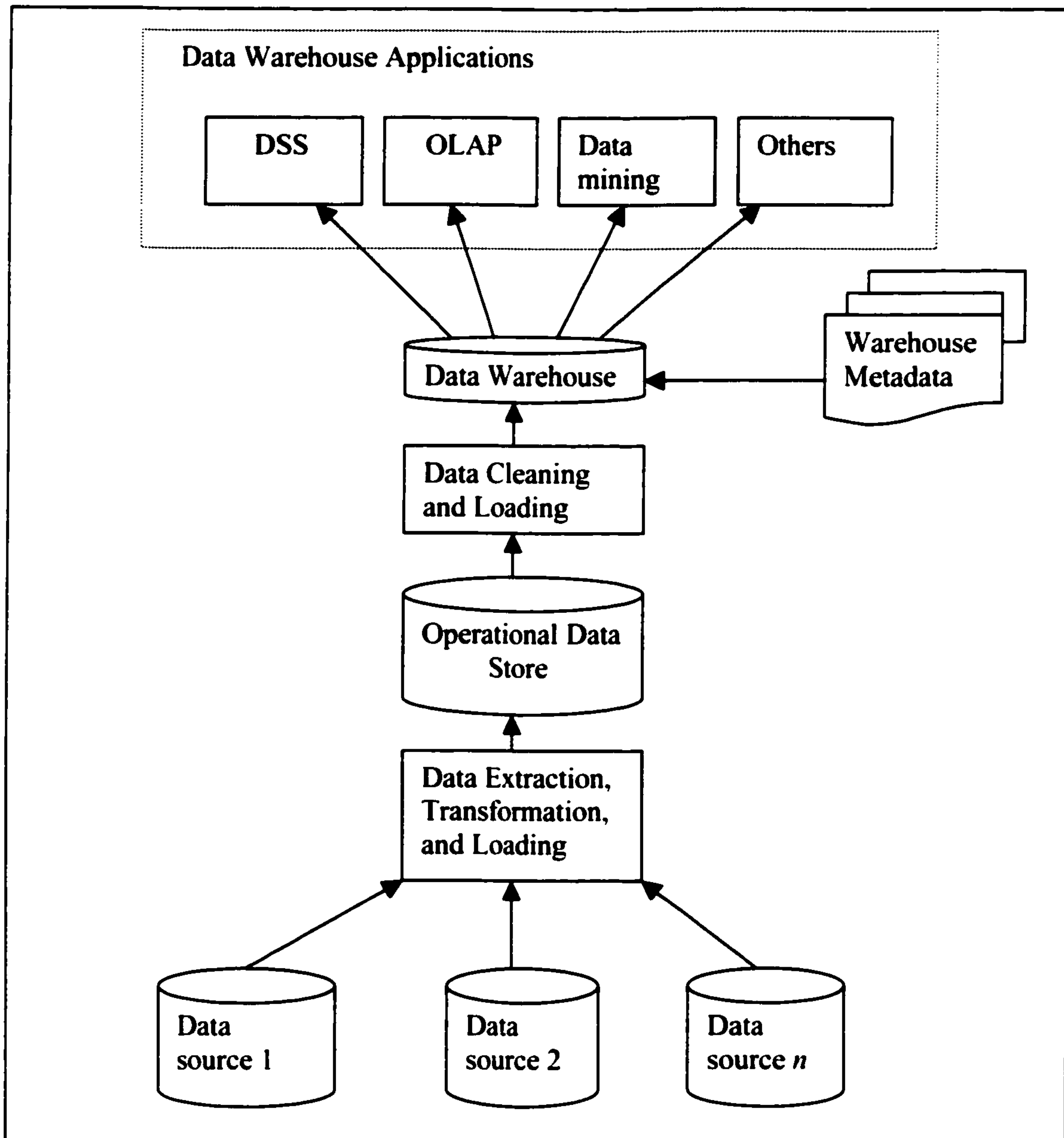

Figure 1.1 A simple data warehouse architecture.

The first level includes three major steps, which are:

(a) The extraction of data from the operational source databases. This includes changing the source data into the target structure (schema) of the warehouse. For example an address field in a source table may include street number and city, while the desired warehouse schema separates the street number from the city.

(b) The transformation of the data. The transformation stage involves cleaning of the data. This is mainly cleaning of the individual source databases. Some frameworks also allow for multi-table cleaning or multi-database cleaning [GFS+00c, RH01]. These frameworks are discussed in Section 2.3. The data from the various source databases are also integrated at this stage.

(c) The loading of the data, which is simply the transfer of the data to the operational data store.

The integrated data is held temporarily in the operational data store. Further cleaning steps are carried out (concentrating on errors resulting from integration), and the data is uploaded to the data warehouse.

1.2 Introduction to Data Cleaning

According to [MM00], there is no commonly agreed definition of data cleaning. The particular area of application determines the definition given to the process. Data cleaning is especially important in defining the processes for the following application areas: data warehousing, knowledge discovery in databases (also termed data mining), and data/information quality management (e.g. Total Data Quality Management – TDQM). "Data cleaning, also called data cleansing or scrubbing, deals with detecting and removing errors and inconsistencies from data in order to improve the quality of data" [RD00]. There are data quality problems existing in single data collections, such as files and databases. When these stand-alone sources are integrated (e.g., in data warehouses, federated database systems or global web-based information systems), the data quality problems are inadvertently escalated. Federated database systems, sometimes referred to as multi-databases systems, are distributed systems in which the data reside on stand-alone computers. The data in federated database systems are not pre-integrated, but are accessed when needed. Global web-based information systems are similar to federated

systems with the distinction that the stand-alone systems are geographically further apart, and are linked via the web. Data quality problems (also referred to as "dirty data") could be as a result of different data formats within distinct sources, data entry errors, or the unavailability of common keys among the different data sources to aid integration. Examples of dirty data include the following:

- Spelling, phonetic and typing errors (e.g., typing "Smit" instead of "Smith", or "Karl" instead of "Carl")
- Word transpositions (e.g., in a free-form text field for address, there could be two representations of the same address as follows: "Sunset Ave., 401" and "401 Sunset Ave.")
- Extra words (e.g., a name field may have the following two representations for the same name: "Miss Alice Smith" and "Alice Smith")
- Missing data (i.e., fields with null entries)
- Inconsistent values (e.g., having a character entry in a number field)
- Multiple values in a single free-form field (e.g., if a field for cities is free-form, there may be entries of the form "Windsor Ontario" in that field)
- Mis-fielded values (e.g., entering the last-name in the first-name field, and vice versa)
- Illegal values (e.g., having 13 in a month field, or 200 in a student-age field)
- Synonyms and nicknames (e.g., using both "csc" and "compsci" to represent "Computer Science" in different records of the data)
- Abbreviations, truncation and initials (e.g., entering "IBM" for "International Business Machines" or "CalTech" for "California Institute of Technology")
- Data that do not obey functional dependencies and referential integrity

A number of the instances listed above could be solved by using data integrity checks (e.g., having a minimum and a maximum possible data value), and by using more structured schemas (e.g., avoiding free-form fields and null values whenever possible). Others (e.g., spelling, phonetic and typing errors, abbreviations, etc), however, can only be solved by explicit cleaning routines.

A major consequence of dirty data is the existence of duplicates (i.e., multiple entries in the database – stand-alone or integrated, referring to the same real world entity). The removal of duplicates constitutes a major cleaning task. This is even more pronounced in data warehouses because they are used in decision support systems. Thus, data warehouses require and provide extensive support for data cleaning [CD97]. They load and constantly refresh very large amounts of data from heterogeneous sources so the probability that some of the sources contain "dirty data" is high. Due to the wide range of possible data inconsistencies and the large data volume, data cleaning is considered to be one of the biggest problems in data warehousing [RD00]. The data-cleaning step is usually part of the extraction, transformation and loading (ETL) process in data warehouse construction. A large number of tools of varying functionalities are available to support the cleaning step, but often a significant portion of the cleaning and transformation work has to be done manually or by hard-coded programs that are difficult to write and maintain [RD00]. Research on the subject however is tending towards greater automation of the cleaning process.

Generally, an approach for data cleaning should detect and purge all major errors and inconsistencies both in the individual source data and in the integrated database [RD00]. Data cleaning should also be supported by tools, which limit manual inspection and programming effort, and be extensible to easily handle new data sources. Data cleaning should be performed together with schema-related data transformations based on comprehensive metadata, and not in isolation. Figure 1.2 presents a classification of data-quality problems (requiring cleaning) based on data sources. The classification in Figure 1.2 bases the data-quality of a source on the degree to which it is structured by schema and integrity constraints controlling allowable data values. In sources without schema, such as files, there are few restrictions on what data can be recorded and stored, resulting in a high rate of inconsistencies and errors. However, for database systems, specific restrictions such as referential integrity are enforced. "Schema-related data quality problems, thus, occur because of the lack of appropriate model-specific integrity constraints, e.g., due to data-model limitations or poor schema design, or because only a few integrity constraints were defined to limit the overhead for the integrity control.

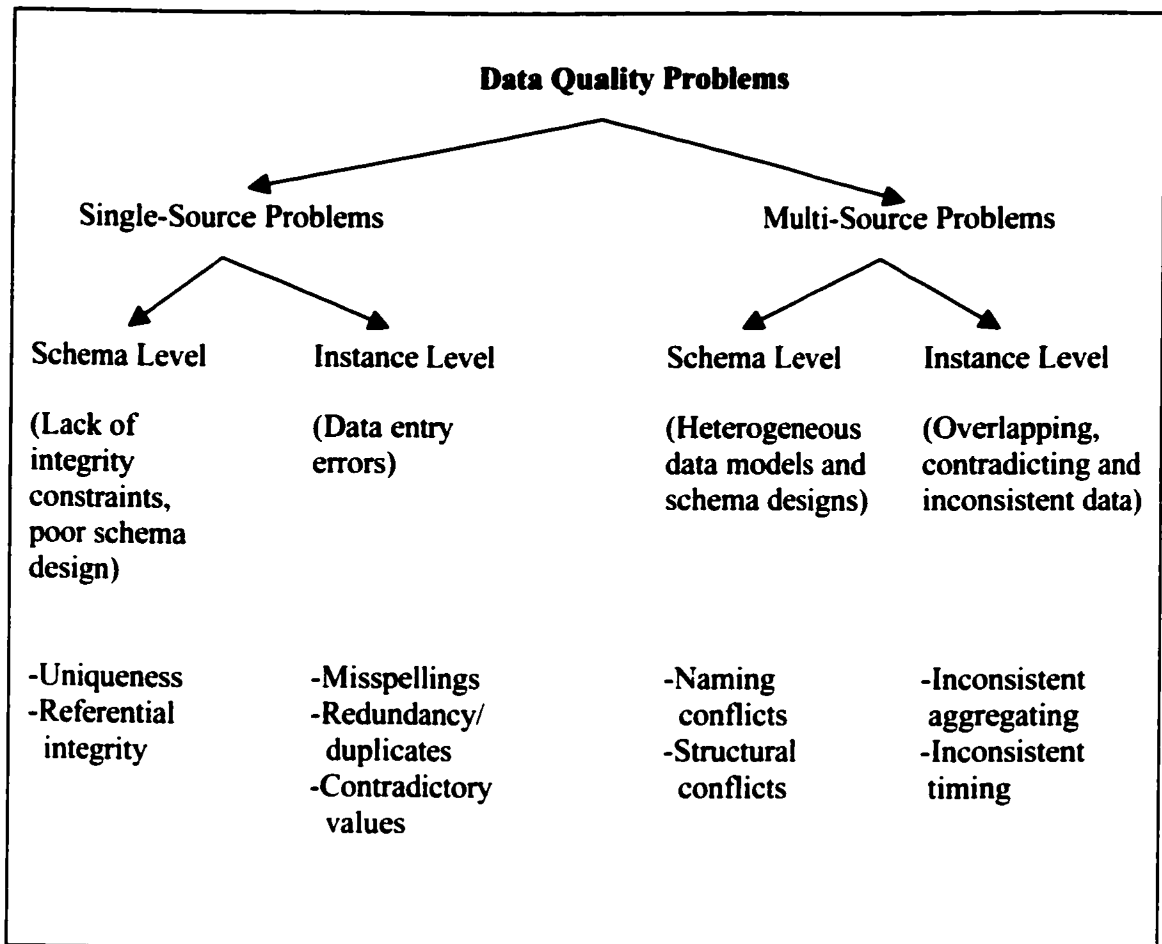

Figure 1.2. Classification of data quality problems in data sources [RD00]

Instance-specific problems relate to errors and inconsistencies that cannot be prevented at the schema level (e.g., misspellings)" [RD00]. An important data quality problem is the existence of duplicates. Duplicate records may exist in single-source data tables, or may result from the integration of multiple "clean" data tables. For example, in Table 1.1 the two source databases (Table 1.1a and Table 1.1b) do not contain any dirty data, but when these are integrated (as in Table 1.1c), some duplicated records result (records 1 and 4).

1.3 Approaches to Data Cleaning

Data cleaning should be viewed as a process. This process may be done during acquisition and definition of data, or may be applied to already acquired and defined data, to improve data quality in an existing system. Three phases for data cleaning are outlined in [MM00]. These are:

- Define and determine error types

CID	Name	Street	City	Sex
015	Kristen Smith	6 Felix Str	Windsor, ON N9C 3P4	0
021	Christian Smith	Felix St, 6	Windsor, ON	1
030	Jessica Parker	203 Campbell	Windsor	0

a: Clean data table from first source

Cno	LastName	FirstName	Gender	Address
55	Smith	Kris, L.	F	6 Felix St, Windsor, ON N9C 3P4
72	Forester	James	M	203 Mill Street, Windsor, ON

b: Clean data table from second source

No	Lname	Fname	Sex	Address
1	Smith	Kristen	F	6 Felix Str, Windsor ON N9C 3P4
2	Smith	Christian	M	6 Felix St, Windsor, ON
3	Parker	Jessica	F	203 Campbell, Windsor
4	Smith	Kris, L.	F	6 Felix St., Windsor, ON N9C 3P4
5	Forester	James	M	203 Mill Street, Windsor, ON

c: Integrated table from two clean tables, but with duplicates

Table 1.1: An Example of Multiple-Source Data Quality Problem

- Search and identify error instances
- Correct the uncovered errors

Each of the three phases is a complex task in itself. Several general steps for accomplishing the data cleaning process are listed in [RD00]. These include data analysis, definition of transformation workflow and mapping rules, verification, transformation, and backflow of cleaned data.

Data analysis is applied to the actual instances of data to obtain data characteristics or unusual value patterns. This is used to augment the metadata reflected in schemas in

determining the data quality of a source. Data analysis can also effectively contribute to the identification of attribute correspondences between source schemas (schema matching), based on which automatic data transformations can be derived [DDL00]. Data profiling and data mining are two related approaches to data analysis.

Data profiling emphasizes on the instance analysis of individual attributes. Information collected include data type, length, value range, discrete values and their frequency, variance, uniqueness, occurrence of null values, typical string pattern (e.g., the string pattern for phone numbers could be determined to be three digits, followed by a hyphen, then another set of three digits, followed by space, then a set of four digits). On the other hand, data mining helps discover patterns in large data sets, e.g., relationships existing between several attributes. These can be used to complete missing values, correct illegal values and identify duplicate records across data sources [SHM+99].

1.4 Thesis Problem and Contributions

This thesis addresses the problem of duplicate elimination in data-warehouse tables. Duplicates may exist in data-warehouse tables as a result of the data-quality problems discussed in Section 1.2. Current techniques used in solving the duplicate elimination problem in industry address specific problem domains (e.g., addresses in the United States). The work done by Monge and Elkan [ME96a, ME97] proposes techniques for achieving domain-independence in duplicate elimination. This thesis identifies two layers of domain-independence in duplicate elimination namely, duplicate-elimination at the attribute level and duplicate-elimination at the record level. The first contribution of this thesis is an enhancement to the recursive field-matching algorithm presented in [ME96a]. This enhanced recursive field-matching algorithm is termed the positional algorithm in this thesis. We show that the positional algorithm overcomes the major shortcomings of the recursive algorithm with respect to accuracy.

The second contribution of this thesis is a scheme for assigning weights (or importance) to the attributes of records in the data table independent of the problem domain. All previous work on field matching and data cleaning had relied on domain experts to assign

the importance of fields in the problem domain being considered. The problem with relying on domain experts is that they are not always available to the users of the data-cleaning program. The technique proposed in this thesis does not aim at totally replacing the domain experts, rather, the objective is to give the user an alternative in situations where domain-specific knowledge is unavailable or unreliable. The proposed technique uses data characteristics in assigning weights to the fields of the record for the purpose of identifying and matching duplicate records. This process is called data profiling.

We show that the proposed positional algorithm achieves domain independence at the attribute level, and when used with the field-weights assigned using data profiling, achieves domain independence at the record level. Finally, the thesis proposes a dynamic, domain-independent approach for determining matching thresholds for the purpose of duplicate elimination. Previous work in data cleaning suggest that domain experts set the match threshold experimentally. The details of these proposed techniques are given in Chapter 3 of this thesis.

Our experiments show that the positional algorithm returns more accurate results than the recursive algorithm. Our experiments also show that the data profiling technique used in this thesis effectively assigns field importance for the purpose of de-duplication. However, from our experiments, the threshold function discussed in this thesis fails to approximate the best thresholds for our experimental data. Details of the experiments are presented in Chapter 4 of this thesis.

1.5 Outline of the Thesis

In this chapter, an introduction of heterogeneous databases and data warehousing was presented. Chapter two presents a detailed discussion of existing algorithms and techniques for data cleaning. Chapter three presents the techniques used in this thesis for achieving domain-independence in data-warehouse duplicate elimination. In Chapter four, experimental results are discussed, and conclusions are drawn in Chapter five.

2. PREVIOUS / RELATED WORK

The data-cleaning problem is composed of several issues. Researchers in the data-cleaning field aim at solving one or more of these problems. The major problems in data cleaning are duplicate elimination, field matching, and user-machine interaction (or framework for data cleaning). This chapter surveys research in these three areas and presents work in each area in chronological order.

2.1 Duplicate Elimination

Duplicate elimination is also referred to as record linkage or record matching in research literature [Wi94, Wi95b]. The word record is used to refer to a syntactic representation of some real-world object, such as a tuple in a relational database. The duplicate elimination problem arises whenever records that are not identical, in a bit-by-bit sense, or in a primary key value sense, may refer to the same real life object [Mo00].

Almost all published work on duplicate elimination emphasize specific application domains, and propose algorithms that directly handle particular domains [HS95, ME95, He96, Hy96, ME96b, ME96c, HS98, MM00]. For example, some papers discuss duplicate elimination for customer addresses, census records, or variant entries in a lexicon. Other work on duplicate elimination are not domain specific, but assume that domain specific knowledge will be supplied by a human for each application domain [Mo97, LLL+99, LLL00, Mo00]. Thus, duplicate elimination algorithms vary by the amount of domain specific knowledge that they use. Many algorithms use pair-wise record matching algorithms that are domain specific [HS95, He96, Hy96, HS98]. They also use transformation rules that are based on domain-specific knowledge. Creating such rules can be time consuming and the rules must be continually reviewed whenever new data is added to the mix that does not follow the patterns by which the rules were originally developed [Mo00].

The rest of this section will review published work on duplicate elimination techniques. It is worth noting here that these techniques build upon each other. They all use some sort

of sorting (or clustering), and some sort of attribute comparison technique to determine the similarity between two records. The comparison techniques (also called field matching algorithms) are presented in Section 2.2.

2.1.1. Standard Duplicate Elimination Techniques

We use "standard' here to refer to all the earlier techniques developed to handle duplicate records in files. These techniques address the detection of both exact and approximate duplicates. The technique here is to use sorting to achieve initial clustering and then perform pair-wise comparisons of nearby records [NKA+59, BD83]. Exact duplicates are guaranteed to be next to each other in the sorted order regardless of which part of a record the sort is performed on.

In the case of approximate matches, there are no guarantees as to where duplicates are located relative to each other in the sorted order. In the best case, the approximate duplicate records may not be next to each other but will be close by. In the worst case they will be at opposite ends of the sorted table or lists. The results obtained will depend on the choice of the sort field, and errors present in the records. Thus, in order to match all possible duplicate records, every possible pair of records must be compared, leading to a quadratic number of comparisons. This approach is inefficient, and possibly infeasible in a typical data warehouse given that the records may be in the order of hundreds of millions.

2.1.2 Sorted-Neighborhood Technique

The sorted-neighborhood technique is an extension of the standard techniques adapted to handle the large numbers of records in modern-day databases. It aims at reducing the number of pair-wise comparisons performed in detecting duplicate records by comparing only records that are within a certain distance from each other [HS95]. The authors in [HS95] compare nearby records by sliding a window of fixed size over the sorted database. As a window of size W slides over the database one record at a time, the new record is compared with the other $W - 1$ records in the window. The size of the window is fixed, thus, as a record slides in, the uppermost record slides out of the window. Now

the number of record comparisons decreases from $O(T^2)$ to $O(TW)$ where T is the total number of records in the database and W is the number of records in the subset of the table being compared.

It is worth noting here that a tradeoff exists between the number of comparisons performed and the accuracy of the detection algorithm. The larger the value of W (i.e., number of records in the window), the better the system will do in detecting duplicate records. However, this also increases the number of comparisons performed and thus leads to an increase in running time. A way around the problem is to scan the records more than once but in a different order (i.e., sorted on another key) and apply the fixed windowing strategy to compare records and combine the results from different passes. This is called the multi-pass sorted neighborhood approach [HS98]. The authors in [HS98] reported that experiments showed that combining the results of several passes over the database with small window sizes yields better accuracy for the same cost than one pass over the database with a large window size.

Although the sorted neighborhood approach successfully adapts the standard approaches to handle large databases, it is still domain specific. For example, if our relational schema is ***R*** (firstname, lastname, address), the sorted neighborhood approach requires a key on which to sort the relation. The design of the key requires knowledge of the domain, and the fields that are more prone to errors. Secondly, the merging of records depends on predefined rules. The definition of these rules also requires domain expertise. For our example schema, we may have a rule as given below:

> Given two records, r_1 and r_2.
>
> IF the last name of r_1 equals the last name of r_2,
>
> > AND the first names ***differ slightly***,
> >
> > AND the address of r_1 equals the address of r_2
>
> THEN
>
> > r_1 is equivalent to r_2

The definition of "***differ slightly***" in the rule is based upon the computation of a distance function, which is compared with a predefined threshold. The selection of the distance

function and threshold are knowledge intensive. We will be discussing algorithms for calculating the distance function in Section 2.2.

2.1.3 Duplicate Elimination Sorted-Neighborhood Method (DE-SNM)

This approach is an improvement of the sorted-neighborhood approach [He96]. It has the following steps: Given a collection of two or more databases, concatenate them into one sequential list and perform the following steps.

1. Create Keys: A key is computed for each record in the list by extracting relevant fields or portions of fields.
2. Sort Data, Eliminating Duplicates: Sort the data in the data-list using the key created in step 1. Divide the sorted output into two lists – in the first list (called the "duplicates" list), put all records for which duplicate keys are detected (i.e., all records that share the same sort key with other records in the list). All other records (i.e., those with unique sort keys) are put in a second list called the "no-duplicates" list.
3. Sort the duplicates list: This step is necessary because the duplicates list is generated incrementally during the sort phase in step 2, and may not be in order.
4. First Window Scan: Move a "small" window through the sequential list of duplicate records limiting the comparisons to those records in the window having the same key. When the window gets full, the first record slides out of the window. Let u be the size of the small window. Every record that is about to enter this window either has the same key as all other records already in the window or its key is different from the keys of all the other records in the window. If the key of the new record is the same as the keys of the records in the window, then this new record is compared with all the current records in the window to find "matching" records (the matching uses the equational rules discussed in the sorted-neighborhood approach). On the other hand, if the key of the new record is not the same as the keys of all the other records in the window, then the following steps are followed:
    - Append to the "returned list" of records all previous records with the old key, that were not matched with any other record.

- For each group of matched records, append to the "returned list" the record that was matched the most with other records with the old key. This record will become the "prime representative" of its key in the later steps.
- Move the window $u - 1$ positions, making the new record the first one in the window.

5. Merge: Merge the "returned list" of records with the records in the "no-duplicates sorted list". An extra bit-field is added to the resulting sorted data to indicate whether a record came from the "returned list" or the "no-duplicates" list.
6. Second Window Scan: Move a fixed sized window through the sequential list produced in Step 5, limiting the comparisons for matching records to those records in the window. If 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 (as detected by the equational theory). If the record entering the window originated from the "returned list" (detected using the extra bit field), then it is compared only with records that did not come from the "returned list". The reason for this is that those records were already compared during the first window scan in Step 4 and were found to be "non-matching". On the other hand, a record coming from the "no-duplicates" list should be compared with all the w-1 previous records for it has not yet been compared to any record before. As in the sorted-neighborhood approach, the first record in the window slides out when a new record enters the window.

Table 2.1 shows an example of a data table that requires cleaning. Using the Duplicate Elimination Sorted-Neighborhood approach, the first step to cleaning the data in Table 2.1 is to select a sort key and sort the data. For example, the Last name attribute could be selected as the sort key. Once the table is sorted, it is divided into two tables. One table would contain all the records that have exact duplicates in the sort field. This table is called the duplicates table (Table 2.2). The second table would contain all the other records and is called the no-duplicates table (Table 2.3). Next, the duplicates table is sorted and a window size selected for traversing the duplicates table. For this example, let the window contain 3 records. The first three records in Table 2.2 are compared using an equational theory of rules (as with the sorted neighborhood method). All three records are

declared a match. A representative of these three matched records, customer 002, is entered into the returned table (Table 2.4). The fourth record enters the window, and the first record slides out of the window (which only takes three records). The new record is compared to the other two records in the window.

Customer_id	First name	Last name	City
001	Adams	White	London
002	Joseph	Smith	Kingston
003	Barry	White	Kingston
004	Joe	Smith	Kingston
005	A.	White	London
006	J.	Smith	Kingston
007	Jerry	Smith	Kingston
008	Adam	White	London
009	Adams	Whyte	London
010	Jerry	Goldsmith	London

Table 2.1: A sample table of unclean data.

Customer_id	First name	Last name	City
002	Joseph	Smith	Kingston
004	Joe	Smith	Kingston
006	J.	Smith	Kingston
007	Jerry	Smith	Kingston
001	Adams	White	London
003	Barry	White	Kingston
005	A.	White	London
008	Adam	White	London

Table 2.2: A sample duplicates table in DE – SNM.

Customer_id	First name	Last name	City
010	Jerry	Goldsmith	London
009	Adams	Whyte	London

Table 2.3: A sample no-duplicates table in DE – SNM.

Customer_id	First name	Last name	City
002	Joseph	Smith	Kingston
007	Jerry	Smith	Kingston
001	Adams	White	London
003	Barry	White	Kingston

Table 2.4: A sample returned table in DE – SNM.

This time, no match is found for the fourth record. Because the fourth record in the duplicates table couldn't be matched with any other record with its sort key, it is marked for entry into the returned-list table. Records marked for entry into the returned list may subsequently be unmarked if records that enter the window after them are matched with them. In such instances only a representative record of the matched group of records will be included in the returned-list table. Next, the fifth record in the duplicates table (with customer_id 001) enters the window. This time the sort key of the new record differs from the sort key of the records previously in the window. Thus, all the previous records in the window are removed, and all the records currently marked for inclusion to the returned list table are entered into the returned list table. For this example, only the fourth record in the duplicates table, with customer_id 007, is currently marked for inclusion into the returned list table. The sixth and seventh records in the duplicates table enter the window, and are compared. The sixth record (with customer_id 003) is not matched to any of the other two records, thus it is marked for inclusion into the returned list table. The other two records in the window match. The eight record in the duplicates table enters the window, and the fifth record leaves. The eighth record matches the seventh record. In this case, the fifth, seventh and eighth records were found to be matches. A representative record is chosen for this set of matched records (record five with customer_id 001), and is entered into the returned list. The eighth record for this example

is the last record in the duplicates table. Thus, all the records currently marked for inclusion to the returned-list table are entered into the returned-list table. In this case, only record six with customer_id 003 is marked for inclusion into the returned-list table (among the records with last name "White", which is the current sort key of the window being considered). Finally, the returned-list table (Table 2.4) is merged with the no-duplicates table (Table 2.3) and the traditional Sorted Neighborhood Approach applied to the resultant table (with the exception that records from the returned list are only compared to records from the no-duplicates list).

The duplicate-elimination sorted-neighborhood method (DE-SNM) was reported in [He96] to have performance gains over the multi-pass sorted-neighborhood approach.

2.1.4 Duplicate Elimination with Pre-processing

Duplicate elimination with pre-processing was proposed in [LLL+99] as an enhancement of the sorted-neighborhood approach. It aims at improving the accuracy of the results obtained using the sorted-neighborhood approach. Duplicate elimination with pre-processing has 5 basic steps namely:

i. Scrub dirty data fields
ii. Sort tokens in data fields
iii. Sort records
iv. Compare records
v. Merge matching records

Steps i and ii are enhancements to the sorted-neighborhood approach which increase the possibility that matching records will be brought closer during the sorting. Scrubbing dirty data fields is based on the use of "high integrity" external source files to validate the data and resolve any data conflicts. The records in the external source files are in the same format as those in the database being cleaned. Each field in the database being cleaned is validated against the external source database. There could also be dictionary and synonym look-ups for abbreviations.

The second step involves the tokenizing and sorting of the data fields. For example, if the entries in the address fields of two records are as follows: {301 Sunset Ave., Windsor} and { Sunset Ave. 301, Windsor}. Sorting on the address field will keep the two records very far from each other because one starts with a digit while the other starts with an alphabet. The proposed solution is to break each of the fields into its constituent components, sort the components before sorting the database based on the field. In our example, the two fields will yield an exact match i.e. {'301' 'Ave' 'Sunset' 'Windsor'}.

The rest of the steps in this approach are very similar to those in the sorted-neighborhood approach. The authors in [LLL+99] proposed the field-weightage algorithm for field matching, this is discussed in section 2.2.4. The authors however did not provide any experimental results to show the performance gains of the duplicate elimination with pre-processing over the sorted-neighborhood approach.

2.1.5 Adaptive Duplicate Detection Approach

This approach was introduced in [Mo00] as an enhancement to the sorted-neighborhood approach. Two major enhancements to the sorted-neighborhood approach were introduced. These relate to the implementation of the transitive closure of "is a duplicate of" matches, and the window size for comparisons. The enhancements are discussed in the paragraphs that follow.

The problem of detecting duplicates in a database can be described in terms of keeping track of the connected components of an undirected graph (assuming transitivity). "Let the vertices of a graph *G* represent the records in a database of size *T*. Initially, the graph will contain T unconnected vertices, one for each record in the database. There is an undirected edge between two vertices if and only if the records corresponding to the pair of vertices are found to match according to the pair-wise record-matching algorithm" [Mo00].

The second enhancement of the adaptive approach addresses the inherent weakness in the use of a fixed size window to scan the database. For instance, if a cluster in the database

has more duplicate records than the size of a window, then it is possible that some of these duplicates will not be detected because enough comparisons are not being made (for example, a record sliding into the window may have matched a record that has already slid out, with no comparable record still in the window). Furthermore, if a cluster has very few duplicates or none at all, then it is possible that comparisons are being done which may not be needed. This gives rise to the need for an approach that responds adaptively to the size and homogeneity of the clusters discovered as the database is scanned, in effect expanding/shrinking the window when necessary. The adaptive approach achieves this by replacing the fixed size window with a priority queue of duplicate records. The technique works as follows:

1. Sort the database on a chosen key, and create sets of clusters from the database
2. Set the first record as the representative for the first cluster in the priority queue
3. **Compare** the second record to the entry in the priority queue. **If** there is an exact match, **update** the graph (i.e. the graph structure representing the entire database is updated to show the link between the two records) and move on.

   If there isn't an exact match, perform an approximate match. **If** the result of the approximate match meets the matching threshold, update the graph and move on. **If** the result of the approximate match does not meet the matching threshold but falls short a little, perform an approximate match between the record and all the records in the cluster represented by the record in the priority queue. **If** a match is detected at any stage, update the graph. **If** the record being considered is a member of another cluster, merge the two clusters. **If** the record is sufficiently different from the record representing the cluster in the priority queue, then include this record as part of the representatives for that cluster in the priority queue.

   Else, if a match is not detected with any member of the set represented in the priority queue, then the record being considered must belong to a cluster that has

not yet been visited. **Therefore**, add the record to the priority queue as a representative of a new cluster, and give it the highest priority.

4. Take the next record in the database and repeat step 3. When the priority queue reaches its maximum size, the set of entries with the least threshold is removed from the queue (the cluster with the latest discovered member has the highest priority).

For example, using the table of unclean data shown in Table 2.1, the first step in the adaptive approach is to represent the entire database in an undirected graph as shown in Figure 2.1. The nodes of the undirected graph must uniquely represent records in the data table. For this example, the customer_id uniquely identifies the records in the data table, thus the customer_id numbers are used as the nodes of the undirected graph that represents the sample data table. The next step in this algorithm is to select a sort key for the data table and sort the table. For this example, the last name attribute is chosen as the sort key. The sorted data table is shown in Table 2.5. Next, sets of clusters are identified from the sorted data based on the sort key chosen (last name). In this example four clusters are identified, these are: records with last name "Goldsmith", records with last name "Smith", records with last name "White", and records with last name "Whyte".

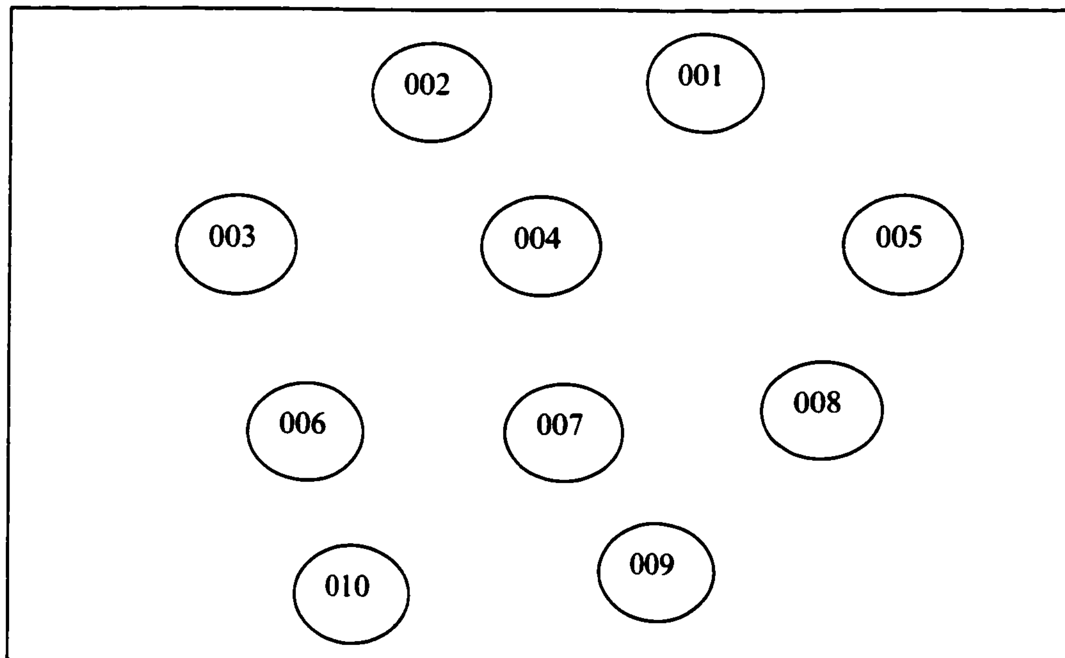

Figure 2.1: A sample unclean undirected graph in adaptive duplicate elimination

Customer_id	First name	Last name	City
010	Jerry	Goldsmith	London
002	Joseph	Smith	Kingston
004	Joe	Smith	Kingston
006	J.	Smith	Kingston
007	Jerry	Smith	Kingston
001	Adams	White	London
003	Barry	White	Kingston
005	A.	White	London
008	Adam	White	London
009	Adams	Whyte	London

Table 2.5: A sample sorted table of unclean data

The cleaning process is now started. At the start of the cleaning process, the priority queue (window) has no records. The maximum size of the priority queue is set by the user. For this example, the maximum size of the priority queue is chosen to be five records. The first record in the sorted data table (i.e. the record with customer_id 010) is entered into the priority queue. This record serves as the representative of its cluster in the priority queue, but in this case, the cluster has only one record. Next, the second record on the table (with customer_id 002) is compared with the record in the priority queue. The two records are first compared for exact match. The two records are not exact matches of each other, so they are compared for approximate match. The approximate match in this case also fails. Thus, the second record is included in the priority queue and serves as a representative of its cluster. The cluster that the second record belongs to has four records. These are records with last name "Smith" and customer_ids {002, 004, 006, 007}. Next, the third record in the data table is compared with the records in the priority queue. The priority queue at this point has two records {010 and 002}, with the latest entrant into the queue having the highest priority. An approximate match is found between the third record (i.e record with customer_id 004) and a record in the priority queue (with customer_id 002). Since a match is found, there are no additions to the

priority queue, and a link is created between nodes 002 and 004 in the undirected graph representing the database. Next, the fourth record in the data table is compared with the records in the priority queue. The priority queue still has two records {002 and 010}. Again an approximate match is found between the fourth element {006} and record 002 in the priority queue. A link is created between these two records in the undirected graph to show a match. Next, the fifth record in the table {007} is compared to the records in the priority queue. The priority queue still has two records {002 and 010}. First an exact match is checked for, but neither of the two records in the priority queue matches exactly with the new record. Next an approximate match is tried, and no match is found still. However, the approximate match between {007} and {002} is sufficiently high to require further investigation. Going by the adaptive algorithm, record 007 is compared with all the records in the cluster represented by record 002. This cluster has records {002, 004, 006 and 007}. Record 007 is compared with all the other records in the specified cluster, first for exact match, then for approximate match. An approximate match is found between record 007 and record 006. A link is created between records 006 and 007 in the undirected graph representing the data table. Also, going by the algorithm, we include record 007 in the priority queue. This is because the record that represents the cluster to which record 007 belongs to in the priority queue (i.e. record 002) differs sufficiently enough with it to require a second representative of that cluster on the priority queue. The process continues until the last record in the table is reached. The resultant undirected graph for the data table is shown in Figure 2.2. This graph has links between nodes representing records that have been identified as duplicates. All groups of nodes for which a link can be established in the undirected graph are collapsed into one record. Thus for this example, records represented by nodes 002, 004, 006, and 007 are collapsed into one record, and records represented by nodes 001, 005, 008, and 009 are collapsed into one record. Records represented by nodes 003 and 010 are left as independent members of the cleaned data table. The adaptive approach is reported to perform much fewer comparisons than the approaches discussed in [HS95] and [HS98] (as much as 75% savings), while maintaining the same level of accuracy [ME97, Mo00]. In [ME97] the authors achieved domain independence by adapting the Smith-Waterman algorithm

[SW81] to solve the field-matching problem. The Smith-Waterman algorithm is discussed in Section 2.2.3.

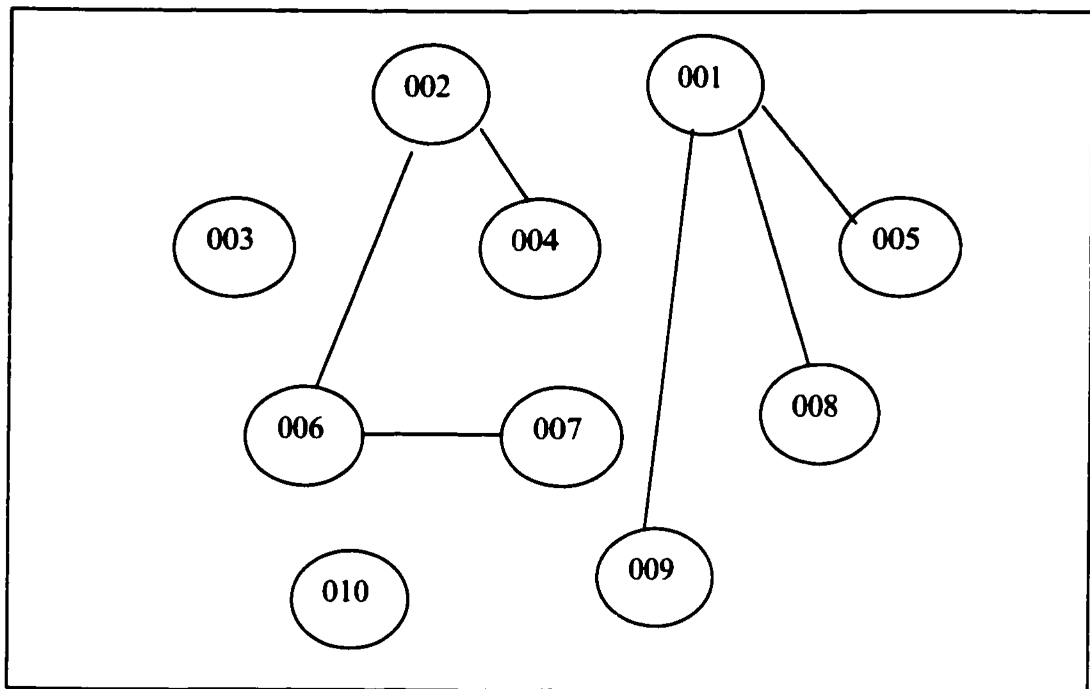

Figure 2.2: A sample cleaned undirected graph in adaptive duplicate elimination.

2.2 Field Matching and "Differ Slightly"

Field-matching algorithms are used to compute distance functions between two fields. The results usually range from 0.0 to 1.0 (zero for no resemblance and one for exact match). The input to a field-matching algorithm is the two fields being tested for semantic equivalence [ME96a]. In this section, published research work on field-matching algorithms are reviewed. The presentation takes a chronological order, going from the earlier basic approach to the more recent research work.

A related area of research to field matching is "approximate string matching" [Uk85, Ta94, BN99, Na01]. Approximate string matching is one of the main problems in classical string algorithms, with applications to text searching, computational biology, pattern recognition, etc. Unlike the field-matching problem where the focus is on

determining if two fields are similar, approximate string-matching aims to determine the occurrences of a given pattern or string in a much longer text.

2.2.1 The Basic Field Matching Algorithm

"An atomic string is a sequence of alphanumeric characters delimited by punctuation characters. A simple definition of the degree to which two fields match is the number of their matching atomic strings divided by their average number of atomic strings" [ME96a]. Stop words (e.g., *and, in, for, the, of, on, &, -, /,* etc) and punctuations are not included in counting the atomic strings. In the basic field-matching algorithm, two atomic strings match if they are the ***same*** string or one is a ***prefix*** of the other. For example, consider the two fields given below:

A = " Compt. Sci. Dept., University of Windsor, Ont, Canada" and

B = " Department of Computer Science, Univ. Windsor, Ontario"

The steps in the basic algorithm are

1. Break up each string into atomic strings, and remove the stop words and punctuations
2. Starting from the first atomic string in field A, compare each atomic string in A with all the atomic strings in B. A match occurs if an atomic string in A is the same as an atomic string in B, or one atomic string is a prefix of the other.
3. Compute the matching function as $K/((|A| + |B|)/2)$, where K is the number of matches of A in B, $|A|$ is the number of atomic strings in field A, and $|B|$ is the number of atomic fields in field B.

Table 2.6 shows the two strings and the matches found. Note that the basic field-matching algorithm does not take into account abbreviations that are not prefixes. Thus, for this example, the matching distance is:

$$4/((7 + 6)/2) = 0.62$$

The user of the cleaning program is expected to input the threshold distance to which the computed matching distance is compared [HS95, ME96a, HS98, LLL+99]. The choice of the threshold distance is subjective [BBD+00] and could be guided by an experimental application of the algorithm over a subset of the data, with the user investigating the performance of the algorithm using various threshold distances.

A	B	Match (Yes/No)	Comment
Canada		No	
Compt	Computer	No	Compt is not prefix to Computer
Dept	Department	No	Dept is not prefix to Department
ON	Ontario	Yes	Prefix match
Sci	Science	Yes	Prefix match
University	Univ	Yes	Prefix match
Windsor	Windsor	Yes	Exact match
7 atomic strings	6 atomic strings	4 matches	

Table 2.6: Example of Basic Field Matching

2.2.2 The Recursive Field Matching Algorithm

The recursive algorithm is an extension to the basic algorithm. In this algorithm, the recursive structure of typical textual strings is used. The base case is that A and B match with degree 1.0 if they are the ***same*** atomic string or one ***abbreviates*** the other; otherwise their degree of match is 0.0 [ME96a]. The desired base case has to be selected by the user, for example, the base case could be when the fields have been decomposed to their individual words, or it may be when each word in the field has been decomposed into its constituent characters. Every layer above the base case is taken as a sub-field. "Each sub-field of A is assumed to correspond to the sub-field of B with which it has highest score. The score of matching A and B then equals the mean of these maximum scores" [ME96a]. For example, given two strings, and assuming a character-level base case:

A = {Canada, Compt, Dept, Ont, Sci, Univ, Windsor}, and

B = {Computer, Department, Ontario, Science, University, Windsor}

The steps required in computing the matching distance are given below.

1. Take the first sub-field in A and break it into its individual alphabets.
2. For each sub-field in B, break the sub-field into its individual composing characters and obtain a match-distance by performing a modified field match with

the sub-field from A (i.e., match $(A_i, B_j) = K/ |A_i|$, where K is the number of constituent characters of Ai found in B_j).

3. Compute the matching distance of A and B as the mean of all the maximum match distances of the sub-fields.

Thus, for this example there are the following iterations:

$\mathbf{A}_i$ = "Canada" = {c, a, n, a, d, a }

$\mathbf{B}_j$ = "Computer" = {c, o, m, p, u, t, e, r}

K = 1 (i.e., 'c' is the only character or digit in "Canada" found in "Computer")

Match(Canada, Computer) = 1/ (|Canada|) = 1/6 = 0.17

[The recursive algorithm is not symmetrical, i.e., Match(Canada, Computer) is not equal to Match(Computer, Canada) which evaluates to 0.13. A two-way match could always be performed to determine the higher match, but this would increase the time complexity of the algorithm. Two-way matching is only applied at the base case in the specification of the recursive algorithm in [Mo97]. At the other layers, the words from the first field are simply matched against the words in the second field].

$\mathbf{B}_j$ = "Department" = {d, e, p, a, r, t, m, e, n, t}

K = number of constituent characters or digits in "Canada" that can be found in "Department", with repetitions allowed. Thus K = 5 (i.e., 'a','n','a','d','a'). Match(Canada, Department) = 5/ (|Canada|) = 5/6 = 0.83

The process is repeated for B_j = "Ontario", "Science", "University", and "Windsor". The maximum match score achieved is then assigned as the match score for "Canada".

The match score assigned to A_i = "Canada" is the highest match score it could achieve through all the B_j iterations. In this case, Score(Canada) = 0.83. The loop is then repeated for A_i = "Compt", "Dept", "Ont", "Sci", "Univ", and "Windsor". For each of the A_i's, all the B_j's are considered and the Score determined. The recursive matching distance is then computed as the mean of all the scores for the A_i's. For our example the scores will be:

A = {Canada, Compt, Dept, Ont, Sci, Univ, Windsor}

Score = { 0.83, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0 }

Recursive matching distance (with character base) = Total Match / |A| = 6.83/7 = 0.98

Recursive matching algorithm with character base tolerates most errors in words, however, its shortcoming is its high levels of false positives (e.g., matching "Canada" and "Department" with a score of 0.83).

The dissertation presented in [Mo97] experimented the recursive algorithm with word base. In this case, given two records of a data table to be matched, each field in the first record is compared to all the fields in the second record. The highest match score is assigned as the score for the field. In comparing two fields, each field is broken into its individual words. Each word in the first field is then checked for in the set of words of the second field, and a score of 1 is given if the word is matched, and 0 if otherwise. In the experiments reported in [Mo97], a two-way comparison is used to match the words. The two words being compared are not broken into individual characters, rather a match is declared if the two words are equivalent or one abbreviates the other. Word abbreviations used in the experiments were prefix or prefix/suffix combinations (i.e., "Ont" and "Ontario" will match, just as "dept" and "department" will match).

The recursive algorithm with word base does not suffer the excessive false-positive shortcoming of the character-base variant. However, the recursive algorithm with word base does not recognize many error types (e.g., "brain" and "brian" will not match). Given the two strings below, and using the recursive algorithm with word-base:

A = {Canada, Compt, Dept, Ont, Sci, Univ, Windsor}, and

B = {Computer, Department, Ontario, Science, University, Windsor}

All the words in A will be matched except "Canada" and "Compt", thus Match(A,B) = 5/7 = 0.71.

2.2.3 The Smith-Waterman Algorithm

The Smith-Waterman algorithm was originally developed for finding evolutionary relationships between biological protein or DNA sequences [SW81]. This algorithm was later adapted into an edit-distance algorithm for field matching in [ME96a], and proven to be domain-independent in [ME97] and [Mo97]. A field-matching algorithm is said to be domain independent if it can be used without any modifications in a range of applications. The Smith-Waterman algorithm was proven to be domain independent based on two assumptions: that records have the same high-level schema, and that records are made up of alphanumeric characters.

The Smith-Waterman algorithm has three main adjustable parameters. Given an alphabet-set of size W, the first parameter is a $|W|$ x $|W|$ matrix of the match scores for each pair of symbols in the alphabet. For example, if the 128 characters of the Unicode are being considered, a |128| x |128| array could be used to represent the matrix. Each cell in the matrix contains a score given when the two characters in the cell are the two characters being considered in the fields to be matched. A match could be exact, approximate, or no-match. The other two parameters required in the Smith-Waterman algorithm are penalty (or cost) for starting a gap and penalty (or cost) for continuing the gap. Gaps are simply differences between the two characters being compared. The changes could be as a result of mutations, insertions, and deletions. Thus, given two sequences, the Smith-Waterman algorithm uses dynamic programming to find the lowest cost of changes that can convert one sequence into the other. In the implementation of the Smith-Waterman algorithm presented in [ME96a], the authors defined an alphabet set consisting of the lower and upper case alphabetic characters, the ten digits, and three punctuation symbols: space, comma, and period. Best-fit scores were experimentally determined to be 5 for exact matches, 2 for approximate matches, and 0 for no-match. An exact match occurs when the two alphabets are the same or when there is only a case difference. The authors also set approximate matches to occur between two characters if they were both in one of the following subsets: {d t} {g j} {l r} {m n} {b p v} {a e i o u} {, .} .These subsets were determined by the authors from their observations and experience as the most likely characters easily mistaken in the English language. A penalty of –5 is set for starting a

gap (or change) and –1 for continuing the gap. For example, if two strings {dept} and {department} are to be compared, the emphasis will be to find the lowest cost of transforming the shorter string into the longer one. Note that the algorithm allows for unmatched characters as shown below:

D	e	p	a	r	t	m	e	n	t	
D	e	p							t	
5	5	5	-5	-1	-1	-1	-1	-1	5	= 10

We could have chosen to match 't' to the first occurrence of that character, but that would have been more expensive as shown below:

D	e	p	a	r	t	m	e	n	t	
D	e	p			t					
5	5	5	-5	-1	5	-5	-1	-1	-1	= 6

For this example, the first option is chosen, thus, there will be a score of 5 for the first position, 5 for the second, 5 for the third. A gap starts at the forth position because 't' is neither an exact match of 'a' nor an approximate match. Thus, a penalty of –5 is charged for starting a gap. Then 't' is compared to the next character in the second string, i.e., the fifth character which is 'r'. 't' again is neither an exact match nor an approximate match to 'r', therefore a penalty of –1 is charged for continuing the gap. The penalty of –1 is charged for all the character positions till the last position where 't' is matched with a score of 5. The total score awarded less the penalties is 5 + 5 + 5 – 5 – 1 –1 –1 –1 –1 + 5 = 10. The total score that would have been awarded if all the four characters in the shorter string had exact matches in the second string without gaps is 5 x 4 = 20. Therefore the match score for this example is 10/20 = 0.5

Though the Smith-Waterman algorithm is domain-independent, its implementation is language or alphabet dependent. This means that the sets of characters that could qualify as approximate matches in English may differ from those for French. A second limitation of the algorithm is that it does not handle instances where entries in a field are transposed, for example if the first-name is placed before the last-name in one field entry, and the last-name is placed first in the second field entry, the algorithm will perform very badly. Furthermore, the algorithm does not handle acronyms. For example if we have

"IBM" in one record's field, and "International Business Machines" in the second record's field. The penalty charges that will accrue will far outweigh the scores given for the matches. The result will be a no-match. Experiments performed in [ME96a] and [Mo97] determined that results from the Smith-Waterman algorithm were less accurate than those from the basic and recursive algorithms.

2.2.4 The Field-Weightage Approach

The field-weightage algorithm was proposed in [LLL+99]. The authors suggest that fields in the records to be compared be given scores to represent their relative importance in computing the degree of similarity. The user provides the field-weights, and the sum of all the field weights should be equal to 1. For example, if the data table has four fields {id, name, address, and sex}, we may decide that only the name and address fields determine if two records match. Thus, we will assign the name and address fields weights of 0.5 each, and assign a weight of 0 to the other fields. Note that we could have assigned weights of 0.6 and 0.4 to the two fields. The weights assigned depend on the user and his judgment of the importance of the fields in determining a match.

For every field in the record that is assigned a weight greater than zero, a similarity score is computed between the two records being compared. The approach proposes that the entries in the two fields be broken down into tokens and sorted. The tokens in a row are then compared to the tokens in the second row. 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 $1/x$ 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 "carly" and "karl", then Match(*carly*) = 1 – 1/5 – 1/5 = 0.6. We subtracted the first 1/5 (the denominator is the number of characters in "carly") because 'c' is not in "karl", and the second 1/5 because 'y' is not in "karl". Match(*karl*) = 1 – 1/ 4 = 0.75. The field similarity is then calculated as the sum of the match scores for all the tokens in the two rows being compared, divided by the number of tokens in both rows. The field similarity obtained for each field under consideration is multiplied by the assigned field-weight, and the resulting products summed to give the record similarity.

This technique is analogous to performing two-way recursive matches with character base (see section 2.2.2) on all the word-tokens of the two fields being compared, and taking the mean of the match scores. This method works well for single-error matching, prefix sub-string matching, and exact string matching. It however does not handle other forms of abbreviations. The authors propose that the data be preprocessed prior to the matching phase to eliminate acronyms and abbreviations. The preprocessing could be done through lookups to external files.

2.2.5 String Matching with *n*-grams

An *n*-gram is a vector representation that includes all the *n*-letter combinations in a string. The work presented in [Hy96] uses a trigram (or 3-gram) vector for string matching. Formally, a trigram vector $\hat{A}$ for a string *s* is defined as

$$\hat{A}_s = \{a_{aaa}, a_{aab}, \ldots, a_{d5f}, \ldots, a_{999}\},$$

Where a_{aaa} = number of times "aaa" appears in *s*.

Thus, the trigram vector for a string "computer" contains six components: "com", "omp", "mpu", "put", "ute", and "ter". The string comparison algorithm forms trigram vectors for the two input strings (excluding spaces and punctuations), and subtracts one vector from the other. The magnitude of the resulting vector difference is compared to a threshold value; if the magnitude of the difference is less than the threshold, the two strings are declared to be the same. In [Hy96], the threshold *T* used for comparing two strings with a total of *n* distinct trigrams was determined experimentally as $T = 2.486 + 0.025n$.

For example, given two strings:

s1 = "Machine Vision"

$\hat{A}_{s1}$ = {mac, ach, chi, hin, ine, nev, evi, vis, isi, sio, ion }

s2 = "Machien Vision"

$\hat{A}_{s2}$ = {mac, ach, chi, hie, ien, env, nvi, vis, isi, sio, ion}

Vector difference = $\hat{A}_{s1} - \hat{A}_{s2}$ = {hin, hie, ine, ien, nev, env, evi, nvi }

$$= \sqrt{(1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2 + 1^2)} = 2.828$$

$$\leq T = 2.468 + 2.025\,(15) = 2.861$$

Therefore, the two strings are declared a match. This approach however is language dependent, and requires domain knowledge to experimentally set the threshold.

2.3 Frameworks for Data Cleaning

Some recent research on data cleaning have focused on how best to integrate the user into the data cleaning process [HAC+99, RCH99, GFS+00a, GFS+00b, GFS+00c, VVS+00, GFS+01a, GFS+01b, RH01]. The emphasis of the research reported in these work was improving the usability of data cleaning systems. This could be achieved in a variety of ways, some of which are: building data cleaning operations into familiar commands (e.g. SQL or XML commands), thus enabling the user to issue simple SQL statements that would achieve extraction, integration, cleaning, and loading of source database tables into the target warehouse tables; providing an elaborate graphic user interface with the cleaning system to give the user a view of what is going on at the background (the user is also given control capabilities to undo any changes that may be undesirable); making the cleaning process dynamic and interactive. Earlier related research focused on integration frameworks for multi-database systems [PGU96, ZHK96, MZ98, CR99] or web data sources [CDS+98, Co98b]. These earlier research did not elaborate on the data cleaning problem nor its integration into their proposed frameworks. In the subsections that follow, three research prototypes that emerged from recent work on frameworks for data cleaning are discussed.

2.3.1 ARKTOS: A Tool for Data Cleaning and Transformation

The ARKTOS prototype was developed at the National Technical University of Athens, Greece to achieve three basic goals [VVS+00]. These goals are:

- graphical and declarative facilities for the definition of data warehousing transformation and cleaning tasks;
- measurement of the quality of data through specific quality factors; and

- optimized execution of complex sequences of transformation and cleaning tasks.

The prototype is modeled as units of activities peculiar to data integration and cleaning. The basic extensions of these activities are:

i. Error type: Grouped under this heading are Primary key violations, Reference violation, NULL value existence, Uniqueness violation, and Domain mismatch.

ii. Policy: these simply instruct the system on the next action to take for the detected error type. Examples include: ignore, delete row, report to a contingency file, and report to a contingency table.

iii. Quality factors: these commands are used to keep statistics of the cleaning process. Such statistics could be the number of rows deleted, the columns with the most errors, etc.

ARKTOS also has commands for data transformation and trace management (used to keep an audit trail of the activities performed during the cleaning process).

All these extensions are implemented and are made available to the user in three ways. These are:

- through a graphical user interface in which the user only needs to click on the appropriate buttons;
- declaratively as an XML variant for data warehouse processes, on the basis of a well defined DTD (Document Type Declaration). This XML variant and associated DTD, are provided with the system;
- declaratively as an extension to SQL, i.e., the extensions are coded into simple commands that the user can use in normal SQL statements.

2.3.2 AJAX: An Extensible Tool for Data Cleaning

The AJAX prototype, developed at INRIA Rocquencourt, France, aims at utilizing the inherent capabilities of database management systems in solving data integration and cleaning problems [GFS+00a, GFS+00b, GFS+00c]. Thus, in AJAX, the authors put extensions to SQL to handle activities peculiar to data integration and cleaning. These extensions were implemented in C language, and work as SQL commands. AJAX is modeled along three services. These are: data transformation; multi-table or approximate

join; and duplicate-elimination. The services each comprise of macro-operators each of which corresponds to an SQL-like command. The AJAX prototype allows the addition of new data-cleaning algorithms. These are treated as external functions and simply attach to the existing system. Thresholds for declaring duplicate tuples are graduated on a scale. Tuples with match scores above some given threshold are declared duplicates automatically while those with lower match scores are presented to the user for decision. The thresholds are specified with a 'decision' macro.

AJAX can handle both single-table and multi-table matching. For multi-table matching, the merge/purge problem is first solved in the individual tables to ensure record uniqueness, then a generalized join operation is performed over the *n* tables. The comparison is done two tables at a time. A similarity value between 0 and 1 is associated to every pair of tuples in the Cartesian product of the two tables being compared. The best match is chosen amongst all the scores, and a single table with unique, non-duplicate records results. For single-table matching, the prototype proposes the use of a nested loop in which each record is compared with all the records it encloses in the nested loop (this is a naive implementation of a Cartesian product of the table to itself). An enhancement to this is a hash join in which the records are only compared if a predefined condition is met. The condition is usually based on a relationship between the two strings being compared. The matching makes use of the usual edit-distance algorithms (the user is allowed to add any edit-distance function to the one available on the system).

2.3.3 Potters Wheel: An Interactive Framework for Data Transformation and Cleaning

Potters Wheel is a research prototype developed by researchers at the University of California, Berkeley. It is an interactive data cleaning system that tightly integrates transformation and discrepancy detection [RH01]. Users gradually build transformations by adding or undoing transforms, in an intuitive, graphical manner through a spreadsheet-like interface; the effect of a transform is shown at once on records visible on the screen (i.e., pipelined execution of the transformation and discrepancy detection phases). The system also allows the users to specify complex transformation rules by examples. The

structure from the example is then extracted (or inferred) and applied to the rest of the specified data.

The user is expected to specify the domain of each field in the records to be cleaned. Domains could be addresses, names, chemical formulae, etc. Each domain has a suitable algorithm provided in the system for discrepancy detection. The domains could be extended, and new algorithms added as the need arises. Discrepancy detection is done automatically in the background, on the latest transformed view of the data (based on the identified domains) and anomalies flagged off as they are detected. If a domain is not specified nor inferred for a field in the records being compared, a generic cleaning algorithm is applied. Thus, the system automatically parses a string into a structure composed of user-defined domains, and then applies suitable discrepancy detection algorithms.

3. PROPOSED TECHNIQUE FOR DOMAIN-INDEPENDENT DATA CLEANING

This chapter gives details of the proposed technique for achieving domain independence in data cleaning. The problem scope we address in this thesis is the elimination of duplicates in data-warehouse tables. The duplicate-elimination problem in data-warehouses can be solved by first eliminating duplicates from the warehouse dimension tables, and then reflecting the record matches in the warehouse fact table. Thus, if we remove all the duplicates in all the constituent dimension tables, we would have achieved de-duplication of the data-warehouse. This thesis proposes to solve this problem independent of the domain of the data stored in the data-warehouse.

For the purposes of this thesis, two levels of domain independence are identified namely: domain-independence at the attribute level; and domain-independence at the record level. Individual attributes of a record constitute data domains that could be cleaned differently. For example, a domain-specific algorithm could be developed to determine when two entries in the age field are equivalent or differ slightly. This could also be done for the other fields in the record. The work done by Hernandez and Stolfo in [HS95] and [HS98] uses this approach. Monge and Elkan in [ME96] and Monge in [Mo97] propose algorithms that could be used to determine field matches in two records irrespective of the domain of the field (i.e., the same algorithm can be used for the age field, the address field, the name field, etc). The work presented in [ME96] achieves domain independence at the attribute level. This was extended to the record level in [Mo97]. The work presented in [LLL+99] also achieves domain independence at the attribute level, but has many shortcomings (see Section 2.2.4). Experiments done by Monge and Elkan in [ME96], and Monge in [Mo97] show that the domain-independent techniques perform as well as the domain-specific techniques in identifying duplicates in the database. The experiments also show that the recursive algorithm returns more accurate results than the other proposed algorithms (see Section 2.2.2). This thesis proposes an enhancement to the recursive algorithm which overcomes some shortcomings in the algorithm presented

in [ME96] and [Mo97]. The proposed enhancement to the recursive algorithm is called the positional algorithm, and is presented in Section 3.2.

The second level of domain independence identified in this thesis is domain independence at the record level. The challenge here is identifying the fields in the database schema that should be given higher preference in determining matching records. For example, the last name attribute could easily be seen as being more important than the gender (sex) attribute in determining if two records in a given database table match. The approach used in commercial data cleaning packages and in most published research, is to have the user or a domain expert specify the levels of importance of the fields in the database. In cases where the packages or algorithms were developed for specific application domains (e.g., names and addresses within the United States), the field importance is hard-coded in the algorithms. In [Mo97], the scheme assigns equal weights to all the fields in the records being compared (except where any of the fields has a null entry, in which case that field is not counted as part of the fields in the records being compared). Such assignment of importance has its obvious shortcomings. This thesis proposes a premier approach to assign levels of importance (or weights) to the individual fields of the database for the purpose of record matching, thus, achieving domain independence at the record level (i.e., the technique will work irrespective of the database schema or the data stored in the data tables). To the best of the author's knowledge, no previous research had proposed an approach for assigning weights to the fields of the database for the purpose of duplicate elimination. The proposed scheme for field weighting is presented in section 3.5.

3.1 The De-duplication Scheme

It is worth noting at this point that this thesis does not propose any specific framework for data warehouse de-duplication. The contributions of this thesis are components that can be used with any de-duplication scheme to achieve domain-independence. However, for the purpose of presenting a wholesome view of the techniques proposed in this thesis, the sorted neighborhood approach is used – devoid of the windowing and multiple passes (see section 2.1.1). Given any data table, the following steps will be followed to generate

a de-duplicated table irrespective of the data domain (depicted algorithmically in Figure 3.1):

1. Profile the attributes of the data table, assign field weights, and select the fields to use in matching records based on the field weights (using the algorithm of Figure 3.5).
2. Starting from the first record, compare each record with all the records below it in the table (since no window is used to limit the number of comparisons). The algorithm used to determine if two records match is depicted in Figure 3.2. The record match algorithm takes in two records and compares the entries in their selected fields for matches using the positional algorithm (see Figure 3.3). The match score for each of the selected fields is multiplied by the respective field weight to get the field score. The sum of these field scores is compared to the record threshold to determine if the two records match. The scheme for determining the record threshold is discussed in section 3.3.
3. Merge identified duplicates in the data table.

```
Algorithm Main_De-duplicate (data table) returns de-duplicated_data_table
Input: The data table to be de-duplicated
Output: A de-duplicated data table
Begin
Set start_record = first record in data table
// make a call to algorithm in Figure 3.5//
Compute_Field_Weights (data table, start_record, Address_of_selected_fields)
For each record in the table, starting from the first
    Set record1 = the current record in the data table
    For each record below record1 in the data table
        Set record2 = the current record in the data table
        Match = Match_Records (record1, record2) // call to Figure 3.2 //
        IF (Match) Add record_ids for record1 and record2 to set of match pairs
    ENDFOR
ENDFOR
Scan through the set of match pairs and create a transitive closure of all match pairs
Merge each cluster/set of records into one record
Return table
END MODULE
```

Figure 3.1: Algorithm Main_De-duplicate

```
Algorithm Match_Records (record1, record2) returns match
Input: The selected fields of two records to be matched with their assigned field weights
Output: A Boolean match decision (match or no-match)
Begin
Set RecordMatchScore = 0, Set RecordThreshold = 0
For each selected field in the record
    1. Set String1 = Contents of that field in record1
    2. Set String2 = Contents of that field in record2
    3. Set FieldThreshold as variable to store field match threshold
    4. Set FieldWeight = Weight assigned to the field //an input to this module //
    // call the module that computes field matches in Figure 3.3 //
    5. Field_Match_Score = Match_Fields (String1, String2, FieldThreshold)
    6. Set FieldScore = FieldWeight * Field_Match_Score
    7. Add FieldScore to RecordMatchScore
    8 Set Threshold = Field_Weight * FieldThreshold
    9. Add Threshold to RecordThreshold
END FOR
IF RecordMatchScore >= RecordThreshold THEN
    1. Return Match = True
ELSE
    1. Return Match = False
ENDIF
END MODULE // end of the algorithm to match two records //
```

Figure 3.2: Algorithm Match_Records

3.2 Positional Algorithm

The positional algorithm is an enhancement to the recursive algorithm [ME96a, ME96c, Mo97] discussed in section 2.2.2. The basic idea used in the recursive algorithm (with character base) is to locate the characters that make up one entry in a field in the second entry. Thus, if there are two strings "dept" and "department" in two entries within a field, the algorithm checks for the existence of the characters that make up "dept" in "department". For this example, the result is an exact match, which is correct. However, there are instances where this approach fails. For example, given two strings which are entries in the name field of a database:

A = " John Johnson"

B = " Johns Smith"

Using the recursive algorithm with character base, the following steps would be taken to determine if the two strings are duplicates in the database.

1. Tokenize the strings. The resulting tokens are:

   A = {"John" "Johnson"}

   B = {"Johns" "Smith"}

2. Break up each token into its individual atomic characters, as follows:

   A = {" 'j', 'o', 'h', 'n' " " 'j', 'o', 'h', 'n', 's', 'o', 'n' "}

   B = {" 'j', 'o', 'h', 'n', 's' " " 's', 'm', 'i', 't', 'h' "}

3. Take each token in string 'A' and compare with each token in string 'B'. In each case, check for the constituent atomic characters in one token that are contained in the second token. The score is assigned as the ratio of the discovered characters to the length of the word. The highest score achieved for each word in string 'A' is taken to be the match score for that word.

In the example being used, "john" from string 'A' is compared to "johns" in string 'B' with a score of 1. Also, "john" from string 'A' is compared to "smith" in string 'B' with a score of 0.25. The higher score, which is 1, is taken as the match score for "john" in string 'A'. Going by the recursive algorithm, the second token in string 'A', which is "johnson", is compared with all the tokens in string 'B'. First, "johnson" is compared with "johns", resulting in a score of 1, then "johnson" is compared with "smith" with a score of 0.29. The higher of the scores, which is 1, is taken as the match score for this token. Taking the average of the match scores for string 'A', a match of 1 (exact match) is arrived at in error. This is the first shortcoming of the recursive algorithm solved in this thesis.

A second shortcoming of the recursive algorithm with character base is shown in the following example. Given two strings:

A = "Tims"

B = "Smith"

Going by the recursive algorithm with character base, a match is found if all the characters that constitute a token in one string are found in the second string. In this example, all the individual characters that constitute string 'B' can be found

in string 'A', thus an exact match is declared in error. This shortcoming is addressed in the positional algorithm.

When using the recursive algorithm with word base, the first shortcoming discussed above is still present (i.e., when A = {"John" "Johnson"} and B = {"Johns" "Smith"}). The second shortcoming is not encountered because the words are not broken into characters. However, the recursive algorithm with word base is very rigid and does not tolerate errors in words (as discussed in section 2.2.2). Furthermore, the design of the recursive algorithm (both word-base and character-base) makes it difficult for acronyms to be matched. The positional algorithm is designed to overcome the shortcomings of both versions of the algorithm. The positional algorithm tolerates errors in words while maintaining reasonable accuracy levels, and also supports the matching of acronyms.

The steps in the positional algorithm are outlined below.

1. Tokenize the two strings being compared into words. Denote the string with fewer word-tokens as the first string, and the other string as the second (lines 7 to 12 in Figure 3.3).
2. If one of the strings has only one word-token and the other string has more than one word-token (e.g., A = "IBM" and B = {"International", "Business", "Machines"}), then, break up the word-token in the string with one token into its constituent atomic characters (i.e., A = {"I", "B", "M"}). If the number of characters in 'A' is equal to the number of words in 'B', then compare each atomic character in the string with one word-token, with the first character of the word-tokens in the second string in order. Declare a match if all the atomic characters in the first string are matched with the first characters of the tokens in the second string. End if there was a match (i.e., skip steps 3 and 4). See lines 13 to 20 in Figure 3.3.
3. Starting with the first word-token in the first string (the first string is assigned as the string with less word-tokens), compare each word-token in the first string to the word-tokens in the second string. Once a match is found (i.e., the threshold is reached), a match score of 1 is assigned to the word-token, the comparison is

stopped and the token position in the second string is marked as the last match position and is un-matchable for subsequent matches, and subsequent word-tokens from the first string are compared first with word-tokens in the second string that are positioned to the right of the matched position, and subsequently the rest of the word-tokens if a match is not found right of the last matched position (lines 21 to 35 in Figure 3.3). Steps 1 to 3 are presented in Figure 3.3. The word-match in step 3 uses a call to Algorithm Match_Words, which is presented in Figure 3.4. The details of the word-match algorithm are presented in step 4 below. The threshold calculation used in Figures 3.3 and 3.4 is explained in section 3.3.

4. At the word token level, the match problem is reduced to a search of the constituent characters of the shorter word token in the longer word token. The characters in the shorter string are searched for in the longer string one after the other. If the length of the shorter word is less than or equal to half the length of the longer string, then the first characters of both words must match for the process to continue (see line 9 in Figure 3.4). Once a match is found, the position is noted. If there is a gap between two matches of the constituent characters, a penalty is charged. The penalty is set at –0.2 for each character space (in the first occurrence of a gap), and is doubled each time a new set of gaps is encountered (lines 17 to 21 in Figure 3.4).

The positional algorithm overcomes the shortcomings of the recursive algorithm as shown in the following examples. Let the two strings being compared be A and B. Given:

A = {"John" "Johnson"}

B = {"Johns" "Smith"}

The positional algorithm will compare "John" in string A with "Johns" in string B. Since there is a match, all subsequent comparisons would not involve "Johns" in string B. Thus, "Johnson" in string A is never compared with "Johns" in string B.

```
1  Algorithm Match_Fields (String1, String2, Adress_of_FieldThreshold) returns
       Field_Match_Score
2  Input:   1. Two strings from a particular field of a data table
3           2. A variable to hold the field threshold computed for the two strings
4  Output:  1. A field match score for the two strings compared
5           2. The field threshold computed
6  Begin
7  Tokenize the two strings being compared into words
8  Count the number of words in each token
9  Set String_Set1 = String with fewer words
10 Set String_Set2 = String with more words
11 Set no_of_Tokens1 = number of words in String_Set1
12 Set no_of_Tokens2 = number of words in String_Set2
13 IF no_of_Tokens1 is equal to 1 AND no_of_Tokens2 is greater than 1 THEN
14   1. Tokenize String-Set1 into characters
15   2. Compare each character in String_Set1 with the first characters of the words in
        String_Set2 in order
16   IF all the characters match THEN
17      1. Set FieldThreshold = 1
18      2. Return Field_Match_Score = 1
19   ENDIF
20 ENDIF
21 Set LastMatchPosition = -1//assuming string index starts with 0//
22 For each word in String_Set1, starting from the first
23      Compare with unmatched words in String_Set2, starting from the right of
    LastMatchPosition, if no match is found till the end of String_Set2, then compare with
    unmatched words from the first position to the LastMatchPosition as follows:
24      Set word1 = The word in String_Set1
25      Set word2 = The next word from String_Set2
26      // call to module that compares two words Figure 3.4 //
27      Word_Match_Score = Match_Words (word1, word2)
28      IF Word_Match_Score equals 1 THEN // means there is a match //
29         //once a match is found, do not compare subsequent words to the matched word //
30         1. LastMatchPosition = The position of the word token in String_Set2
31         2. Add 1 to Total_Match_Score and mark the token position as unmatchable
32         3. Break // stop further comparisons with the word from String_Set1 //
33     ENDIF
34 END FOR
35 Field_Match_Score = Total_Match_Score / no_of_Tokens1
36 Set AverageLength = (no_of_Tokens1 + no_of_Tokens2) / 2
37 IF AverageLength <= 4 THEN
38   Set FieldThreshold = 1 - [ (0.25 * AverageLength) / no_of_Tokens1]
39 ELSE
40   Set FieldThreshold = 1 - [ (0.20 * AverageLength) / no_of_Tokens1]
41 ENDIF
42 Return Field_Match_Score
43 END MODULE
```

Figure 3.3: Algorithm Match_Fields

```
1  Algorithm Match_Words (String1, String2) returns Match
2  Input: Two words to be matched
3  Output: Boolean match decision
4  Begin
5  Tokenize the two string words into characters
6  Count the word lengths, set the shorter word as String1 and the longer word as String2
7  Set LastMatchPosition = -1 //assuming the string index starts at 0//
8  Set Score = 0
9  IF (lengthString1 > lengthString2/2 +1 OR the first characters of the two words are equal)
10 Then
11   For each character in String1, starting from the first
12     Search for that character in String2, right of the LastMatchPosition
13     IF a match is found THEN
14       1. Add 1 to the Score
15       2. Mark that character position as un-matchable for subsequent matches
16       3. Set CurrentMatchPosition to the character position in String2
17       IF there is a gap between the CurrentMatchPosition and the LastMatchPosition
18       THEN
19           Charge a gap penalty for each character position in the gap. [The gap penalty is
             set to -0.2 for each character position in the first set of gaps, and double the last
             gap penalty for all subsequent sets of gaps discovered]
20       ENDIF
21       4. Set the LastMatchPosition = CurrentMatchPosition
22     ELSE IF a match is not found AND LastMatchPosition > 0 THEN
23         1. Search for that character in String2, from the first character position to the
              character position just before LastMatchPosition
24         IF a match is found THEN
25            1 Set LastMatchPosition = current character position in String2
26            2. Mark that character position as un-matchable for subsequent matches
27         ENDIF
28     ENDIF
29   END FOR
30 // the total score is divided by the number of characters in the smaller word //
31 Word_Match_Score = score / no_of_Characters_String1
32 Set AverageLength = (no_of_Characters_String1 + no_of_Characters_String2) / 2
33 IF AverageLength <= 4 THEN
34   Set WordThreshold = 1 - [ (0.25 * AverageLength) / no_of_Characters_String1]
35 ELSE
36   Set WordThreshold = 1 - [ (0.20 * AverageLength) / no_of_Characters_String1]
37 ENDIF
38 IF Word_Match_Score >= WordThreshold THEN
39   Set Match = 1 // True //
40 ELSE
41   Set Match = 0 // False //
42 ENDIF
43 Return Match
44 END MODULE
```

Figure 3.4: Algorithm Match_Words

Also given:

A = "Tims"

B = "Smith"

The positional algorithm will break each of the tokens into its constituent characters as shown below.

A = " 't', 'i', 'm', 's' "

B = " 's', 'm', 'i', 't', 'h' "

The search problem would be to locate the characters of the shorter string in the longer string; thus, the constituent characters in "tims" would be searched for in "smith". The search locates 't' in the 4th position in "smith", a score of 1 is given, 't' is marked as already matched, and the match position recorded. The next character is 'i', and it is searched for in character positions right of 't' in "smith". In this case, there is no 'i' right of 't', so the search starts afresh. The character 'i' is then located at the 3rd position in "smith", a score of 0 is assigned (because of positional disorder), and the match position recorded for further searches (notice that no score is given for a match in lines 24 to 27 of Figure 3.4, to account for positional disorder). The characters 'm' and 's' are also searched for, each yielding a 0 due to positional disorder relative to the most recent match position. Finally, although all the characters in "tims" were located in "smith", only 't' got a score of 1. Thus, total score for the token is 1 divided by the number of characters in the token (in this case, the token score is 0.25). This is an improvement over the recursive algorithm that would have yielded an exact match score of 1 (in error).

The example in the paragraph above showed the penalty scheme for positional disorder. Effectively, in matching the constituent characters of two tokens, a penalty of –1 is given for positional disorder. Penalties are also charged for gaps between matching characters that are in order. The gap penalty is set to –0.2 for each character position in the first set of gaps, and the penalty is set to double the previous penalty (for each character position) for all subsequent sets of gaps (lines 17 to 20 of Figure 3.4). For example, given:

A = "department"

B = "dean"

In the two strings given above, all the characters in "dean" are also contained in "department", thus going by the recursive algorithm with character base, an exact match would be reported. It is also worth noting that the characters in "dean" are in the same positional order in "department" so penalties for positional disorder will not be effective in this case. The positional algorithm uses the gap penalty to handle instances like this. For the example tokens above, the match will be done as follows:

- The search problem is locating the constituent characters of "dean" in "department".
- The first character, 'd', is located, and a score of 1 given. The second character, 'e', also yields a score of 1.
- The third character 'a' is located after one positional gap, thus a score of 1 is given, but with a penalty of –0.2.
- The last character, 'n', is located after four positional gaps, a score of 1 is given, and since this is the second set of gaps, each gap position attracts a penalty that is twice that used for the previous set of gaps. Thus, each positional gap in this second case attracts a penalty of –0.2*2, i.e., -0.4. The four positional gaps would attract a total penalty of -1.6.
- The total score is 1+1+1-0.2+1-1.6 = 2.2, thus, the token match score is 2.2/4.0 = 0.55. This is a great improvement over the exact match score of 1 that would have resulted with the recursive field-matching algorithm with character base.

3.3 Establishing the Match Threshold

For the purpose of determining if two records in a data table are duplicates, there has to be a threshold to which the match score obtained from comparing the two records is checked. Most previous research on data cleaning and duplicate elimination suggest that the threshold match score be experimentally determined by a domain expert [HS95, Mo97, ME97, HS98, LLL+99, Mo00, RH01], some authors however give schemes for determining the threshold score, though these were restricted to specific domains [Hy96]. Researchers working on the AJAX prototype (see section 2.3.2) empirically determined (through experiments with data from various domains), that the best allowable maximum

distance between two word-strings = 20% * Maximum (length(s1), length(s2)), where s1 and s2 are the two word-strings being compared[1]. This thesis adapts the scheme above and extends it into a general scheme for setting the match thresholds at the word-token level, attribute level, and record level. The proposed scheme is domain independent, and can be overridden by a threshold set by a domain expert user. The threshold scheme proposed in this thesis is described in the following paragraphs.

Threshold at the token level

At the token level, the maximum permitted error is set as 25% of the average length of the two tokens being compared if the average length is ≤ 4 characters, and 20% of average length if otherwise. For example, given two tokens

A = "department"

B = "dept"

The average length here is (10 + 4)/2 = 7, thus, maximum allowable error = 0.20 * 7 = 1.4. The threshold is then calculated as 1 – ratio of maximum allowable error to length of shorter token. Thus, threshold = 1 - 1.4/4 = 0.65.

Threshold at the attribute level

The threshold at the attribute level is based on the average of the number of tokens, using the same percentage scheme as in the scheme for token threshold. For example, given the following strings as entries in the address attribute of a data table:

A = " Comput. Sci. Dept., University of Windsor, ON, Canada" and

B = " Department of Computer Science, Univ. Windsor, Ontario"

The two are tokenized and sorted to give:

A = {Canada, Comput, Dept, ON, Sci, University, Windsor}, and

B = {Computer, Department, Ontario, Science, Univ, Windsor}

Each of the tokens in the strings is treated like a character in the token threshold scheme. String A has 7 tokens while string B has 6 tokens. Thus, the maximum allowable error is 0.20 * ((7 + 6)/2) = 1.3. The threshold is then calculated as 1 – ratio of maximum

[1] Based on correspondence with Helena Galhardas, a researcher on the AJAX project. This empirical result is not published.

allowable error to number of tokens in the shorter string. Thus, threshold = 1 - 1.3/6 = 0.78.

Threshold at the record level

The threshold used at the record level is simply the sum of the products of the attribute thresholds and the attribute weights.

3.4 Comparison of the Recursive Algorithm with the Positional Algorithm

The positional algorithm overcomes some of the shortcomings of the recursive algorithm by reducing the possibilities of false positives in duplicate matching (that allows errors in words), thus improving the accuracy of the de-duplication process. The positional algorithm however, does not improve on the time complexity of the recursive algorithm. Both the positional algorithm and the recursive algorithm are non-symmetrical (i.e., match(A,B) is not always the same as match(B,A)). The positional algorithm is non-symmetrical only in cases where the numbers of word tokens in the fields being compared are the same (for field matching), or the numbers of characters in the words being compared are equal (for word matching). For all other cases, the positional algorithm is symmetrical because the algorithm always searches for the shorter string in the longer one. The two algorithms have quadratic time complexities $O(n^2)$ at both the word-match level and the field-match level, where n is the number of tokens (characters or words respectively) in the longer string. This is because at the worst case, each character (or word) in one string must be compared with all the characters (or words) in the second string. The comparisons are the major costs in both algorithms as the tokenization of a string has an $O(n)$ time complexity, where n is the number of tokens in the string.

At the record-match level, the recursive algorithm also has a quadratic time complexity $O(n^2)$, where n is the number of fields in each record. This is because each field in one record is matched with every field in the second record. The positional algorithm however compares only corresponding entries in the selected fields (with assigned weights); thus, the number of fields in the records being compared does not affect its time

complexity. The result is an $O(k)$ time complexity at the record level, where k is the number of fields with assigned weights. The use of field weights was introduced in [LLL+99], thus the credit for the reduction in time complexity at the record level goes to [LLL+99]. The authors in [LLL+99] leave the assignment of the field weights to a domain expert user. This thesis uses a data profiling technique to assign the field weights irrespective of the data domain. The data profiling technique employed is discussed in section 3.5.

3.5 Field Weighting with Data Profiling

Data profiling is simply the characterization of data through the collection and analysis of data content statistics. Most duplicate-elimination schemes provide a way of discriminating between the fields (attributes) of the records in the database. Some fields are assigned as being more important than others in determining if two records should be declared as duplicates. Data cleaning algorithms developed for specific domains hard-code the field importance into the duplicate elimination algorithms [HS95, Hy96, HS98]. While some existing domain-independent techniques leave the assignment of the field weights or importance to the user [LLL+99, RH01], others give equal weights to all the fields of the records being compared [Mo97]. The authors of the domain-independent algorithms propose that a domain expert be used to assign the field weights, or that the weights be assigned through several experimentations and observation of results.

This thesis uses a systematic way to assign the field weights correctly. The technique may still be used by a domain expert to guide her judgment, or could be used in cases where domain knowledge is scarce or limited. The scheme used in this thesis is based on the discriminating power of the attributes of the records in a database to uniquely identify the individual records of the database. The technique gathers data content statistics from a subset of the data table to be cleaned, and uses the information to assign weights to the attributes of the data table. Thus, the technique adapts well to changing data domains. The scheme is outlined in the following steps:

1. Given a database table of N records, where N is a large number, select a subset of the table (n records) and collect the following statistics from the data contained in

the fields of the n records: uniqueness, presence of null values, and field length. The major characteristic of interest here is uniqueness. Uniqueness measures the percentage of the data contained in each of the attributes of the n records that are without repetition. In determining the uniqueness factor, each field in the n subsets of the database table is grouped into clusters. Two entries are merged into a cluster if they are exact matches or one is a prefix of the other. Also, fields with lengths greater than 100 are grouped into one cluster. The ratio of the number of resulting clusters to the number of records in the subset is the percentage uniqueness (note that one resulting cluster is equivalent to zero uniqueness).

2. After processing the n-records subset of the data table, the scheme shown in Table 3.1 is used to determine the score of each attribute.
3. Given a table with K attributes, if all the K attributes performed perfectly well in step 2 above, then the total score of all the attributes would be 100*K. The fields are then ranked in descending order of their scores. The first l fields that achieve a total score of (100*K)/2 (minimum of 2 fields if $K \geq 2$), up to a maximum of 5 fields, are given weights relative to their scores and used in the field-matching algorithm. If all the fields do not achieve a total of at least (100*K)/2, then the top (K/2) + 1 attributes from the ranked list of attributes, up to a maximum of 5 fields, will be assigned weights relative to their scores and used in the match.
4. If l fields are selected from step 3 above, each with a score of t, the total score for the selected fields, T is then the sum of the t_l's. The weight w for each field is then computed as: $w = t/T$

Figure 3.5 presents the four steps discussed above algorithmically. The major activity in the data-profiling algorithm is the clustering of the entries in each field to determine uniqueness. To determine clusters, each entry in a particular field of the database would be compared to all the previous clusters in that field. In the worst case, all the entries would be unique clusters, resulting in n/2(n-1) comparisons, where n is the number of records in each subset of the data-table selected for profiling. This results in an $O(n^2)$ time complexity for scoring each field in the subset of the database. Given that the data table has k attributes, then since each attribute needs to be scored, the time complexity for scoring each subset of the database would be $O(kn^2)$. Once all the attributes are scored,

Characteristic	**Percentage Achieved**	**Score**
Uniqueness	95% - 100%	100
	90% - 94%	90
	80% - 89%	80
	70% - 79%	70
	60% - 69%	60
	50% - 59%	50
	40% - 49%	40
	30% - 39%	30
	20% - 29%	20
	10% - 19%	10
	0% - 9%	0
Null Values	< 5%	- 0
	5% - 10%	- 5
	11% - 20%	- 10
	21% - 30%	- 15
	31% - 40%	- 20
	41% - 50%	- 25
	> 50%	- 30
Field Length (> 50 Characters)	< 5%	- 0
	5% - 10%	- 5
	11% - 20%	- 10
	21% - 30%	- 15
	31% - 40%	- 20
	41% - 50%	- 25
	> 50%	- 30

Table 3.1: Scheme for Scoring Field Weights

```
Algorithm Compute_Field_Weights (data table, start_record, Address_of_selected_fields)
Input: Data table, The record to start computation from: K, the number of records to traverse:
      n, Address to store the selected fields and their weights
Output: Selected fields with their assigned weights
Begin
Input n // a number much smaller than the number of records in the data table. Suggested
        value is 100. //
For each field in the data table
      Starting from the Kth record, and going through n records
      DO
         1. Group the data in the field into clusters. Data that are exact matches or with one
            being the prefix of the other are placed in the same cluster
         2. Count all the null value entries in the field
         3. Count the number of data entries in the field with character lengths > 50
            characters
      END DO
      Set Uniqueness_Ratio = (no_of_Clusters / n) * 100
      Set Null_Ratio = (Null_Count / n) * 100
      Set Off_Length_Ratio = (Off_Length_Count / n) * 100
      Uniqueness_Score = getUniquenessScore (Uniqueness_Ratio) // using scheme shown
      in Table 3.1 //
      Null_Penalty = getNullPenalty (Null_Ratio) // using scheme shown in Table 3.1 //
      Off_Length_Penalty =getOffLengthPenalty (Off_Length_Ratio)//using scheme shown
      in Table 3.1 //
      Field_Score = Uniqueness_Score + Null_Penalty + Off_Length_Penalty
END FOR
Sort the field scores in descending order
IF sum of the field scores >= 100 * no_of_Columns / 2 THEN
      1. Select the first l fields whose scores add up to 100 * no_of_Columns / 2, l should be
         a minimum of 2 fields if no_of_Columns >= 2, and a maximum of 5 fields
      2. Sum the scores of the l selected fields
      3. For each of the l selected fields
            1. Set Field_Weight = Field_Score / sum_of_Scores
            2. Add the Field_Name and Field_Weight to the set of selected fields
         END FOR
ELSE
      1. Select the first [ (no_of_Columns / 2 ) + 1] fields to a maximum of 5 fields
      2. Sum the scores of the selected fields
      3. For each of the selected fields
            1. Set Field_Weight = Field_Score / sum_of_Scores
            2. Add the Field_Name and Field_Weight to the set of selected fields
         END FOR
END IF
Return
END MODULE
```

Figure 3.5: Algorithm Compute_Field_Weights

the other major activity would be the ranking/sorting of the field scores, which has an $O(k \log k)$ time complexity. The dominant cost is thus $O(kn^2)$, and this is the time complexity of profiling each selected subset of the data table. If the algorithm is set up such that the profiling is done m times, then the time complexity for the entire data profiling process would be $O(kmn^2)$. The time complexity shows that n should be kept reasonably small for best performance. The performance of the algorithm would also degrade as k and m become excessively large.

The four steps outlined above must be performed in the first instance before the duplicate elimination process begins. The field with the highest weight could then be chosen as the first sort key for the data cleaning process. The user can then set an interval after which the data may be profiled again and the scores averaged with the existing scores. The algorithm adapts to the new scores and assigns weights accordingly. For example, given a data table with 100,000 records, we could choose to profile the first 100 records in the first instance. The scores obtained from this first profiling are then used to assign weights to the fields of the database, and to select the initial sort key. We can also choose to profile the data after every 20,000 records; thus, in the first pass of the multi-pass merge/purge technique we could profile the data 5 times. Each of the profiling operations updates the scores of the data fields and consequently, the weights and selection of fields that should participate in the matching decision. The highest scoring attribute of the table at the end of each pass (of the multi-pass merge/purge technique) that has not been used previously as the sort-key is selected as the sort key for the next pass.

For example, given two data tables with the same schema but with data from different data domains as shown in Table 3.2, the weights assigned to the different attributes of the database reflect the characteristics of the data in the respective domains. Applying the proposed field-weighting algorithm to Table 3.2a, the resulting scores for each of the attributes is shown below.

ID: The record_id (primary key) is not considered.

Name: 100 for uniqueness, -0 for null values, -0 for field length, giving a total of 100.

Sex: 40 for uniqueness, -0 for null values, -0 for field length, giving a total of 40.

ID	Name	Sex	Street	City	State
001	Ingrid Green	F	605 Thompson Dr.	N. York	ON
002	George Georges	M	1450 Concession Ave.	Hamilton	ON
003	Miriam Acer	F	443 Finch Street	Toronto	Ontario
004	Mary Smith	F	720 Rankin Ave	Sudbury	ON
005	Jerry Adams	M	1777 Tecumseh Road	Windsor	ON

Table 3.2a: A sample table of customer addresses in Ontario

ID	Name	Sex	Street	City	State
001	Jennifer Forrester	F	695 Felix Street	Windsor	ON
002	Jason Kidd	M	470 College Ave.	Windsor	ON
003	Anna Plant	F	290 University Avenue	Windsor	Ontario
004	Allison Fraser	F	722 Rankin Ave	Windsor	ON
005	Jane Neale	M	570 Indian Road	Windsor	ON

Table 3.2b: A sample table of customer addresses in Windsor, Ontario

Street: 100 for uniqueness, -0 for null values, -0 for field length, giving a total of 100.

City: 100 for uniqueness, -0 for null values, -0 for field length, giving a total of 100.

State: 0 for uniqueness, -0 for null values, -0 for field length, giving a total of 0.

Going by the algorithm, the number of attributes K is 5 (ID is not counted), thus (100*K)/2 = 250. If the attributes are ranked in descending order of their scores, the first fields that their cumulative scores add up to (100*K)/2 are Name, Street, and City. These three fields are then selected, and each is given a weight that is a ratio of their score to the total score of the selected fields. For this example, each of the fields will be assigned a weight of 1/3 (because they all had equal scores). These three fields would be the only ones considered in the field-matching algorithm until an update in the field-weighting algorithm suggests otherwise.

Repeating the process for the data in Table 3.2b, all the attributes, except City, retain their scores. The City field in this case is not unique and receives a uniqueness score of 0. The total score for City in this case is also 0. If the attributes are ranked in descending order of scores, the result is shown below:

Name: 100

Street: 100

Sex: 40

City: 0

State: 0

ID: Not considered.

Going by the algorithm, since the attributes could not achieve a total score of (100*K)/2, the first (K/2)+1 attributes in the ranked list would be used for the field matching. For this example, Name, Street, Sex, and City with a total score of 240 would be assigned weights. Thus, Name, Street, Sex, and City would have weights of (100/240), (100/240), (40/240), and (0/280) respectively. This example clearly shows how the algorithm adapts to data-content changes in assigning field weights, even where the database schema is the same.

4. EXPERIMENTATION AND EVALUATION

This chapter presents the results of experimental evaluation of the positional algorithm, recursive algorithm with word-base, and recursive algorithm with character-base. All the experiments were performed on a 1GHz Intel Celeron PC with 256 megabytes main memory, running Windows XP Home Edition. All the programs are written in Java. The data sets were stored as Oracle data tables (on Oracle 8i Personal Edition), and were retrieved using JDBC.

Two data sets were used to test the algorithms for accuracy. The data sets were set up from real-world subscription lists and the duplicates were introduced to especially increase the probability of false positives in the match results. The set-up of the experiments and the results achieved for accuracy and response time are discussed in sections 4.1 and 4.2 respectively.

4.1 Experiments for Accuracy

Two data sets were used to test the algorithms for accuracy. The data sets were set up such that the probability of false positives would be high. The experiments use two measures of accuracy to evaluate the effectiveness of the algorithms in de-duplicating the data sets. The first measure of accuracy used is recall. Recall is the ratio of true matches returned by a matching algorithm to the number of duplicates in the original database. The second measure of accuracy used is precision. Precision measures the ratio of true matches returned by a matching algorithm to the total number of matches returned by the algorithm. Thus, precision gives a measure of the level of false positives in the returned results. The three algorithms were run using threshold values ranging from 0.1 to 1.0. The positional algorithm was additionally run using the threshold scheme discussed in Chapter three. The field weights used for the positional algorithm were assigned using the data-profiling scheme discussed in Chapter three. The data tables were set up such that the first column is a unique identifier. This is useful in evaluating the results of the various runs. The results of the experimental runs on the two data sets are discussed in the paragraphs below.

Test Data 1

This data set was developed from a real-world subscription list with six fields (name, address, city, province, postal code, and telephone). Six additional fields were added to the data, these are record id, membership year, level of education, discipline, occupation, and gender. The record id field was not used in any of the matching algorithms. Its function was in the evaluation of match results. The data set was set up to contain many records with similar field entries, but which were not duplicates. A hundred and nineteen records were used in the evaluation, with fourteen pairs of duplicated records (i.e., twenty-eight duplicate records). The experiment was set up such that each record is compared with all subsequent records in the table (resulting in a progression of the form N/2(N-1) where N is the number of records). Thus this data set resulted in 7021 pairs of records, with 14 duplicate pairs. The duplicates were set up with acronyms, character substitutions, deletions, insertions, and transpositions, with multiple errors in each duplicated record. There were no exact duplicates in this data set, and each duplicated record was allowed only one duplicate. Table 4.1 presents the results achieved when the three algorithms were run on test data 1. Figures 4.1 and 4.2 depict the percentage precision and percentage recall achieved for varying thresholds respectively.

Results for test data 1 (Table 4.1) show that the positional algorithm achieves a higher precision than the other two algorithms for all levels of recall. The results also show that the threshold function discussed in chapter three does not approximate the best threshold. The auto-threshold achieved a recall of 28.57% and a precision of 57.14%. This obviously does not approximate the best result as 50% precision was achieved at a recall of 100% using a threshold of 0.6. The highest recall achieved by the recursive algorithm with word base for test data 1 is 87.51%, and its best precision for this threshold is 0.2%. The recursive algorithm with character base achieved a 100% recall, but its best precision at that recall level is 0.22%. Test data 1 was clearly challenging for all three algorithms; however, the positional algorithm outperforms the other two algorithms with respect to accuracy on this data set.

Threshold	Positional Algorithm		Recursive Algorithm with word-base		Recursive Algorithm with character-base	
	%Recall	%Precision	%Recall	%Precision	%Recall	%Precision
0.1	100	0.20	85.71	0.20	100	0.20
0.2	100	0.20	50	0.14	100	0.20
0.3	100	0.26	50	0.25	100	0.20
0.4	100	2.89	42.86	0.47	100	0.20
0.5	100	15.73	28.57	0.87	100	0.20
0.6	100	50	14.29	2.15	100	0.22
0.7	42.86	37.5	14.29	9.52	92.86	0.29
0.8	14.29	50	14.29	33.33	57.14	0.86
0.9	0	N/A	0	N/A	21.43	25
1.0	0	N/A	0	N/A	0	N/A
Auto Threshold	28.57	57.14				

Table 4.1: Experimental Results on Test data 1

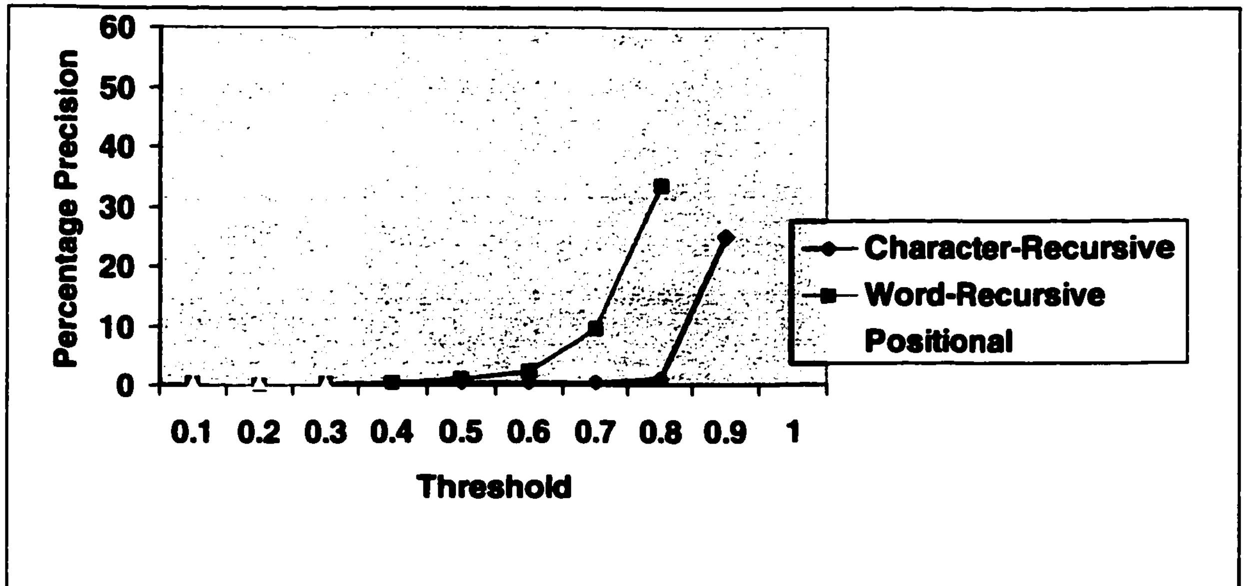

Figure 4.1: Percentage Precision versus Threshold on Test data 1

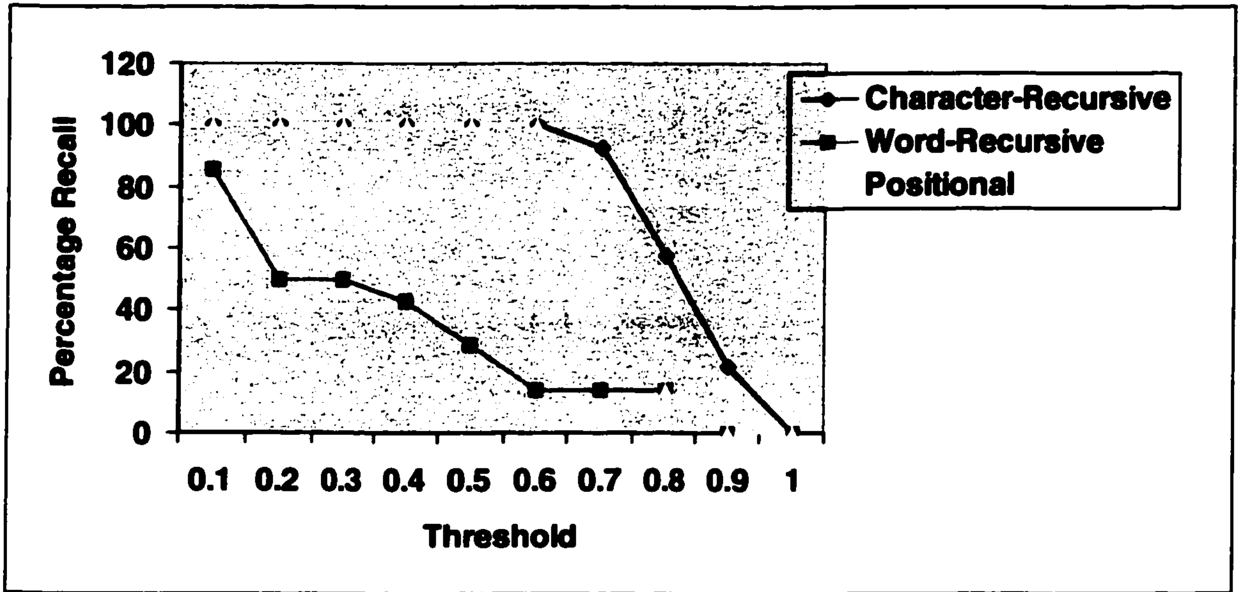

Figure 4.2: Percentage Recall versus Threshold on Test data 1

__Test Data 2__

This data set was developed from a real-world list of doctors in Ontario, Canada. It consists of two fields (record id and name). The record id field was not used in any of the matching algorithms. Its function was in the evaluation of match results. This data set was set up especially to evaluate the field-matching effectiveness of the various algorithms irrespective of other constraints such as field weights. The data set is made up of one hundred and thirty five records, with fourteen duplicate pairs. The experiment was set up (as with test data 1) such that each record is compared with all subsequent records in the table (resulting in a progression of the form N/2(N-1) where N is the number of records). Thus this data set resulted in 9045 pairs of records, with 14 duplicate pairs. The duplicates in this case were developed with word transpositions, deletions, substitutions, insertions, as well as exact duplicates, thus it is expected that a hundred percent precision can be achieved at a recall that is higher than zero. The results from running the three algorithms on this data set are presented in Table 4.2. Figures 4.3 and 4.4 depict the percentage precision and percentage recall achieved for varying thresholds respectively.

The three algorithms performed better with test data 2 than with test data 1. The positional algorithm however, still outperforms the other two algorithms on accuracy, achieving a higher precision for every level of recall. As with test data 1, the auto threshold function failed to approximate the performance of the best threshold tested. The auto threshold achieved a recall of 71.43% with 50% precision. This does not measure up to 87.5% precision achieved at a 100% recall. As with test data 1, the recursive algorithm with word base failed to attain a 100% recall, achieving a maximum recall of 71.43%. The best precision achieved at this level of recall for the recursive algorithm with word base is 6.21%. The recursive algorithm with character base achieved 100% recall levels, but with much lower precision than the other two algorithms. The best precision achieved by the recursive algorithm with character base at 100% recall is 0.47%.

Threshold	**Positional Algorithm**		**Recursive Algorithm with word-base**		**Recursive Algorithm with character-base**	
	%Recall	%Precision	%Recall	%Precision	%Recall	%Precision
0.1	100	0.17	71.43	6.17	100	0.16
0.2	100	0.19	71.43	6.17	100	0.17
0.3	100	0.31	71.43	6.21	100	0.20
0.4	100	0.53	64.29	8.26	100	0.28
0.5	100	1.06	64.29	8.26	100	0.47
0.6	100	77.78	14.29	28.57	85.71	0.92
0.7	100	87.5	7.14	16.67	71.43	2.11
0.8	71.43	100	7.14	16.67	71.43	8.40
0.9	35.71	100	7.14	16.67	42.86	35.29
1.0	21.43	100	7.14	16.67	14.29	33.33
Auto Threshold	71.43	50				

Table 4.2: Experimental Results on Test data 2

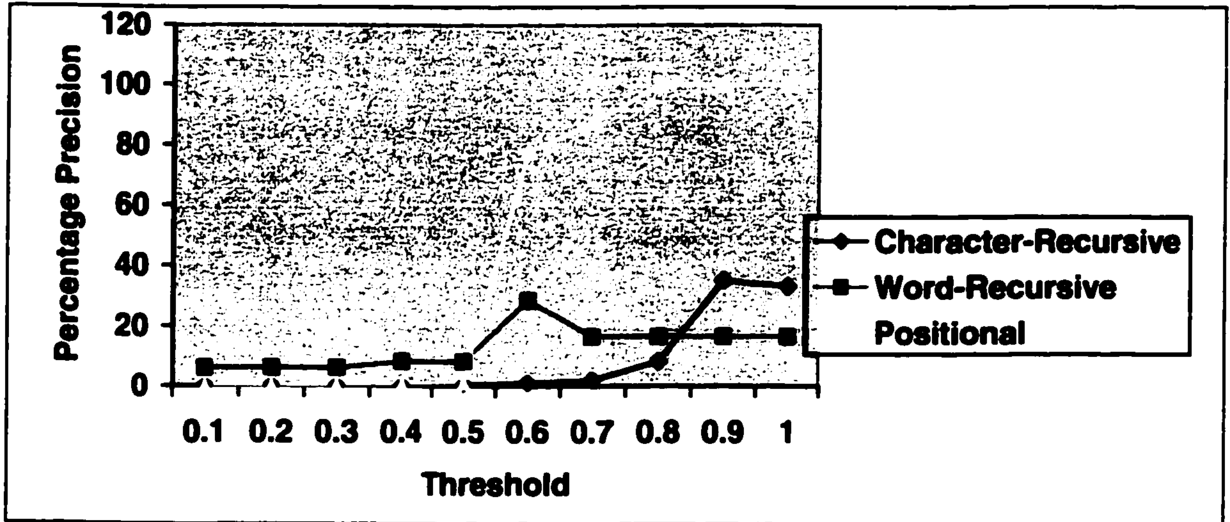

Figure 4.3: Percentage Precision versus Threshold on Test data 2

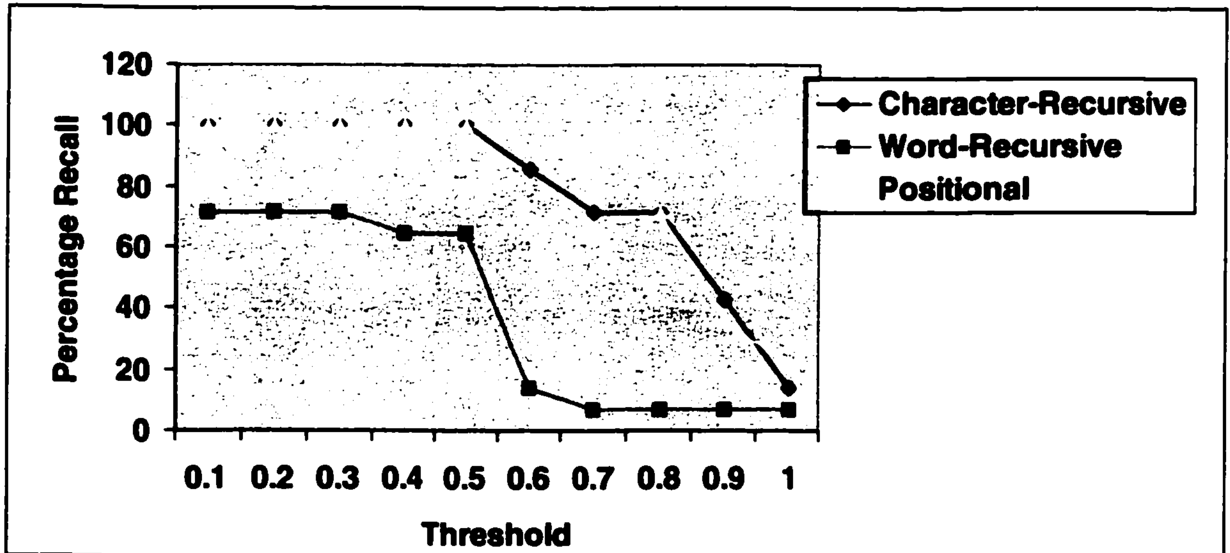

Figure 4.4: Percentage Recall versus Threshold on Test data 2

4.2 Experiments for Response Time

The algorithms were tested for speed using data sets that consist of a single field (names). The aim of these experiments is to show that at the field match level, the positional algorithm has the same time complexity as both the recursive algorithm with character base and the recursive algorithm with word base. All the three algorithms were established as having quadratic time complexities at the field level in section 3.4. The data sets were made to be single field data so as to remove the quadratic record-matching

disadvantage of the recursive algorithm (as discussed in section 3.4), and show the performance of the algorithms only at a field level. The algorithms were run on data sets with 100, 200, 400, 800, 1600, 3200, 6400, and 12800 records, with threshold set to zero. The results are shown in Table 4.3. The results show that all three algorithms performed similarly at the various record levels, and they all exhibited a quadratic time complexity. The results are presented diagrammatically in Figure 4.5.

Number of Records	Time (seconds)		
	Positional Algorithm	Recursive Algorithm with word-base	Recursive Algorithm with character-base
100	1	1	1
200	3	3	3
400	7	7	7
800	23	23	21
1600	84	87	76
3200	330	340	327
6400	1243	1271	1236
12800	5085	5246	5052

Figure 4.3: Experimental Results on Response Time

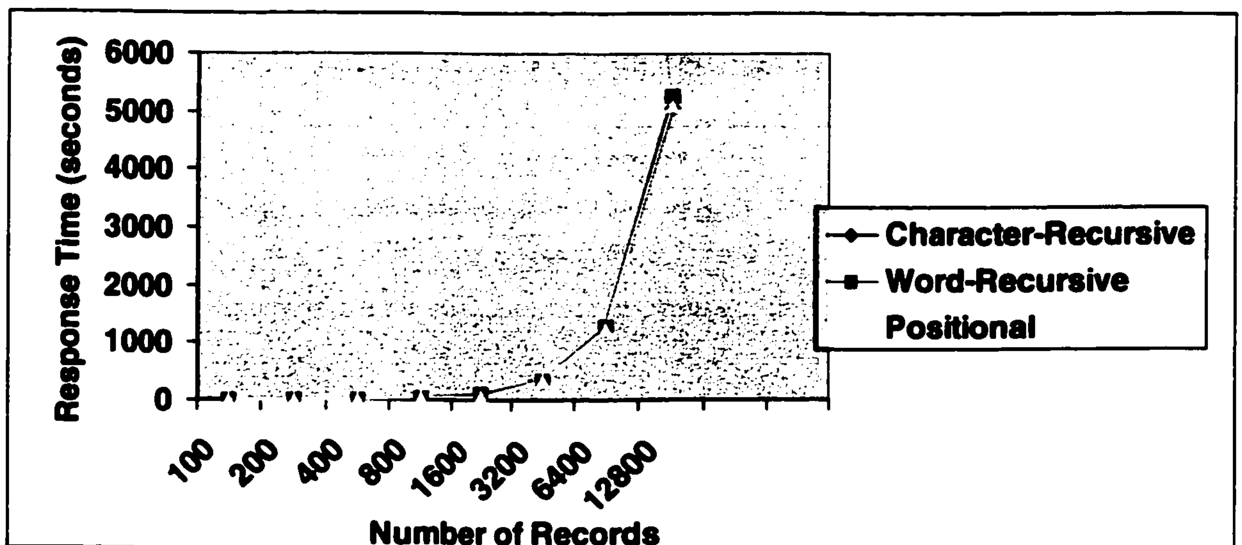

Figure 4.5: Response Time versus Number of Records

5. CONCLUSIONS AND FUTURE WORK

Data cleaning will continue to be an important and demanding phase in data warehouse development. The removal of duplicates is a major challenge in cleaning data warehouses. Domain-independent approaches to data cleaning (and de-duplication in particular) present an interesting and worthwhile field of research. The domain-independent approaches reduce the repetitive efforts required in data warehouse development, and from a Software Engineering perspective, offer reuse and standardization of data cleaning tools.

This thesis contributes an enhancement to the recursive field-matching algorithm for domain-independent field matching. The enhancement is termed the positional algorithm. The thesis also contributes a data profiling technique for assigning field importance for the purpose of de-duplication. The enhanced approach is shown to overcome the major shortcomings of the recursive field-matching algorithm with respect to accuracy. The approach however does not handle cases of field mismatches (for example entering the first name in the last name field and vice versa). Furthermore, the concept of domain independence used here applies only to data from the Unicode character set. An area of future research is to extend the domains covered to include image data, biological data, etc.

Experiments show that the positional algorithm performs better than the recursive algorithm with respect to accuracy, while running at similar response times. Experiments also confirm that the data profiling technique used in this thesis effectively assigns field weights to the fields of the database. The experiments however show that the threshold scheme suggested in this thesis does not approximate the best threshold. Areas of future research include the further investigation of a domain-independent threshold function for de-duplication, and the enhancement of the data profiling technique with machine learning principles. We also intend to evaluate the positional algorithm (at the word-match level) against existing classical text-searching techniques.

REFERENCES

[ACM97] S. Abiteboul, S. Cluet, T. Milo, Correspondence and Translation for Heterogeneous Data, In Proc. of the International Conf. on Database Theory (ICDT), pages 351 – 363, January 1997.

[ACM+99] S. Abiteboul, S. Cluet, T. Milo, P. Mogilevsky, J. Simeon, S. Zohar, Tools for Data Translation and Integration, IEEE Data Engineering Bulletin, 22(1), pages 3 – 8, 1999.

[AE95] S. Adali, R. Emery, A Uniform Framework for Integrating Knowledge in Heterogeneous Knowledge Systems, In Proc. Of the 11th Intl. Conference on Data Engineering, pages 513 – 520, 1995.

[BN99] R. Baeza-Yates, G. Navarro, Faster Approximate String Matching, Algorithmica 23, 2, pages 127 – 158, 1999.

[BLN86] C. Batini, M. Lenzerini, S.B. Navathe, A Comparative Analysis of Methodologies for Database Schema Integration, ACM Computing Surveys, 18(4): 323 - 364, December 1986.

[BBD+00] K. Bharat, A. Broder, J. Dean, M.R. Henzinger, A Comparison of Techniques to Find Mirrored Hosts on the WWW, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, pages 21 – 26, 2000.

[BD83] D. Bitton, D.J. DeWitt, Duplicate Record Elimination in Large Data Files, ACM Transactions on Database Systems, 8(2): 255 – 65, 1983.

[BMY98] M. Bobrowski, M. Marre, D. Yankelevich, A Software Engineering View of Data Quality, In Proc. of the 2nd Annual International Software Quality Week, Europe (QWE'98), 1998.

[CDR99] D. Calvanese, G. De Giacomo, R. Rosati, Data Integration and Reconciliation in Data Warehousing: Conceptual Modeling and Reasoning Support, Network and Information Systems, 4(2), pages 412 – 432, 1999.

[CS91] A. Chatterjee, A. Segev, Data Manipulation in Heterogeneous Databases, ACM SIGMOD Record, Vol. 20, No. 4, pages 64 – 68, 1991.

[CD97] S. Chaudhuri, U. Dayal, An Overview of Data Warehousing and OLAP Technology, ACM SIGMOD Record, Vol. 26, No. 1, pages 65 – 74, 1997.

[CR99] K. Claypool, E. Rundensteiner, Flexible Database Transformations: The SERF Approach, IEEE Data Engineering Bulletin, 22(1), pages 19 – 24, March 1999.

[CDS+98] S. Cluet, C. Delobel, J. Simeon, K. Smaga, Your Mediators need Data Conversion, In Proc. of ACM SIGMOD Conference on Data Management, pages 177 – 188, 1998.

[Co98a] W. Cohen, Integration of Heterogeneous Databases without Common Domains using Queries based on Textual Similarity, In Proc. of ACM SIGMOD Conference on Data Management, pages 201 – 212, 1998.

[Co98b] W. Cohen, The WHIRL Approach to Integration: An Overview. In the AAAI – 98 Workshop on AI and Information Integration, pages 1 – 8, 1998.

[DDL00] A.H. Doan, P. Domingos, A.Y. Levy, Learning Source Description for Data Integration, in: Proceeding of the 3rd International Workshop on The Web and Databases (WebDB), pages 81 – 86, 2000.

[ET98] R. Engels, C. Theusinger, Using a Data Metric for Preprocessing Advice for Data Mining Applications, Proc. of the 13th European Conf. on Artificial Intelligence (ECAI 98), pages 430 – 434, 1998.

[Ez01] C.I. Ezeife, Selecting and Materializing Horizontally Partitioned Warehouse Views, In Data and Knowledge Engineering, Vol. 36, pages 185 – 201, 2001.

[GFS+00a] H. Galhardas, D. Florescu, D. Shasha, E. Simon, Declaratively Cleaning your Data using AJAX, In Journees Bases de Donnees, Oct. 2000.

[GFS+00b] H. Galhardas, D. Florescu, D. Shasha, E. Simon, AJAX: An Extensible Data Cleaning Tool, Proc. ACM SIGMOD Conference, page 590, 2000.

[GFS+00c] H. Galhardas, D. Florescu, D. Shasha, E. Simon, An Extensible Framework for Data Cleaning, In Proceedings of the International Conference on Data Engineering (ICDE), San Diego, CA, pages 312 – 344, 2000.

[GFS+01a] H. Galhardas, D. Florescu, D. Shasha, E. Simon, C. Saita, Improving Data Cleaning Quality using a Data Lineage Facility, In Proc. of the Inter. Workshop on Design and Management of Data Warehouses (DMDW'2001), pages 3 – 16, 2001.

[GFS+01b] H. Galhardas, D. Florescu, D. Shasha, E. Simon, C. Saita, Declarative Data Cleaning: Language, Model, and Algorithms, Proceedings of the 27th VLDB Conference, Roma, Italy, pages 371 – 380, 2001.

[GHI+95] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom, Integrating and Accessing Heterogeneous Information Sources in TSIMMIS, In Proc. of the AAAI Spring Symposium on Information Gathering, pages 61 – 64, 1995.

[GJJ+98] M. Gebhardt, M. Jarke, M.A. Jeusfeld, C. Quix, S. Sklorz, Tools for Data Warehouse Quality, In IEEE Proc. of the 10th International Conference on Scientific and Statistical Database Management, pages 229 – 232, July 1998.

[HAC+99] J.M. Hellerstein, R. Avnur, A. Chou, C. Olston, V. Raman, T. Roth, C. Hidber, P. Haas, Interactive Data Analysis: The Control Project, IEEE Computer, 32(8), pages 51 – 58, August 1999.

[He96] M. Hernandez, A Generalization of Band Joins and the Merge/Purge Problem, Ph.D. Thesis, Columbia University, 1996.

[HS95] M. Hernandez, S. Stolfo, The Merge/Purge Problem for Large Databases, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 127 – 138, May 1995.

[HS98] M.A. Hernandez, S.J. Stolfo, Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem, Data Mining and Knowledge Discovery 2(1): 9-37, 1998.

[Hy96] J. Hylton, Identifying and Merging Related Bibliographic Records, Master's Thesis, Massachusetts Institute of Technology, June 1996.

[LLL00] M.L. Lee, T.W. Ling, W.L. Low, IntelliClean: A Knowledge-Based Intelligent Data Cleaner, In Proc. ACM SIGKDD'2000, pages 290 – 294, Boston, 2000.

[LLL+99] M.L. Lee, H. Lu, T.W. Ling, 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.

[Li95] W. Li, Knowledge Gathering and Matching in Heterogeneous Databases, Working Notes of the AAAI Spring Symposium on Information Gathering from Heterogeneous, Distributed Environments, pages 116 – 121, 1995.

[MM00] J.I. Maletic, A. Marcus, Data Cleansing: Beyond Integrity Analysis, in: Proceedings of The Conference on Information Quality (IQ 2000), pages 200 – 209, October 2000.

[Mi98] R.J. Miller, Using Schematically Heterogeneous Structures, Proceedings of the ACM SIGMOD, 27(2): 189 – 200, Seattle, WA, June 1998.

[MZ98] T. Milo, S. Zohar, Using Schema Matching to Simplify Heterogeneous Data Translation, In Proc. of the International Conference on Very Large Databases (VLDB), pages 122 – 133, New York, 1998.

[Mo97] A. Monge, Adaptive Detection of Approximately Duplicate Database Records and The Database Integration Approach to Information Discovery, Ph.D. Thesis, Dept. of Comp. Sci. and Eng., Univ. of California, San Diego, 1997.

[Mo00] A.E. Monge, Matching Algorithms within a Duplicate Detection System, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, pages 14 – 20, 2000.

[ME95] A.E. Monge, C.P. Elkan, WebFind: Automatic Retrieval of Scientific Papers over the World Wide Web, In Working Notes of the Fall Symposium on AI Applications in Knowledge Navigation and Retrieval, page 151, AAAI Press, 1995.

[ME96a] A.E. Monge, 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.

[ME96b] A.E. Monge, C.P. Elkan, Integrating External Information Sources to guide Worldwide Web Information Retrieval, Technical Report CS96 –

474, Dept. of Computer Science and Engineering, Univ. of California, San Diego, January 1996.

[ME96c] A.E. Monge, C.P. Elkan, The WEBFIND Tool for Finding Scientific Papers over the Worldwide Web, In Proc. of the Third Inter. Congress on Computer Science Research, pages 41 – 46, Tijuana, Mexico, 1996.

[ME97] A.E. Monge, C.P. Elkan, An Efficient Domain-Independent Algorithm for Detecting Approximately Duplicate Database Records, Proceedings of the SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, pages 23 – 29, Tucson, Arizona, May 1997.

[Na01] G. Navarro, A Guided Tour to String Matching, ACM Computing Surveys, 33(1), pages 31 – 88, March 2001.

[NKA+59] H.B. Newcombe, J.M. Kennedy, S.J. Axford, A.P. James, Automatic Linkage of Vital Records, Science, Vol. 130: 954 – 959, October 1959.

[PGU96] Y. Papakonstantinou, H. Garcia-Molina, J. Ullman, MedMaker: A Mediation System Based on Declarative Specification, In Proc. of the International Conference on Data Engineering (ICDE), pages 132 – 141, 1996.

[PS98] C. Parent, S. Spaccapietra, Issues and Approaches of Database Integration, Communications of the ACM, 41(5), pages 166 – 178, 1998.

[RD00] E. Rahm, H.H. Do, Data Cleaning: Problems and Current Approaches, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, pages 3 – 13, 2000.

[RCH99] V. Raman, A. Chou, J.M. Hellerstein, Scalable Spreadsheets for Interactive Data Analysis, ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery (DMKD), May 1999.

[RH01] V. Raman, J.M. Hellerstein, Potters Wheel: An Interactive Framework for Data Cleaning and Transformation, Proc. of the International Conference on Very Large Databases (VLDB), pages 381 – 390, Roma, Sept. 2001.

[SMK+98] S. Samtani, M. Mohania, V. Kumar, Y. Kambayashi, Recent Advances and Research Problems in Data Warehousing, Proc. of ER Workshops, pages 81 – 92, 1998.

[SHM+99] C. Sapia, G. Hofling, M. Muller, C. Hausdorf, H. Stoyan, U. Grimmer, On Supporting the Data Warehouse Design by Data Mining Techniques, in: Proceedings of GI-Workshop on Data Mining and Data Warehousing, 1999.

[SW81] T.F. Smith, M.S. Waterman, Identification of Common Molecular Subsequences, Journal of Molecular Biology, 147:195 – 197, 1981.

[Ta94] T. Takaoka, Approximate Pattern Matching with Samples, In Proc. ISAAC'94, LNCS 834, pages 234 – 242, Springer-Verlag, 1994.

[Uk85] E. Ukkonen, Finding Approximate Patterns in Strings, Journal of Algorithms, 6: 132 – 137, 1985.

[VVS+00] P. Vassiliadis, Z. Vagena, S. Skiadopoulos, N. Karayannidis, T. Sellis, ARKTOS: A Tool for Data Cleaning and Transformation in Data Warehouse Environments, Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, pages 42 – 47, 2000.

[Wi95a] J. Widom, Research Problems in Data Warehousing, in: Proceedings of the 4th International Conference on Information and Knowledge Management (CIKM), pages 25 – 30, November 1995.

[Wi92] G. Wiederhold, Mediators in the Architecture of Future Information Systems, Computer 25(3): 38-49, 1992.

[Wi94] W.E. Winkler, Advanced Methods for Record Linkage, Technical Report, Statistical Research Division, Washington DC: US Bureau of the Census, 1994.

[Wi95b] W.E. Winkler, Matching and Record Linkage, In Brenda G. Cox, editor, Business Survey Methods, pages 355 – 384, Wiley, 1995.

[ZHK+95] G. Zhou, R. Hull, R. King, J. Franchitti, Supporting Data Integration and Warehousing Using H2O, IEEE Data Engineering Bulletin, 18(2), pages 29 – 40, 1995.

[ZHK96] G. Zhou, R. Hull, R. King, Generating Data Integration Mediators that use Materialization, Journal of Intelligent Information Systems, 6(2/3): 199-221, 1996.

VITA AUCTORIS

NAME	Ajumobi Okwuchukwu Udechukwu
YEAR OF BIRTH	1976
PLACE OF BIRTH	Nnewi, Nigeria
EDUCATION	M.Sc., Computer Science University of Windsor Windsor, Ontario Canada 2001 – 2002
	B.Sc., (First Class Honors) Computer Science with Economics Obafemi Awolowo University Ile-Ife Nigeria 1992 - 1998