

Copyright 2010 Randy Siran Gu

DATA CLEANING FRAMEWORK:
AN EXTENSIBLE APPROACH TO DATA CLEANING

BY

RANDY SIRAN GU

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Associate Professor Kevin Chang

Abstract

The growing dependence of society on enormous quantities of information stored electronically has led to a corresponding rise in errors in this information. The stored data can be critically important, necessitating new ways of correcting anomalous records. Current cleaning techniques are very domain-specific and hard to extend, hindering their use in some areas. This work proposes an extensible framework for data cleaning, allowing users to customize the cleaning to their specific requirements. It defines categories of common cleaning operations, allowing more robust support for user-implemented cleaning functions in these categories. The experimental results show that the proposed data cleaning framework is an effective approach to cleaning data for arbitrary domains.

To my family ...

Acknowledgements

This endeavor would not have been possible without the assistance of many people. I would like to thank the Department of Computer Science and the University of Illinois at Urbana-Champaign for providing my university education and teaching me the skills to succeed.

Most importantly, I would like to thank my family, for always being there to lend their support, and for being my motivation in making it through my five years at the university.

Table of Contents

Chapter 1 Introduction	1
Chapter 2 Problem Definition.....	4
2.1 Error Types.....	4
2.2 Current Approaches.....	7
2.3 Problem Statement	8
Chapter 3 Conceptual Model.....	10
3.1 Integrity Check (INTEG)	10
3.2 Mapping (MAP)	12
3.3 Merging (MERGE).....	14
3.4 Statistics (STAT).....	17
3.5 Application	20
3.6 Composition	23
Chapter 4 System Architecture	29
4.1 Command Parser	30
4.2 Execution Engine	32
4.3 User Interface	34
Chapter 5 System Evaluation	38
5.1 Soccer Matches Database	39
5.2 Historical Weather Database	42
5.3 Analysis.....	47
Chapter 6 Related Work	49
Chapter 7 Conclusion	51
References.....	53
Appendix A.....	55
Appendix B	58

Chapter 1

Introduction

Since the dawn of recorded history, humans have produced data. With the advent of the information era, the flow of knowledge plays an ever more critical role in the economy, and enormous quantities of data are produced on a daily basis. However, the process of recording and storing such vast amounts of data often leads to inconsistencies in formats and adherence to constraints, due to factors such as multiple data sources and entry errors. These errors and inconsistencies can cause problems ranging from minor to business-critical [1]. The existence of such “dirty” data necessitates a change in the way we manage our data. To this end, the concept of data cleaning, alternatively known as data cleansing or scrubbing, has emerged to describe a broad range of methods to discover and eliminate dirty data.

There is no comprehensive definition of data cleaning, because data cleaning targets errors in data and the definition of what constitutes an error is highly domain specific [2]. No set of operations is sufficient to eliminate dirty data for all possible domains. In general, data cleaning involves the purging of errors and resolution of inconsistencies, followed by transformation into a uniform format before use [3]. The exact type of cleaning varies depending on system implementation, but major categories

include parsing, data transformation, integrity constraint enforcement and statistical methods [4].

With such large volumes of data, manual data cleaning is extremely costly and inefficient. Millions of dollars are spent by organizations every year simply to detect errors. Manually processing data by hand is time consuming, and is itself prone to errors. The need for tools that minimize human involvement in the data cleansing process are necessary and are the only practical and cost effective way to achieve a reasonable quality level in large data sets [5]. As a result, many data cleansing solutions have been developed using highly domain-specific heuristics. These solutions, while powerful, suffer from being too dependent on a particular environment and are complicated, greatly hindering their reuse and extension into other domains [2].

The Data Cleaning Framework proposed in this thesis is a new way of solving the extensibility problem faced when trying to clean data from arbitrary domains. The Data Cleaning Framework is designed with extensibility as the central idea, and it enables users to customize data cleaning operations to meet their needs, rather than trying to adapt to the rules set forth by the system. This thesis provides the following contributions:

- Identifying the obstacles in data cleaning and the limitations of current approaches.
- Generalizing data cleaning operations into major categories that are applicable to most data sets.
- Building a framework using those operation categories to enable users to create custom data cleaning operations and extend existing operations.

The remainder of the thesis is organized as follows. Chapter 2 examines the major challenges in data cleaning and formally defines the problem. Chapter 3 explains the intuition behind the conceptual model for the framework and details the major operation types. Chapter 4 describes the system architecture for the Data Cleaning Framework. Chapter 5 evaluates the framework against two scenarios using real-world data and Chapter 6 describes related work in the field. Finally, Chapter 7 ties all of the work together and concludes the thesis.

Chapter 2

Problem Definition

In order to formally define the problem this thesis tries to solve, we must first examine the concept of “dirty” data, and the common error types that result in dirty data. We must also consider current approaches to the data cleaning problem.

2.1 Error Types

Data cleaning is a broad field with no comprehensive listing of error types. However, a description of the most frequently encountered types of errors follows [2, 6, 7, 8].

Lexical error: Differences between the structure of the data items and the specified format. For example, a relation has n attributes when it should have m attributes, so the structure of the data does not conform to the format. Given the following relations R and I , where R contains a lexical error, I is the ideal schema we want to produce, and a_i denotes the attribute with index i in the relation:

$$R(a_1, \dots, a_n) \qquad I(a_1, \dots, a_m)$$

We define a lexical error as $n \neq m$, meaning that the number of attributes in R differs from the number of attributes in I. This means that the number of attributes in R is either more or less than the number of attributes in I.

Domain format errors: Errors where the given value for an attribute does not conform to the expected domain format. For example, a value is required to be exactly n characters long but instead is m characters long. Given the following relation R, where R contains a domain format error at attribute a_i , and D is the domain of valid formats for a given attribute:

$$R(a_1, \dots, a_i, \dots, a_n) \quad \text{where } a_i \notin D$$

We define a domain format error as $a_i \notin D$, meaning that the value stored in a_i does not belong to a valid format for that particular attribute.

Inconsistencies: Non-uniform use of values, units and abbreviations. Inconsistencies occur if different units are used to record a measurement and are problematic if the values are assumed to be uniform. Given the following relation R, where R contains an inconsistency at attribute a_i , and U denotes a uniform set of the representations for a given attribute:

$$R(a_1, \dots, a_i, \dots, a_n) \quad \text{where } a_i \notin U$$

We define an inconsistency as $a_i \notin U$, meaning that the value stored in a_i does not have a uniform representation associated with that particular attribute. U is a uniform set where all members are recorded with the same representation.

Integrity constraint violations: Relations that do not satisfy one or more real-world restrictions on the set of valid instances. For example, a value is negative even though this would be impossible for real-world data. Given the following relation R, where R contains an integrity constraint violation

at attribute a_i , and c_i is a rule representing a real-world constraint for that attribute value:

$R(a_1, \dots, a_i, \dots, a_n)$ where a_i does not satisfy c_i

We define an integrity constraint violation as a_i does not satisfy the rule c_i , meaning that the value stored in a_i does not satisfy a real-world constraint for that attribute.

Contradictions: Values in a relation or between relations that violate some kind of dependency between the values. Contradictions are a special case of integrity constraint violations where the constraint being violated is directly derived from other attributes in the relation. For example, conflicting values for age and date of birth. Given the following relation R, where attributes a_i and a_j in R contradict each other on some dependency d :

$R(a_1, \dots, a_i, \dots, a_j, \dots, a_n)$ where $a_j \neq d(a_i)$

We define a contradiction as $a_j \neq d(a_i)$, meaning that the value of a_j does not match the value derived from a_i using some dependency d .

Missing values: Omissions of values that should exist after performing data collection, but are not represented. Given the following relation R, where attribute a_i has a missing value:

$R(a_1, \dots, a_i, \dots, a_n)$ where $a_i = \emptyset$

We define a missing value as $a_i = \emptyset$, meaning that a_j does not have a set value even though it is expected to.

2.2 Current Approaches

In the context of detecting and correcting dirty data, there exist four major approaches. Almost all data cleaning solutions available use at least one of these approaches. The four methods are described below [2, 3, 9].

Parsing: Used to detect syntax errors. A parser for a grammar decides for a given string whether it is an element of the language defined by the grammar. In data cleaning, the strings are usually attribute values from a domain. The grammar used for parsing is based on the domain format. Strings which do not correspond to the domain format are syntax errors and have to be corrected.

Data Transformation: Maps data from its given format into the format expected by the application. The transformations can affect the schema of the relations as well as their values. Schema transformation may be used to map data into a new schema better fitting the needs of the intended application. Input data which do not conform to the new schema can be corrected during the transformation. Standardization and normalization are transformations with the intention of removing irregularities in data.

Integrity Constraint Enforcement: Ensures the satisfaction of integrity constraints on a collection of data. The two different approaches are integrity constraint checking and integrity constraint maintenance. Integrity constraint checking rejects transactions that, if applied, would violate some integrity constraint. Integrity constraint maintenance uses a set of integrity constraints to perform modifications on the data collection so that the collection is corrected of integrity constraint violations.

Statistical Methods: Detection and elimination of complex errors often involve relationships between multiple attributes which do not violate integrity constraints. Outliers are one example of such an error. By analyzing the data using the values such as mean, standard deviation, and range, unexpected values may be discovered indicating potential invalid relations. The correction of such errors is often impossible because the true values are unknown. Possible solutions include statistical methods like setting the values to the average or other statistical value.

2.3 Problem Statement

Collections of real-world data encompass an infinite set of domains. Since all data, no matter which domain it comes from, has the potential to be dirty, it is neither feasible nor possible to design a system with a set of data cleaning operations applicable to all domains.

Cleaning can be done through parsing, data transformation, integrity constraint enforcement, and statistical methods. However, none of the methods by itself is comprehensive enough to completely clean a set of data with an arbitrary number of error types. Current data cleaning systems focus specifically on using just a few of approaches, and do so by being highly specific to the problem domain. This makes expansion of application to other domains difficult, as the user does not have an easy way to make modifications to the system functionality.

We make the observation that if a single cleaning method is limited, multiple methods can be used to cover deficiencies in a particular method. As a result, a concept for an extensible data cleaning framework came about. Recognizing the need for users to be able to customize their

data cleaning experience to match any particular domain, extensibility is the main focus of the Data Cleaning Framework described in this thesis.

The main problems that the Data Cleaning Framework addresses:

- Enabling users to combine parsing, data transformation, integrity constraint enforcement, and statistical methods for data cleaning in a single system.
- Allowing users to write new data cleaning functions that can extend the base functionality of the framework to meet their particular cleaning needs.
- Providing a visual interface to let users construct a sequence of commands to perform data cleaning, as well as previewing the effects of their changes to the database.

The notion of customizability will be a key focus in the conceptual model and architecture of the Data Cleaning Framework. Providing an extensible general data cleaning framework is essential to the creation of a cleaning solution suitable for all data domains, and therefore, it is the main problem this thesis attempts to address.

Chapter 3

Conceptual Model

The key feature of the Data Cleaning Framework is its extensibility. In order to go beyond the limitations of a particular domain, we will generalize the problem of data cleaning into categories upon which we will build our framework. We propose four main categories of cleaning operations. These categories are the integrity check, mapping, merging, and statistical operations. We will describe how these four categories of operations will cover the error types described previously.

3.1 Integrity Check (INTEG)

The integrity check operator performs data cleaning on a single attribute. It takes an attribute as input, performs some transformation on that attribute, and then outputs it. This operator is useful when any changes to an attribute do not depend on other attributes. Mapping can theoretically be used to perform integrity checking as it is a one-to-one operation; however, it is useful to define integrity checking as a separate operation because it is more intuitive to think of integrity checking as correcting errors in the values for an attribute rather than mapping a set of corrected values back to the attribute. We represent an integrity check

operation using the symbol φ . An integrity check of relation R on attribute a_i using the function f is defined as follows:

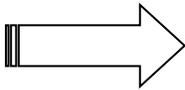
$$\varphi(R(a_1, \dots, a_i, \dots, a_n), i, f) \quad \text{where } f(a_i) = \{ a_i' \}$$

$$= R(a_1, \dots, a_i', \dots, a_n)$$

The parameters to the integrity check operation are the relation R, the attribute a to be modified, represented by its index i , and the user-defined transformation function f . The integrity check operator applies the function f to attribute a_i . The function f takes an attribute a_i and applies some transformation on it, producing a corrected attribute a_i' as output, and updating a_i with it.

Consider Figure 3.1. From the data, it is clear that the Age attribute should not contain negative values. In this case, the most likely course of action would be to convert the negative age into a positive value. This cleaning operation requires no information stored in other relations, as such, the operation falls into the integrity check category.

Name	Age
Alice	17
Bob	-42
Charles	25



Name	Age
Alice	17
Bob	42
Charles	25

Figure 3.1 Integrity Check Operation

In the Data Cleaning Framework, the integrity check operation is represented by the INTEG operator. INTEG utilizes the input column from the source table. For every record, the transformation given by the parameter name will be applied to said list of attributes. Any changes to an attribute's value will overwrite the initial value. The output columns and

destination tables are unnecessary because changes to data are written back into the original attribute column in the same table.

In terms of SQL, the INTEG operator would correspond to a SELECT on the input column from the source table to fetch the relevant data into a ResultSet object. There would then be an UPDATE operation on the input column for each record in the ResultSet, with the new, updated value corresponding to the result returned by applying the user-implemented function on the attribute value for that record.

```
INTEG input_column
SOURCE source_table
PARAM parameter_name [parameter_arg1] [parameter_arg2] ...

Example: INTEG age SOURCE person PARAM positive;

Corresponding SQL:
ResultSet ← SELECT input_column FROM source_table
for each record r in ResultSet
    x ← f(r) // f() is user-defined function
    UPDATE source_table SET r.input_column = x
```

Figure 3.2 INTEG Syntax

3.2 Mapping (MAP)

The mapping operator is a one-to-many operation. Mapping takes a single attribute as input, performs some transformation on the input, and produces one or more attributes based on the input. This operator is useful when one attribute directly affects one or more other attributes during cleaning. We represent a mapping operation using the symbol ω . A mapping on a relation R of an attribute a_i to attributes a_j, \dots, a_k is as follows:

$$\omega(R(a_1, \dots, a_i, \dots, a_n), i, [j, \dots, k], f) \quad \text{where } f(a_i) = \{a_j, \dots, a_k\}$$

$$= R(a_1, \dots, a_i, \dots, a_j, \dots, a_k, \dots, a_n)$$

The parameters to the mapping operation are the relation R, the attribute a_i to be mapped and a list of attributes a_j, \dots, a_k that will store the results of the mapping, all represented by their indices, and the user-defined transformation function f . The mapping operator applies the function f on attribute a_i . The function f takes an attribute a_i and applies some transformation on it, producing attributes a_j, \dots, a_k as output, and adding them to relation R.

Consider Figure 3.3. Originally, only the Name attribute exists. However, if we require both a first name and a last name for each person, then we can split the first and last names, and create the corresponding attributes First Name and Last Name. This operation requires using data stored in one attribute as input, and outputs the results of the transformation to multiple attributes, so it would be a mapping.

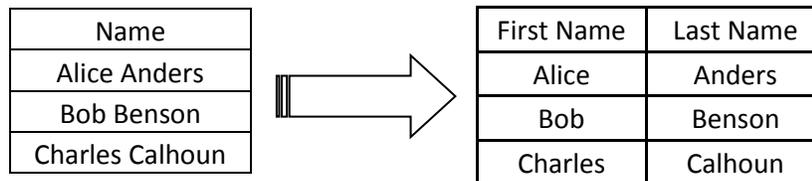


Figure 3.3 Mapping Operation

In the Data Cleaning Framework, the mapping operation is represented by the MAP operator. MAP uses input columns from the source tables. Its outputs are the output columns in the destination tables. For every record, the function given by the parameter name is applied to said attribute. The outputs of the transformation are stored as the appropriate attribute of the corresponding record in the destination tables if destination tables are listed, otherwise output columns will be saved to

the source table. If output columns will be stored in multiple destination tables, the column name must be qualified by the name of the table.

In terms of SQL, the MAP operator would correspond to a SELECT on the input column qualified by the name of the source table to fetch the relevant data into a ResultSet object. If the output columns do not exist, there is an ADD COLUMN query to create the columns the transformed data will be stored in. Finally, there is an UPDATE operation on the output columns for each record in the ResultSet, with the new value corresponding to the result returned by applying the user-implemented function on the attribute value for that record.

```
MAP input_column
TO output_column1 [, output_column2] ...
SOURCE source_table1 [, source_table2] ...
[DEST] [destination_table1] [, destination_table2] ...
PARAM parameter_name [parameter_arg1] [parameter_arg2] ...

Example: MAP name TO first, last SOURCE person PARAM split;

Corresponding SQL:
ResultSet ← SELECT input_column FROM source_table
for each destination_table d
    for each output_column o
        ALTER TABLE d ADD COLUMN o
for each record r in ResultSet
    [x1, ..., xn] ← f(r) // f() is user-defined function
for each destination_table d
    UPDATE d SET r.oi = xi
```

Figure 3.4 MAP Syntax

3.3 Merging (MERGE)

The mapping operator is a many-to-one operation. It takes multiple attributes as input, performs a transformation on the input, and produces one attribute based on the input. This operator is useful when multiple

attributes directly affect a single attribute during a cleaning operation. We represent a merging operation with the symbol μ . A merging on a relation R of attributes a_i, \dots, a_j to an attribute a_k is defined as follows:

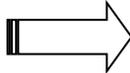
$$\mu(R(a_1, \dots, a_i, \dots, a_j, \dots, a_n), [i, \dots, j], k, f) \quad \text{where } f(a_i, \dots, a_j) = \{a_k\}$$

$$= R(a_1, \dots, a_k, \dots, a_n)$$

The parameters to the merging operation are the relation R, a list of attributes a_i, \dots, a_j to be merged and the attribute a_k that will store the results of the merge, all represented by their indices, and the user-defined transformation function f . The merging operator applies the function f to attributes a_i, \dots, a_j . The function f takes multiple attributes a_i, \dots, a_j and applies some transformation on them, producing a single attribute a_k as output, and adds it to relation R.

Consider Figure 3.5. The original data contains the fields Q1-Q4 Profit. However, if we need an Annual Profit attribute, we can combine the data for Q1-Q4 profit to produce that attribute. We perform an addition on the Q1-Q4 profit attributes to generate the Annual Profit attribute. This operation requires using data stored in multiple attributes as input, and outputs the results of the transformation to a single attribute, so it can be categorized as a merging operation.

Company	Q1	Q2	Q3	Q4
Able	100	80	20	30
Baker	25	135	200	210
Charlie	60	65	55	70



Company	Annual Profit
Able	230
Baker	570
Charlie	250

Figure 3.5 Merging Operation

In the Data Cleaning Framework, the merging operation is represented by the MERGE operator. MERGE uses the input columns

from the source tables. Its output is the output column in the destination tables. For every record, the transformation given by the parameter name will be applied to the input columns. The output of the transformation is stored as the output attribute of the corresponding record in the destination table if destination tables are listed; otherwise the output column is saved to the source table. If input columns originate from multiple source tables, the column name must be qualified by the name of its table.

In terms of SQL, the MERGE operator would correspond to a SELECT on the input columns qualified by the name of the source tables they are from to fetch the relevant data into a ResultSet object. If the output column does not exist, there is an ADD COLUMN query to create the column the transformed data will be stored in. Finally, there is an UPDATE operation on the output columns for each record in the ResultSet, with the new value equal to the result returned by applying the user-implemented function on all specified attribute values for that record.

```

MERGE input_column1 [, input_column2] ...
TO output_column1
SOURCE source_table1 [, source_table2] ...
[DEST] [destination_table1] [, destination_table2] ...
PARAM parameter_name [parameter_arg1] [parameter_arg2] ...

Example: MERGE q1_profit, q2_profit, q3_profit, q4_profit
TO annual_profit SOURCE q1-q4 DEST annual PARAM add;

Corresponding SQL:
ResultSet ← SELECT input_column1 [, input_column2] ...
              FROM source_table1 [, source_table2] ...
for each destination_table d
    ALTER TABLE d ADD COLUMN output_column
for each record r in ResultSet
    x ← f(r)          // f() is user-defined function
    for each destination_table d
        UPDATE d SET r.output_column = x

```

Figure 3.6 MERGE Syntax

3.4 Statistics (STAT)

The statistics operator performs data analysis and statistical inference. STAT uses an input attribute, along with the specified dependency attributes of that attribute. For every record, it calculates some statistic on the input attribute using a subset of relations where the attribute values specified as dependencies in these relations match the corresponding values of the input relation. We represent a statistical operation with the symbol δ . Let $R_x \cdot a_i$ denote the i th attribute of relation R_x . A statistics operation on attribute a_i of a relation R_x with dependencies a_j, \dots, a_k is defined as follows:

$$\delta(R_x(a_1, \dots, a_i, \dots, a_j, \dots, a_k, \dots, a_n), i, [j, \dots, k], f)$$

$$\text{where } f(\sum_{S=1}^t R_S \cdot a_i) = z \quad \text{such that } R_S \cdot a_j = R_x \cdot a_j, \dots, R_S \cdot a_k = R_x \cdot a_k \\ = (R_x \cdot a_i, z)$$

The parameters to the statistics operation are the relation R_x , the attribute a_i that the statistical measure will be computed upon, and a list of dependencies a_j, \dots, a_k , for which the statistical measure on a_i depends on, all represented by their indices. The final parameter is the user-defined transformation function f . Note that in the notation, Σ does not denote summation, but rather an aggregation over the relations. The statistics operator applies the user-function f on the subset of relations which satisfy the dependencies for the input attribute for a particular record. The user-function f performs a statistical aggregation on a_i over all relations in the satisfying subset and then stores that value into a mapping of the attribute a_i for a relation R_x to its inferred value z based on the dependencies a_j, \dots, a_k . This mapping can be accessed by successive operations to modify data.

Consider Figure 3.7. We have a missing value in the table for zip code. However, we can use the information available, such as the existing city to zip code pairings to infer a value. We can use a statistical function such as the frequency, to infer a value that will minimize the chance of an anomaly being introduced. Once we have calculated the frequency, we can store this value so that a future operation, such as an integrity check, can access this result. These types of operations leverage statistical methods, so they fall under the statistics category.

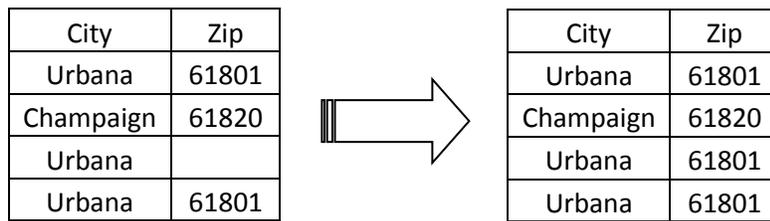


Figure 3.7 Statistical Operation

In the Data Cleaning Framework, the statistics operation is represented by the STAT operator. STAT uses the input column from the source table, along with the specified dependency attributes of that attribute from the destination table. For every record, the operation given by the parameter name will be applied as an aggregation upon the subset of records satisfying the specified dependencies on the input record. Its output stores a mapping of an attribute for a particular record and the statistical measure calculated on the subset of records satisfying the dependencies of that attribute. These statistical measures are computed programmatically on the input attribute for each record, based on the dependencies for that particular record.

In terms of SQL, the STAT operator would correspond to a SELECT on the input column and dependency columns from the source and destination tables, where the values of the dependency columns match between the source and destination tables. These records are grouped as subsets into ResultSet objects based on the dependency columns. Following is a user-implemented aggregation operation on the input columns in each ResultSet. The results of the aggregation are stored as a mapping of an attribute for a particular record to the statistical measure computed on that attribute based on the dependencies listed. The results can later be used by any of the other operation types to modify the data.

```

STAT input_column
SOURCE source_table1
[DEST] [destination_table] [, destination_table2] ...
DEPEND dependency_column1 [, dependency_column2] ...
PARAM parameter_name [parameter_arg1] [parameter_arg2] ...

Example: STAT zip_code SOURCE address DEPEND city
        PARAM maxFreq;

Corresponding SQL:
ResultSet ← SELECT input_column
              FROM source_table, destination_table ...
              WHERE source_table.dependency_column1 =
                    destination_table.dependency_column1 ...
              GROUP BY dependency_column1 ...
for each record r in ResultSet
    (r.input_column, x) ← f(ResultSet)

// f() is user-defined function which returns a mapping of
an attribute to the statistical measure computed by f() on
that attribute using the specified dependencies. The
mapping is stored automatically and can be accessed by
successive cleaning operations

```

Figure 3.8 STAT Syntax

3.5 Application

The integrity check, mapping, merging, and statistical operators form the core of the Data Cleaning Framework, and provide the means with which to clean errors. We will show algebraically how the four operator types can handle the error types listed in the taxonomy. Each error type is matched with an operator that can handle them.

Lexical error – A mapping/merging operation can be used to change the structure of data to fit schema. Mapping/merging operations modify the schema of the data by creating or combining attributes to fix lexical errors:

$$\omega(R(a_1, \dots, a_i, \dots, a_n), i, [j, \dots, k], f) \quad \text{where } f(a_i) = \{a_j, \dots, a_k\}$$

$$= R(a_1, \dots, a_j, \dots, a_k, \dots, a_n)$$

The mapping operator applies the function f to map the attribute a_i to attributes a_j, \dots, a_k in order to extract the missing attributes that were previously combined as a single attribute and fix the lexical error.

$$\mu(R(a_1, \dots, a_i, \dots, a_j, \dots, a_n), [i, \dots, j], k, f) \quad \text{where } f(a_i, \dots, a_j) = \{a_k\}$$

$$= R(a_1, \dots, a_k, \dots, a_n)$$

The merging operator applies the function f to attributes a_i, \dots, a_j in order to combine extraneous attributes into an attribute a_k and fix the lexical error.

Domain format error – An integrity check operation can be used to clean the error by forcing format constraints upon values. Correcting domain format errors is a one-to-one operation, so we can use the integrity check:

$$\varphi(R(a_1, \dots, a_i, \dots, a_n), i, f) \quad \text{where } f(a_i) = \{a_i'\} \text{ and } a_i' \in D$$

$$= R(a_1, \dots, a_i', \dots, a_n) \quad \text{where } a_i' \in D$$

The integrity check operator applies the user-defined function f on attribute a_i which has a domain format error, and the transformation

performed by f produces a_i' , where a_i' is constrained within the boundaries of the format domain D.

Inconsistencies – An integrity check operation can be used to clean the error by converting values into the required units. Inconsistencies do not need additional information from other attributes in order to be corrected, so integrity check is suitable:

$$\varphi(R(a_1, \dots, a_i, \dots, a_n), i, f) \quad \text{where } f(a_i) = \{a_i'\} \text{ and } a_i' \in U \\ = R(a_1, \dots, a_i', \dots, a_n) \quad \text{where } a_i' \in U$$

The integrity check operator applies the user-defined function f on attribute a_i , which has inconsistent representation, and the transformation performed by f converts a_i into a_i' , where a_i' belongs to U, where U is a uniform set where all members are recorded with the same representation.

Integrity constraint violations – An integrity check operation can be used to clean the error by transforming invalid attribute values to valid ones. Correcting domain format errors does not require additional information, so it is one-to-one, and we can use the integrity check:

$$\varphi(R(a_1, \dots, a_i, \dots, a_n), i, f) \quad \text{where } f(a_i) = \{a_i'\} \text{ and } a_i' \text{ satisfies } c_i \\ = R(a_1, \dots, a_i', \dots, a_n) \quad \text{where } a_i' \text{ satisfies } c_i$$

The integrity check operator applies the user-defined function f on attribute a_i , which violates an integrity constraint c_i . The transformation performed by f produces a_i' , where a_i' satisfies a real-world constraint c_i .

Contradictions – A mapping/merging operation can be used to fix contradictory values by deriving them again from other attributes. Contradictions require dependencies between attributes, so operators which operate on multiple attributes, such as mapping/merging are needed:

$$\omega(R(a_1, \dots, a_i, \dots, a_j, \dots, a_k, \dots, a_n), i, [j, \dots, k], f) \text{ where } f(a_i) = \{a_j', \dots, a_k'\}$$

$$= R(a_1, \dots, a_i, \dots, a_j', \dots, a_k', \dots, a_n)$$

The mapping operator applies the user-defined function f to a_i in order to derive correct values for a_j, \dots, a_k that do not contradict a_i .

$$\mu(R(a_1, \dots, a_i, \dots, a_j, \dots, a_k, \dots, a_n), [i, \dots, j], k, f) \text{ where } f(a_i, \dots, a_j) = \{a_k'\}$$

$$= R(a_1, \dots, a_i, \dots, a_j, \dots, a_k', \dots, a_n)$$

The merging operator applies the user-defined function f to attributes a_i, \dots, a_j to derive a correct value for a_k , which does not contradict a_i, \dots, a_j .

Missing values – A statistics operation can be combined with a data modifying operator to fill in a missing value by inferring a likely value using information about existing values. For example, a missing zip code in an address can be inferred if we know the city, by finding the most frequently paired zip code for that particular city. The statistics operator is first used to infer a value by considering the dependencies of the missing value for a specific record, then performing statistical aggregation on other records whose attributes match on the dependencies. It stores the mapping of an attribute of a particular record with its inferred value. This mapping is automatically stored and can be accessed by data modification operators such as an integrity check to fill in the missing value.

$$\delta(R_x(a_1, \dots, a_i, \dots, a_j, \dots, a_k, \dots, a_n), i, [j, \dots, k], f)$$

$$\text{where } f(\sum_{s=1}^t R_s \cdot a_i) = z \quad \text{such that } R_s \cdot a_j = R_x \cdot a_j, \dots, R_s \cdot a_k = R_x \cdot a_k$$

$$= (R_x \cdot a_i, z)$$

$$\varphi(R_x(a_1, \dots, a_i, \dots, a_n), i, f) \quad \text{where } a_i = \emptyset \text{ and } f(a_i) = \{a_i'\} \text{ and } a_i' = z$$

$$= R(a_1, \dots, a_i', \dots, a_n)$$

The statistics operator applies the user-function f on the subset of relations which satisfy the dependencies for the input attribute for a particular record. The user-function f performs a statistical aggregation on

a_i over all relations in the satisfying subset and then stores that value into a mapping of the attribute a_i for a relation R_x to its inferred value z based on the dependencies a_j, \dots, a_k . Following is an integrity check operation, which is automatically passed the attribute/value mappings by the system. The integrity check fills missing values for an attribute in a particular record with the value computed by the statistics operator based on other records matching the dependencies on that attribute.

3.6 Composition

A data cleaning program using the Data Cleaning Framework is defined as one or more commands run in sequence. Several commands may be composed into a data cleaning program by running them in sequence to handle multiple error types. Successive commands operate on the state of the database resulting from the previous command. We will show mathematically and by example, how a series of operations can be composed into a data cleaning program.

We will use the example data shown in Figure 3.9 to demonstrate how the Data Cleaning Framework and its operators function. Suppose we have a series of cleaning tasks we wish to perform on this data. We want to split the “Name” field into “First Name” and “Last Name”. We also want to infer a value for the missing “Zip” entry, and then finally to fill in the missing value using that information. We can accomplish this by performing the sequence of commands shown in Figure 3.10.

Name	City	Zip
Alice Anders	Urbana	61801
Bob Benson	Champaign	61820
Charles Calhoun	Urbana	
Daniel Dillinger	Urbana	61801

Figure 3.9 Person Table

```
MAP name TO firstName, lastName SOURCE person PARAM split;
STAT zip SOURCE person DEPEND city PARAM maxFreq;
INTEG zip SOURCE person PARAM fillEmpty;
```

Figure 3.10 Example Cleaning Operations

The first operation is a MAP operation, which takes a single attribute as input, performs some transformation on the input, and produces one or more attributes based on the input. For this operation, the “Name” attribute from the “Person” table is being mapped to the “First Name” and “Last Name” attributes using the parameter “Split”. Parameters specify the user-defined function which performs the transformation on the data, in this case, splitting. Users can define their own transformation functions to use for data cleaning. We will show an example of the mathematical representation for MAP on the first record:

$$\omega(R(\text{Alice Anders}, \text{Urbana}, 61801), 1, \text{split})$$

where $\text{split}(\text{“Alice Anders”}) = \{\text{“Alice”}, \text{“Anders”}\}$

$$= R(\text{Alice}, \text{Anders}, \text{Urbana}, 61801)$$

In this mapping operation, the first attribute is being mapped, and the transformation function being applied is the “Split” function, so $i = 1$ and $f = \text{split}$. The “Split” function splits “Alice Anders” into “Alice”, “Anders”, and the resulting record becomes $R(\text{Alice}, \text{Anders}, \text{Urbana}, 61801)$. The MAP operation is applied to all records in the table, and the

results are shown in Figure 3.11. The MERGE operation is logically the reverse of the MAP operation, so its specification is similar and not shown.

First Name	Last Name	City	Zip
Alice	Anders	Urbana	61801
Bob	Benson	Champaign	61820
Charles	Calhoun	Urbana	
Daniel	Dillinger	Urbana	61801

Figure 3.11 Person Table after MAP

Next is the STAT operation, which for every record, calculates some statistic on an attribute using a subset of relations where the attribute values specified as dependencies in these relations match the corresponding values of the input record. Each operation operates on the state of the data resulting from the previous operation, so the STAT operation would operate on Figure 3.11. For this operation, a record is missing the “Zip” attribute. The “Zip” values for this record will be inferred by computing some statistical aggregation over all records which match the dependency attribute “City” for that particular record, in this case the most frequent value of “Zip” corresponding to the “City” value. Since the record with a missing “Zip” value has “City = Urbana”, the system will only use other records satisfying “City = Urbana” to infer the missing value, and will not use records with “City = Champaign”. We show an example of the algebraic notation for STAT on the third record:

$$\delta(R_x(\text{Charles}, \text{Calhoun}, \text{Urbana}, \emptyset), 4, [3], \text{maxFreq})$$

where $a_3 = \text{“City”}$ $a_4 = \text{“Zip”}$ $R_s.city = R_x.city$ and $R_x.city = \text{“Urbana”}$

$$\text{and } \text{maxFreq}(\sum_{s=1}^t R_s.zip) = 61801$$

$$= (R_x.zip, 61801)$$

In this statistics operation, the attribute we want to infer, “Zip”, is the fourth attribute; its dependencies are “City”, the third attribute. The transformation function being applied is the “maxFreq” function, so $i = 4$ and $f = \text{maxFreq}$. The system selects a subset of records satisfying the dependency, or records with “City = Urbana”. This is passed to the “maxFreq” function which computes the value of “Zip” most frequently paired with “City = Urbana”. The STAT operation is applied to all records in the table, and it computes and saves a mapping of attributes for a particular record to the statistical measures computed on that attribute based on the dependencies listed. The internal representation of such a mapping is shown in Figure 3.12. This mapping is accessible by any successive operation to be used in data cleaning.

Attribute	Inferred Value
R ₁ .zip	61801
R ₂ .zip	61820
R ₃ .zip	61801
R ₄ .zip	61801

Figure 3.12 Mapping Produced by STAT

STAT is unique in that it stores the result of its calculations and automatically passes this information so that successive operators can use it to perform cleaning. Because operations are modular, the STAT operation does not need to modify data itself, but instead can pass what it computes to data-modifying operators such as INTEG. This chaining of operators reduces the amount of code duplication, because STAT can rely on the data-modifying operations to apply the results of its computation. Therefore, if a user wishes to alter data using the results of the STAT

operation, they must follow the STAT operation with a data-modifying operation, lest the result be overwritten by another STAT operation.

The final operation is an INTEG operation, which takes an attribute as input, performs some transformation on that attribute, and then outputs it. The INTEG operation is a data-modifying operation, so it can be used to apply the results computed by the preceding STAT operation and use them to fill missing values. In this operation, the “Zip” attribute in the Person table is being cleaned using the “fillEmpty” user-defined function. We will show an example of the mathematical representation for INTEG on the third record:

$$\varphi(R(\text{Charles}, \text{Calhoun}, \text{Urbana}, \emptyset), 4, \text{fillEmpty})$$

$$\text{where } \text{fillEmpty}(a_i, 61801) = a_i' \text{ and } a_i' = 61801$$

$$= R(\text{Charles}, \text{Calhoun}, \text{Urbana}, 61801)$$

In this integrity check operation, the fourth attribute is being cleaned, and the transformation function being applied is the “fillEmpty” function, so $i = 4$ and $f = \text{fillEmpty}$. The “fillEmpty” function is automatically passed the mapping derived from the previous STAT operation, and it fills in missing values of a specific attribute in a record with the corresponding value defined in the mapping. The INTEG operation is applied to all records, and the results are shown in Figure 3.13.

First Name	Last Name	City	Zip
Alice	Anders	Urbana	61801
Bob	Benson	Champaign	61820
Charles	Calhoun	Urbana	61801
Daniel	Dillinger	Urbana	61801

Figure 3.13 Person Table after INTEG

The data was successfully cleaned after applying a sequence of operations which compose a data cleaning program. Any transformation that can be performed using the system falls into one of the four categories of data cleaning operations. The operations overlap in their coverage of data errors, so some errors may be cleaned using different operators than the ones listed. Together, these four types of operations can be chained to create a data cleaning program to clean the error types described.

Chapter 4

System Architecture

The Data Cleaning Framework consists of three components. These are the Command Parser, the Execution Engine, and the User Interface. The Command Parser and the Execution Engine contain all of the logic to perform the data cleaning, while the User Interface allows users to specify the cleaning operations and see the changes made.

To use the Data Cleaning Framework, a user builds a sequence of cleaning commands to be run on the database. When the user executes this sequence, the Command Parser extracts the necessary information from the command, such as the inputs and outputs of the operation. It passes this information to the Execution Engine, which decides the type of operator, and the parameters involved in the cleaning. The Execution Engine applies the user-specified transformation on the data in accordance to the operator type. Finally, the User Interface allows the user to view the end results of the cleaning commands.

The system was programmed with the Java language on a Linux system. Java was chosen because of its portability across systems, and its library of supported functions. It was evaluated on a MySQL database and requires the use of JDBC drivers to connect and interact with the database.

4.1 Command Parser

The Command Parser is the first component of the system run when executing a command. It is responsible for extracting the key arguments contained in each command, and separating this information into parts that will later be used by the Execution Engine. The Data Cleaning Framework uses a unique SQL-like syntax for each command. Figure 4.1 shows the structure of this syntax. Any field listed in square brackets may be optional depending on the type of operator used.

```
OPERATOR input_column1 [, input_column2] ...
[TO] [output_column1] [, output_column2] ...
SOURCE source_table1 [, source_table2] ...
[DEST] [destination_table1] [, destination_table2] ...
[DEPEND] [dependency_column1] [, dependency_column2] ...
PARAM parameter_name [parameter_arg1] [parameter_arg2] ...
```

Figure 4.1 Operation Syntax

The syntax used by the Data Cleaning Framework can be divided into seven sections. These are the operator, input columns, output columns, source tables, destination tables, dependency columns, and parameters.

The OPERATOR keyword refers to any of the four supported cleaning operators. These consist of INTEG (integrity check), MAP (mapping), MERGE (merging), and STAT (statistics). The OPERATOR keyword identifies the operation category, and its general effect.

The input columns are between the OPERATOR and TO (or SOURCE) keywords. These are the list of columns that will be read from during the cleaning operation. The specified columns are what the user-specified transformation will operate on. There must be at least one column specified for a command to operate on.

```
input column1 [, input column2] ...
```

Figure 4.2 Input Columns

The output columns lie between the TO and SOURCE keywords. These are the list of columns that will be written to during the cleaning operation. The specified columns indicate where the results of the user-specified transformation will be stored to.

```
[TO] [output column1] [, output column2] ...
```

Figure 4.3 Output Columns

The source tables fall between the SOURCE and DEST (or PARAM) keywords. The list of tables specifies where the input columns are located. This list tells the system which tables the columns to be used as input for the cleaning operation will be found in. There must be at least one table specified for a command to operate on.

```
SOURCE source table1 [, source table2] ...
```

Figure 4.4 Source Tables

The destination tables are between the DEST and PARAM (or DEPEND) keywords. The list of tables is where the output columns will be stored. This list tells the system which tables the columns that will store the outputs for the cleaning operation will be located in.

```
[DEST] [destination table1] [, destination table2] ...
```

Figure 4.5 Destination Tables

The dependency columns are between the DEPEND and PARAM keywords. The list of columns indicates which attributes that the input

column depends on. This list tells the system how to select a subset of records in order to calculate some statistical measure for a column.

```
[DEPEND] [dependency_column1] [, dependency_column2] ...
```

Figure 4.6 Dependency Columns

After the PARAM keyword is the name of the user-specified function to be applied. Immediately following the name is the list of arguments for the user-specified transformation. Any arguments following the PARAM keyword are used solely by the user-implemented function. As such, there is no set format on the parameters being passed. A user-defined function could require no arguments or several arguments, depending on exactly what computation is being performed and what additional information is required. However, each cleaning command requires that a function be specified, regardless of arguments.

```
PARAM parameter_name [parameter_arg1] [parameter_arg2] ...  
// number of arguments vary by function  
Example: // no argument, add all attributes  
PARAM add  
// single argument, round to 2 decimal places  
PARAM round 2
```

Figure 4.7 Parameter Arguments

4.2 Execution Engine

The Execution Engine is responsible for running user-specified transformation functions to perform the data cleaning. The information necessary for the cleaning transformation is extracted from the command by the Command Parser, and passed to the Execution Engine. It uses this information to determine the operator category and the type of

transformation function. It then runs the user-implemented code to perform the corresponding transformation on the relevant data.

The Execution Engine functions by taking the input columns and source tables, and creating an SQL SELECT query used to fetch the relevant data to be cleaned. The data from the query is stored as a ResultSet object, which can be accessed by the user-specified cleaning function. The cleaning function reads from the ResultSet object and performs some computation on the data. The process for obtaining the input data is the same all four operator types. The new values are stored to the output columns and destination tables specified, and the change is reflected in the database.

The engine is not limited to a set of predetermined cleaning transformations on the data. It gives the user the flexibility of integrating additional functionality through user-implemented transformation functions. Figure 4.8 shows an example of how a user might choose to implement a new cleaning transformation. The basic code template with which to implement a transformation has two parts.

```
private void add(ResultSet rs, int currRow, String[] modAttrs,
                String table, String[] paramCons, Object prevResults)
    throws SQLException {
    double value = 0;
    for (int i = 0; i < numCols; i++) {
        value += rs.getDouble(i);
    }
    update(value, currRow, modAttrs, table);
}
```

Figure 4.8 Example User-implemented Function

The first part is the argument section. This contains the set of relevant input data and output attributes, as well as the destination table

where the results of the user-specified function will be stored. It also includes an object containing any results stored by a previous operation, such as statistics calculated and stored for later use in inferring missing values. This information is automatically passed to the user-defined function, and can be accessed as necessary by the user-defined function.

The second part is the functionality section. This is where the core functionality of the user-implemented transformation function is located. Here, the user implements exactly the operations they want to perform upon the data. Implementation is straightforward, since the users need focus solely on the actions needed to clean a single record while the Execution Engine applies the user function to the relevant records in the database. This abstracts away the need to manually get information about the schema format, and then iterate through all of the records.

The Execution Engine provides a separation of concerns when implementing new data cleaning transformations. By categorizing all commands as INTEG, MAP, MERGE, or STAT, the engine knows exactly how to apply the user-defined function to the data. This makes code for user-implemented functionality much shorter, because it only needs to deal with single records, while the Execution Engine does the rest.

4.3 User Interface

The user interface, shown in Figure 4.9, is the component of the system that a user will most frequently interact with. It is designed to help make the steps involved in data cleaning as intuitive as possible. The graphical user interface can be used to view the contents of the database, create a sequence of cleaning commands, and execute or preview the

effects of the commands on the database. The user interface consists of three main areas, or panels.

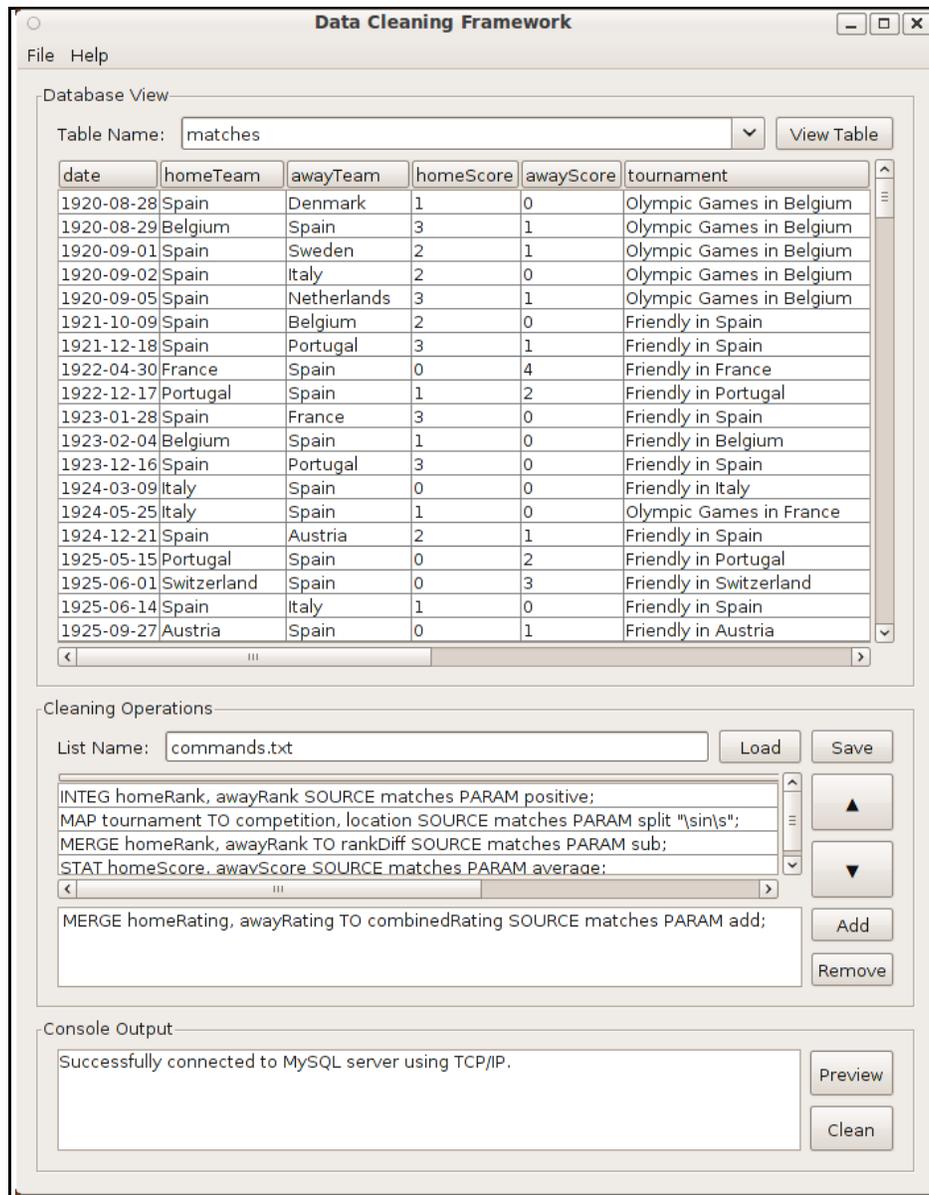


Figure 4.9 Data Cleaning Framework User Interface

The topmost area in the application is the Database View panel, shown in Figure 4.10. Once users have established a connection to the

database, they can choose to select a table within that database from the drop-down list, and view the current contents of the table. This provides users with a clear picture of the data they are operating upon, and gives them a guide with which to write the necessary cleaning commands.

Database View

Table Name: matches View Table

date	homeTeam	awayTeam	homeScore	awayScore	tournament
1920-08-28	Spain	Denmark	1	0	Olympic Games in Belgium
1920-08-29	Belgium	Spain	3	1	Olympic Games in Belgium
1920-09-01	Spain	Sweden	2	1	Olympic Games in Belgium
1920-09-02	Spain	Italy	2	0	Olympic Games in Belgium
1920-09-05	Spain	Netherlands	3	1	Olympic Games in Belgium
1921-10-09	Spain	Belgium	2	0	Friendly in Spain
1921-12-18	Spain	Portugal	3	1	Friendly in Spain
1922-04-30	France	Spain	0	4	Friendly in France
1922-12-17	Portugal	Spain	1	2	Friendly in Portugal
1923-01-28	Spain	France	3	0	Friendly in Spain
1923-02-04	Belgium	Spain	1	0	Friendly in Belgium
1923-12-16	Spain	Portugal	3	0	Friendly in Spain
1924-03-09	Italy	Spain	0	0	Friendly in Italy
1924-05-25	Italy	Spain	1	0	Olympic Games in France
1924-12-21	Spain	Austria	2	1	Friendly in Spain
1925-05-15	Portugal	Spain	0	2	Friendly in Portugal
1925-06-01	Switzerland	Spain	0	3	Friendly in Switzerland
1925-06-14	Spain	Italy	1	0	Friendly in Spain
1925-09-27	Austria	Spain	0	1	Friendly in Austria

Figure 4.10 Database View Panel

Under the Database View panel is the Cleaning Operations panel, seen in Figure 4.11. This panel gives users the ability to load a previously created list of commands from a flat file, or build a new list. The upper pane shows the current contents of the command list, while the bottom pane is where users can type new commands to add. Each command in the list may be reordered or removed to suit the situation. During execution of the command list, each command is processed sequentially. This means that the results of each command are already written to the database by the time the successive command is run. Each command operates on the state of the database resulting from the previous command.

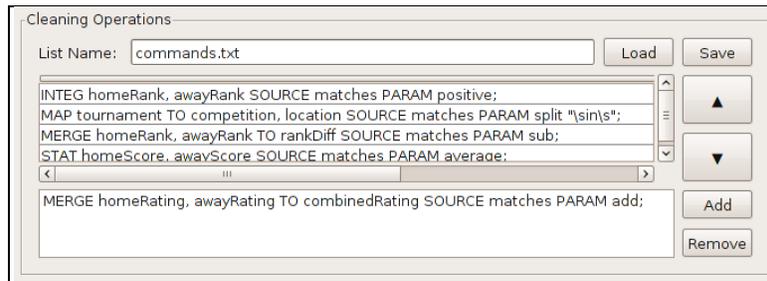


Figure 4.11 Cleaning Operations Panel

The lowermost panel is the Execution Console panel, shown in Figure 4.12. It allows execution and preview of the data cleaning commands. The text area outputs messages and results generated by the cleaning during run time. Cleaning may be performed in preview mode, in which case a new window opens, showing a preview of the contents of any table in the database after cleaning completes. This does not modify any data, and is a convenient way for users to understand exactly what their commands will do. The cleaning commands can then be run with permanent effects, updating the database with the results of the cleaning.

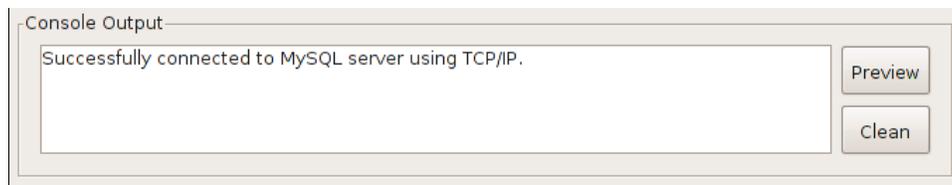


Figure 4.12 Execution Console Panel

The Command Parser, the Execution Engine, and the User Interface combine to form the Data Cleaning Framework. They are designed to minimize the learning curve of extending and customizing the system. These parts constitute the functionality required for the system to be able to be tailored to users' data cleaning needs.

Chapter 5

System Evaluation

To evaluate how the Data Cleaning Framework can be used to perform data cleaning, we will examine two scenarios involving real-world data. For each set of data, we build a sequence of cleaning commands and then apply these commands on the data, with a before and after comparison of the tables to see the end results of the data cleaning. The goal of these tests is to demonstrate how the system and the operator types defined can be applied in a real-world situation.

The Data Cleaning Framework will also be compared to a baseline, in order to evaluate the utility gained by using such a framework. The baseline represents a system where each individual cleaning operation is implemented without utilizing the functionality provided by the Data Cleaning Framework. The baseline uses the same basic code for each cleaning function, but it does not take advantage of the abstractions provided by the framework, such as the automatic fetching and iteration through records, and the functions provided to modify record values. The main metric for this comparison is an estimate of the amount of code necessary to implement the specified functionality. A comparison matrix is used to match the Data Cleaning Framework system against the baseline system for each cleaning command performed in the scenarios.

5.1 Soccer Matches Database

Suppose you are a fan of association football (soccer). You often debate with your friends about the best team in the world. Recently, you discovered a resource that may be able to help settle these disputes. This source of information is the World Football Elo Rankings. The World Football Elo Ratings are based on the rating system used to rank chess players and are used to rate national teams. The World Football Elo Rankings database contains data on thousands of soccer matches. Figure 5.1 shows a selection of records from the table.

Date	Home Team	Away Team	Home Score	Away Score	Competition	Location	Home Rank	Away Rank	Rank Diff
1940-04-02	Croatia	Switzerland	4	0	Friendly	Croatia	8	18	-10
1940-04-21	Switzerland	Croatia	0		Friendly	Switzerland	18	8	10
05/02/1940	Hungary	Croatia	1	0	Friendly	Hungary	8	9	-1
1940-12-08	Croatia	Hungary	1	1	Friendly	Croatia	9	8	1
1941-06-15	Germany	Croatia	5	1	Friendly	Germany	8	9	-1
1941-09-07	Slovakia	Croatia	1	1	Friendly	Slovakia	27	10	17
1941-09-28	Croatia Slovakia		5	2	Friendly	Croatia	9	27	-18
1942-01-18	Germany	Croatia	2	0	Friendly	Germany	8	11	-3
1942-04-05	Italy	Croatia	4	0	Friendly	Italy	2	13	-11
1942-04-11	Croatia	Bulgaria	6	0	Friendly	Croatia	11	44	-3
1942-06-07	Slovakia	Croatia	1.1	2	Friendly	Slovakia	33	9	2
1942-06-14	Hungary	Croatia	1	1	Friendly	Hungary	-13	9	4
1942-09-06	Croatia	Slovakia	6	1	Friendly	Croatia	9	33	-24
1942-10-11	Romania	Croatia	2	2	Friendly	Romania	39	9	30
1942-11-01	Germany	Croatia	5	1	Friendly	Germany	7	11	-4

Figure 5.1 Original Matches Table (First 15 Records)

There are several errors and anomalies in the data due to the time period some of these records are from. The discrepancies are highlighted in red above. To resolve these issues, you decide to clean the data under the following requirements corresponding to the indicated error type:

- Ensure all matches have a home team and an away team listed by splitting team entries containing both teams (Lexical error).
- Change all home scores to integer values by truncating unnecessary digits (Domain format error).
- Convert all date values to yyyy-mm-dd format from other date formats (Inconsistency).
- Ensure all home rankings are non-negative by changing negative values to positive (Integrity constraint violation).
- Correct errors for the rank difference between teams by deriving from the home and away ranking (Contradiction).
- Fill missing away scores to minimize change to statistical measures by using the most common value (Missing value).

Figure 5.2 shows the sequence of actions taken to clean the data. The four categories of operations are used with different parameters in order to correct each type of error. For missing values, the output of the STAT operations is used as inputs to fill empty values during the next round of cleaning. The final cleaned matches table is shown in Figure 5.3.

```
// Split entry into home and away team
MAP homeTeam TO homeTeam, awayTeam SOURCE soccer
  PARAM split;
// Convert all values to integers
INTEG homeScore SOURCE soccer PARAM toInt;
// Change date values to yyyy-mm-dd format
INTEG date SOURCE soccer PARAM ymd;
// Make all values positive
```

```

INTEG homeRank SOURCE soccer PARAM positive;
// Rederive difference from home and away rankings
MERGE homeRank, awayRank TO rankDiff SOURCE soccer
PARAM sub;
// Infer most common value using attribute dependencies
STAT awayScore SOURCE soccer DEPEND awayTeam PARAM maxFreq;
// Use inferred values to fill missing values
INTEG awayScore SOURCE soccer PARAM fillEmpty;

```

Figure 5.2 Data Cleaning Sequence of Actions

Date	Home Team	Away Team	Home Score	Away Score	Competition	Location	Home Rank	Away Rank	Rank Diff
1940-04-02	Croatia	Switzerland	4	0	Friendly	Croatia	8	18	-10
1940-04-21	Switzerland	Croatia	0	1	Friendly	Switzerland	18	8	10
1940-05-02	Hungary	Croatia	1	0	Friendly	Hungary	8	9	-1
1940-12-08	Croatia	Hungary	1	1	Friendly	Croatia	9	8	1
1941-06-15	Germany	Croatia	5	1	Friendly	Germany	8	9	-1
1941-09-07	Slovakia	Croatia	1	1	Friendly	Slovakia	27	10	17
1941-09-28	Croatia	Slovakia	5	2	Friendly	Croatia	9	27	-18
1942-01-18	Germany	Croatia	2	0	Friendly	Germany	8	11	-3
1942-04-05	Italy	Croatia	4	0	Friendly	Italy	2	13	-11
1942-04-11	Croatia	Bulgaria	6	0	Friendly	Croatia	11	44	-33
1942-06-07	Slovakia	Croatia	1	2	Friendly	Slovakia	33	9	24
1942-06-14	Hungary	Croatia	1	1	Friendly	Hungary	13	9	4
1942-09-06	Croatia	Slovakia	6	1	Friendly	Croatia	9	33	-24
1942-10-11	Romania	Croatia	2	2	Friendly	Romania	39	9	30
1942-11-01	Germany	Croatia	5	1	Friendly	Germany	7	11	-4

Figure 5.3 Cleaned Matches Table (First 15 Records)

Data cleaning operations can be combined by running them in sequence. Each operation cleaned one type of error, and successive operations operate on the table after the error handled by the previous operation is cleaned. Figure 5.3 shows that the Data Cleaning Framework

was successful in accomplishing the requirements for correcting all errors. The matrix in Figure 5.4 compares the amount of work required to implement each of the operations necessary to clean the data. We observe that the Data Cleaning Framework is able to perform the required cleaning operations through user extension of the system, with a minimal amount of programming. In contrast, implementing the cleaning functions in the baseline system took many times the effort. Example comparison code from the MAP operation is shown in Appendix A.

Operation	Data Cleaning Framework	Baseline (No Framework)
Map team field to create home and away teams	Lines of Code: ~5	Lines of Code: ~80
Integrity check home scores to be integer values	Lines of Code: ~10	Lines of Code: ~50
Integrity check date values to be yyyy-mm-dd format	Lines of Code: ~20	Lines of Code: ~60
Integrity check home rankings to be positive	Lines of Code: ~5	Lines of Code: ~45
Merge home/away ranking to derive rank difference	Lines of Code: ~10	Lines of Code: ~85
Statistics to find most frequent score to fill missing values	Lines of Code: ~25	Lines of Code: ~60

Figure 5.4 Comparison of Soccer Matches Scenario

5.2 Historical Weather Database

Suppose you are a meteorologist from CERN who has come to Urbana to visit relatives. You have some free time and decide to

investigate meteorological trends in Urbana to see if there is any evidence of climate change in the region. The National Climatic Data Center stores historical weather measurements for Urbana over many decades. From here, you obtain data with which you wish to compare temperature trends for a particular month over the years. Figures 5.5-5.8 show the data tables as they originally appeared.

Month	Day	Location	High Temp	Low Temp
10	1	61801	64.4	39.2
10	2	61801	64.4	35.6
10	3	61801	73.4	42.8
10	4	61801	69.1	42.1
10	5	61801	80.1	42.1
10	6	61801	82.4	48.2
10	7	61801	68	57
10	8	61801	69.8	53.6
10	9	61801	75	43
10	10	61801	79	45

Figure 5.5 October 2008 Table (First 10 Records)

Month	Day	Location	High Temp	Low Temp	Windspeed	Precipitation
10	1	61801	60.1	42.1	4.9	0.54
10	2	61801	60.1	46	9.9	0.38
10	3	61801	55.4	48.2	9.7	0
10	4	61801	63	39	5.2	0
10	5	61801	68	39	2.2	0
10	6	61801	66.9	45	7.1	0.11
10	7	61801	63	39.9	8.8	0.19
10	8	61801	53.6	44.6	4.7	0
10	9	61801	52	46	4.8	2.15
10	10	61801	55.9	32	4.5	0

Figure 5.6 October 2009 Table (First 10 Records)

Month	Day	Years	Location	High Temp Change	Low Temp Change
10	4	2005-2006	61801	1.66	6.23
10	4	2006-2007	61801	2.77	1.12
10	4	2007-2008	61801	-0.9	-0.34
10	5	2005-2006	61801	2.9	-2.89
10	5	2006-2007	61801	-3.33	-3
10	5	2007-2008	61801	5.37	3.76
10	6	2005-2006	61801	-3.09	4.25
10	6	2006-2007	61801	-1.4	1.43
10	6	2007-2008	61801	-8.8	-5.5
10	7	2005-2006	61801	-5.11	-2
10	7	2006-2007	61801	3.48	-1.46
10	7	2007-2008	61801	6.83	4.09
10	8	2005-2006	61801	6.34	5.51
10	8	2006-2007	61801	-0.11	2.77
10	8	2007-2008	61801	-2.56	-1.24

Figure 5.7 Historical Temperature Change (15 Records)

Month	Day	High Temp Change	Low Temp Change
10	1	-4.3	2.9
10	2	-4.3	10.4
10	3	-180	5.4
10	4	-6.1	-3.1
10	5	-12.1	-3.1
10	6	-15.5	
10	7	-5	-17.1
10	8	-16.2	
10	9	-23	3
10	10	-23.1111	-13

Figure 5.8 Original October 08-09 Table (First 10 Records)

The data contains multiple types of errors, highlighted in red in Figure 5.8. To fix the tables, you decide to clean the data under the following requirements corresponding to the listed error type:

- Copy the location attribute over to the new table, which does not have a location column (Lexical error).
- Correct errors in high temperature difference by deriving from the high temperatures of the month between years (Contradiction).
- Round the high temperature change values with extra digits to one decimal place (Inconsistency).
- Fill missing temperature change values, minimizing overall change to statistical measures by using the average value over the same day in years past (Missing value).

Figure 5.9 shows the sequence of actions taken to clean the data. The four categories of operations are used with multiple parameters in order to enact the corrections necessary. In the case of missing values, the output of the STAT operations is used as inputs to fill empty values during the next round of cleaning. The result of cleaning is shown in Figure 5.10.

```
// Copy the location attribute
MAP oct08.location TO october0809.location SOURCE oct08
  DEST october0809 PARAM copy;
// Rederive temperature change from existing data
MERGE oct09.highTemp, oct08.highTemp
  TO october0809.highChange SOURCE oct08, oct09
  DEST october0809 PARAM sub;
// Round values to specified precision
INTEG highChange SOURCE october0809 PARAM round 1;
// Infer average value based on attribute dependencies
STAT lowChange SOURCE october0809 DEST annualDiff
  DEPEND month, day PARAM avg;
// Use inferred values to fill missing values
INTEG lowChange SOURCE october0809 PARAM fillEmpty;
```

Figure 5.9 Data Cleaning Sequence of Actions

Month	Day	Location	High Temp Change	Low Temp Change
10	1	61801	-4.3	2.9
10	2	61801	-4.3	10.4
10	3	61801	-18	5.4
10	4	61801	-6.1	-3.1
10	5	61801	-12.1	-3.1
10	6	61801	-15.5	0.1
10	7	61801	-5	-17.1
10	8	61801	-16.2	2.3
10	9	61801	-23	3
10	10	61801	-23.1	-13

Figure 5.10 Cleaned October 08-09 Table (First 10 Records)

Figure 5.10 shows that the Data Cleaning Framework was successful in correcting all error types through the series of data cleaning operations. Each operation cleaned one type of error, and although no single operation cleaned all types of errors, the combination of all operations met the cleaning requirements. The matrix in Figure 5.11 shows the difference in programming each of the operations necessary to clean the data. Utilizing the functionality provided by the Data Cleaning Framework allows users of the system to minimize the number of lines of code needed. Without this support, implementing the same operations in the baseline system takes several times the amount of code. Example comparison code from the STAT operation is shown in Appendix B.

Operation	Data Cleaning Framework	Baseline (No Framework)
Map to copy location attribute to another table	Lines of Code: ~ 5	Lines of Code: ~75
Integrity check high temperatures to round	Lines of Code: ~ 15	Lines of Code: ~50

values to two decimals		
Merge low temperatures from separate tables to derive temperature change	Lines of Code: ~ 10	Lines of Code: ~85
Statistics to find average low temperature change to fill missing values	Lines of Code: ~ 15	Lines of Code: ~50

Figure 5.11 Comparison of Historical Weather Scenario

5.3 Analysis

The Data Cleaning Framework is built on the principle that there is no set of cleaning functions suitable for any arbitrary data. Therefore, it provides support for user-specified functions by allowing the user to utilize abstractions supplied by the framework and focus solely on the core functionality for their transformation. This not only cuts down the amount of code necessary to implement a new cleaning function, but also makes the process much simpler, as the number of concerns for the user is reduced. We see from the results that this extensibility allows users to implement functions capable of cleaning any of the mentioned error types.

In comparison, the baseline requires that users first implement the functionality for fetching the relevant data to be modified. Next, the user must get information about the schema of the retrieved data themselves. They then must use this information to set up the iteration through the records. It is only after this point that the core functionality can be created, and users specify the cleaning transformation. In the code for the functionality, users must manually implement the necessary methods to actually modify the data and then save the changes to the database.

It is important to note that the Data Cleaning Framework provides additional capabilities beyond simply reducing the amount of programming necessary. This includes a graphical user interface that can be used to view tables and construct a sequence of data cleaning commands to run. The user interface can also give a preview of changes made to table contents, letting users to see in advance what effects their cleaning actions will have on the database. This helps to prevent new errors from being introduced when unintended cleaning operations are run.

From the experimental results, we conclude that users can utilize the Data Cleaning Framework's support for the four categories of cleaning operations to implement functions to clean most common error types. While users pay an upfront cost of implementing their cleaning function using the functionality provided by Data Cleaning Framework, this cost is small compared to the amount of work necessary to implement a cleaning function from scratch. The extensibility of the Data Cleaning Framework proves to be an integral asset when performing data cleaning.

Chapter 6

Related Work

The concept of data cleaning is an area of much continuing research. Many methods have been developed to tackle the problem of identifying and cleaning dirty data. We briefly describe the following systems for performing data cleaning.

ARKTOS is a framework used for modeling and executing the Extraction-Transformation-Load process in data warehouse creation. Data cleaning is a key part of the ETL process, consisting of single steps that extract relevant data from the sources, transform it to the target format, clean it, and then load it into the data warehouse. These steps are cleaning operations called activities. Each activity is linked to input and output relations, where the functionality of an activity is described by an SQL statement. Each statement is associated with a particular error type and a policy which determines what actions to take when an error is found [10].

IntelliClean is a rule based approach to data cleaning that monitors the database without direct user execution. IntelliClean uses four types of cleaning rules to specify cleaning actions to be taken after certain conditions are met. Duplicate identification rules specify how tuples are classified as duplicates, and merge/purge rules specify how these duplicates will be handled. Update rules define how data is to be modified

to satisfy a particular constraint, and alert rules specify trigger conditions that cause the user to be notified [11].

The Data Cleaning Framework focuses on the concept of user extensibility, but it is just one of the myriad approaches towards a solution, in a field where there is no absolute answer. The ARKTOS and IntelliClean systems demonstrate alternative ways of dealing with the problem of data cleaning.

Chapter 7

Conclusion

The huge quantities of data created and stored in the information era inevitably leads to the introduction of data errors and anomalies. With the speed with which information is transmitted, never before has accurate data been so critical for business. Cleaning this “dirty” data has proved to be a major problem in industry, with no consensus on the exact nature of the solution. This thesis proposes an extensible system for data cleaning, the Data Cleaning Framework.

We observed that due to the broad and diverse nature of the data that needs to be cleaned, there exists no comprehensive set of operations suitable for any arbitrary domain. The Data Cleaning Framework was created with extensibility foremost as a result. By grouping data cleaning operations into types, we are able to provide a set of tools and support for users in implementing new cleaning functions. This allows users to focus on creating the core functionality while minimizing the amount of time and programming spent. Evaluating the Data Cleaning Framework on real-world data demonstrated the flexibility of this user-customizable approach.

There is no one-size-fits-all approach to data cleaning. Therefore, we believe that the future of data cleaning will be dominated by extensible systems. These systems will allow users to impose their requirements on

how data should be cleaned, rather than working around the limitations of the system. The Data Cleaning Framework is a system which attempts to accomplish this, and we have established that extensibility for data cleaning is an approach worth pursuing.

References

- [1] T. Redman. The Impact of Poor Data Quality on the Typical Enterprise. In *Communications of the ACM*, Vol. 41, No. 2, 1998, 79-82.
- [2] H. Muller, J. Freytag. Problems, Methods, and Challenges in Comprehensive Data Cleansing. In *Technical Report HUB-IB-164*, Humboldt University, Berlin, 2003.
- [3] E. Rahm, H.H. Do. Data Cleaning: Problems and current approaches. In *IEEE Bulletin of the Technical Committee on Data Engineering*, 2000, 24, 4.
- [4] W.L. Low, M.L. Lee, T.W. Ling. A knowledge-based approach for duplicate elimination in data cleaning. In *Information Systems*, Vol. 26, 2001, 585-606.
- [5] J.I. Maletic, A. Marcus. Data Cleansing: Beyond Integrity Analysis. In *Proceedings of the Conference on Information Quality*, October 2000.
- [6] P. Oliveira, F. Rodrigues, P. Henriques, H. Galhardas. A Taxonomy of Data Quality Problems. In *Proc. of 2nd Int. Workshop on Data and Information Quality (DIQ)*, Porto, Portugal, 2005.
- [7] H. Galhardas, D. Florescu, D. Shasha, E. Simon, C. Saita. Declarative data cleaning: Language, model and algorithms. In *Proc. of 27th Int. Conf. on Very Large Databases (VLDB)*, 371-381. Rome, Italy, 2001.
- [8] M.L. Lee, H. Lu, T.W. Ling, Y.T. Ko. Cleansing Data for Mining and Warehousing. In *Proc. of 10th Int. Conf. on Database and Expert Systems Applications (DEXA)*, 751-760. 1999.

- [9] I. Ahmed, A. Aziz. Dynamic Approach for Data Scrubbing Process. In *International Journal on Computer Science and Engineering*, Vol. 02, No. 02, 2010, 416-423.
- [10] P. Vassiliadis, Z. Vagena, S. Skiadopoulos, N. Karayannidis, T. Sellis. ARKTOS: towards the modeling, design, control and execution of ETL processes. In *Information Systems*, Vol. 26, 2001, 537-561.
- [11] M.L. Lee, T.W. Ling, W.L. Low. IntelliClean: A knowledge-based intelligent data cleaner. In *Proc. of 6th Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 290–294, 2000.

Appendix A

The example code in this section only includes the key functionality of each operation, in order to provide an estimate for the amount of programming needed for a user-implemented function. The amount of code needed to implement a function varies by user, so estimates may be rough in some cases.

Example MAP Function (Using Data Cleaning Framework)

```
1     private void split(ResultSet rs, int currRow, String[] modAttrs,
2         String table, String[] paramCons, Object prevResults) throws
3         SQLException {
4         String value = rs.getString(1);
5         String[] split = value.split(paramCons[0]);
6         update(split, currRow, modAttrs, table);
7     }
```

Example MAP Function (Without Data Cleaning Framework)

```
1     // User-defined transformation function to split attributes
2     protected void split(ResultSet rs, String[] modAttrs, String table,
3         String[] paramCons, Object prevResults) {
4     try {
5         ResultSetMetaData rsmd = rs.getMetaData();
6         // Get information on new columns to add
7         numCols = rsmd.getColumnCount();
8         colType = rsmd.getColumnTypeName(1);
9         schemaSize = rsmd.getColumnDisplaySize(1);
10        Statement st = null;
```

```

8      // Allow queries to modify database contents and structure
      st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                              ResultSet.CONCUR_UPDATABLE);
9      // Add column of specified name, data type, size
      for (int i = 0; i < modAttrs.length; i++) {
10         String attrTable = table;
            // Check if attribute belongs in table
11         String[] tb = modAttrs[i].split("\\.");
12         if (tb.length == 2) {
13             attrTable = tb[0];
14         }
15         if (attrTable.equalsIgnoreCase(table)) {
            // Add new columns to table
16             String exe = "ALTER IGNORE TABLE " + table + "
                ADD " + modAttrs[i] + " " + colType + "(" + schemaSize + ")";
            st.executeUpdate(exe);
17         }
18     }
19 } catch (Exception e) {
20     System.err.println("Exception: " + e.getMessage());
21 }
22 try {
23     while (rs.next()) {
24         String value = rs.getString(1);
25         int currRow;
26         currRow = rs.getRow();
27         String[] split = value.split(paramCons[0]);
28         // Update each output column with new data value
29         for (int i = 0; i < modAttrs.length; i++) {
30             try {
31                 Statement st = null;
32                 ResultSet updateResult = null;
33                 st = con.createStatement(
                ResultSet.TYPE_SCROLL_SENSITIVE, ResultSet.CONCUR_UPDATABLE);
                // Use primary keys to enable modification of data
34                 String exe = primaryKeys + ", " + modAttrs + " from " + table + ";";
35                 updateResult = st.executeQuery(exe);
                // Set position index to row being modified
36                 updateResult.absolute(currRow);
                // Make modification to data, propagate changes to database
37                 updateResult.updateString(modAttrs[i], split[i]);
38                 updateResult.updateRow();
39             } catch (Exception e) {
40                 System.err.println("Exception: " +
                    e.getMessage());
41             }
42         }
43     }
44 } catch (SQLException e) {
45

```

```

46         e.printStackTrace();
47     }
48 }
49 protected void mapOperation(String[] origAttrs, String[] modAttrs,
    String[] fromTables, String[] toTables, String[] params)
    throws SQLException {
    // Check if more than one table is needed
50 if (toTables == null) {
51     for (int j = 0; j < fromTables.length; j++) {
52         // Create query to select input data
53         String query = formatQuery(origAttrs, fromTables[j]);
54         // Get primary keys of table to allow table modification
55         getTableKeys(fromTables[j]);
56         this.mapInputs (query, origAttrs[0], modAttrs,
57             fromTables[j], null, params);
58     }
59 } else {
60     String query = formatQuery(origAttrs, fromTables);
61     for (int j = 0; j < toTables.length; j++) {
62         getTableKeys(toTables[j]);
63         this.mapInputs (query, origAttrs[0], modAttrs,
64             fromTables[0], toTables[j], params);
65     }
66 }
67 // Fetches necessary input data from columns specified in command
68 protected ResultSet mapInputs(String query, String origAttr,
69     String[] modAttrs, String fromTableName, String toTableName,
70     String[] params) {
71 try {
72     // Execute query to fetch data
73     st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
74         ResultSet.CONCUR_UPDATABLE);
75     rs = st.executeQuery(query);
76
77     String[] paramCons = new String[params.length - 1];
78     for (int i = 1; i < params.length; i++) {
79         paramCons[i - 1] = params[i];
80     }
81     // Apply parameters to cleaning operation
82     String paramType = params[0].toUpperCase();
83     applyConditions(rs, origAttr, modAttrs, fromTableName,
84         toTableName, paramType, paramCons);
85 } catch (Exception e) {
86     System.err.println("Exception: " + e.getMessage());
87 }
88 // Pass fetched input data to user function
89 return rs;
90 }

```

Appendix B

Example STAT Function (Using Data Cleaning Framework)

```
1     protected void average(ResultSet rs, String key, String[] paramCons) {
2     double total = 0;
3     double numRows = 0;
4     while (rs.next()) {
5         double value = rs.getDouble(1);
6         total += value;
7         numRows++;
8     }
9     double avg;
10    if (numRows != 0) {
11        avg = (total / numRows);
12        // Map attribute to computed value
13        updateMapping(key, avg);
14    }
```

Example STAT Function (Without Data Cleaning Framework)

```
1     // User-defined function to infer value from average
2     protected void average(ResultSet rs, String key, String[] paramCons) {
3     double total = 0;
4     double numRows = 0;
5     while (rs.next()) {
6         double value = rs.getDouble(1);
7         total += value;
8         numRows++;
9     }
10    double avg;
11    if (numRows != 0) {
12        avg = (total / numRows);
13    }
```

```

12         // Map attribute to computed value
13         updateMapping(key, avg);
14     }
15     // Iterates through records to get subsets matching dependencies
16     public void statOn(String[] origAttrs, String[] modAttrs,
17         String[] fromTables, String[] toTables, String[] depends,
18         String[] params) throws SQLException {
19         // Create query to select entire set of relevant data
20         String query = formatQuery(origAttrs, fromTables, depends);
21         ResultSet all = this.getResultSet(query);
22         ResultSetMetaData allmd = all.getMetaData();
23         int numCols = allmd.getColumnCount();
24         String[][] values = new String[depends.length][numCols - 1];
25         // Create mapping of attributes to inferred value
26         Global.statMapping = new Hashtable<String, String>();
27         while (all.next()) {
28             for (int i = 0; i < values[0].length; i++) {
29                 values[i][i] = all.getString(depends[i]);
30             }
31             // Create query to filter results on matching dependencies
32             String subquery = buildFilter(origAttrs, fromTables, dest,
33                 depends, values);
34             String key = getKeys(depends, values, "");
35             this.executeOperation(subquery, key, "", null, null, params);
36         }
37     }
38     // Fetches necessary input data from columns specified in command
39     protected ResultSet executeOperation(String query, String key, String value,
40         String origAttr, String fromTableName, String[] params) {
41         try {
42             // Allow queries to modify database contents and structure
43             st = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
44                 ResultSet.CONCUR_UPDATABLE);
45             rs = st.executeQuery(query);
46             if (params != null) {
47                 String[] paramCons = new String[params.length - 1];
48                 for (int i = 1; i < params.length; i++) {
49                     paramCons[i - 1] = params[i];
50                 }
51                 String paramType = params[0].toUpperCase();
52                 applyConditions(rs, key, value, origAttr, fromTableName,
53                     paramType, paramCons);
54             }
55         } catch (Exception e) {
56             System.err.println("Exception: " + e.getMessage());
57         }
58     }
59     // Pass fetched input data to user function
60     return rs;
61 }

```