INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Support for User Interaction in a Data Cleaning Process

João Lobato dos Santos Dias

Dissertation submitted to obtain the Master Degree in

Information Systems and Computer Engineering

Jury

President:	Prof. Dr. Ernesto José Marques Morgado
Supervisor:	Prof. Dr. Helena Isabel de Jesus Galhardas
Co-Supervisor:	Prof. Dr. Maria Antónia Bacelar da Costa Lopes
Member:	Prof. Dr. Francisco José Moreira Couto

October 2012

Abstract

Data transformations to clean dirty datasets are difficult to devise and the tuning of these transformations is complicated by the lack of support for tools that detect data quality problems during intermediate stages of the data cleaning process. Moreover, fully automated cleaning solutions are many times not attainable and users have to be involved in the repair of some data quality problems. As proposed in [Galhardas et al., 2011], in order to better support the user involvement in data cleaning processes, defined in terms of graphs of data transformations, can be equipped with data quality constraints that help users identifying the points of the transformation and the records that need their attention and manual data repairs for representing the way users can provide the feedback required to manually clean some data items. In this thesis, we propose a realization of these concepts in the context of AJAX – an existing data cleaning tool that works with data cleaning graphs – and report on the implementation of CLEENEX, a prototypical im- plementation of the proposed solution. CLEENEX provides proof-of-concept for the integration of data quality constraints and manual data repairs in data cleaning programs, demonstrating the feasibility of the approach to user involvement in data cleaning proposed in [Galhardas et al., 2011]. Moreover, CLEENEX is instrumental for performing a thorough evaluation of this approach.

Keywords: data cleaning , data cleaning graphs , user involvement

Resumo

Transformações de dados que permitam limpar dados sujos são difíceis de criar, assim como o afinar destas transformações é complicado pela falta de ferramentas que suportem a deteção de problemas de qualidade de dados durante fases intermédias do processo de limpeza de dados. Para além disso, muitas vezes não se conseguem obter soluções de limpeza completamente automáticas tornando-se necessário envolver os utilizadores para que estes resolvam alguns problemas. Tal como proposto em [Galhardas et al., 2011], para melhor suportar o envolvimento do utilizador em processos de limpeza de dados, a limpeza definida à custa de grafos de transformações pode ser aumentada com restrições de qualidade de dados que ajudam os utilizadores a identificar pontos nas transformações e registos que requerem a sua atenção, assim como reparações manuais de dados para representar a forma como os utilizadores podem fornecer a informação necessária para a limpeza manual de alguns dados. Nesta tese, propõe-se a concretização destes conceitos no contexto do AJAX – uma ferramenta de limpeza de dados preexistente que funciona com grafos de limpeza de dados – e a descrição da implementação do CLEENEX, um protótipo que implementa a solução proposta. O CLEENEX providencia uma prova-de-conceito para a integração de restrições de qualidade de dados e reparações manuais de dados em programas de limpeza de dados, demonstrando a viabilidade da aproximação proposta em [Galhardas et al., 2011] para o envolvimento de utilizadores em limpeza de dados. Para além disso, o CLEENEX é instrumental na avaliação desta aproximação.

Palavras-Chave: limpeza de dados, grafos de limpeza de dados, envolvimento do utilizador

Acknowledgements

First and foremost I'd like to thank my parents for all their love and support. And yes, I know that I don't show my feelings as well and as often as I should. You've made me who I am today, nature and nurture. No matter how high I climb, it is only because you taught me how to walk.

I would also like to thank Prof. Helena and Prof. Antónia, my thesis supervisors. Thanks for all your help during this past year and for making this section of this document the hardest one to write.

To Ana, Carolina, João, Pedro and Sérgio, in no particular order other than alphabetical, I will always remember how you stood (and sat) by me through hardship. I won't forget good times and good laughs either, but bad moments leave scars. You went above and beyond what I could possibly ask of you. I've incurred debt I can never hope to repay, but I won't use it as an excuse not to try.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Organizations typically require large quantities of information to operate. These large datasets are deeply integrated in the core business of the organization and are used in the decision-making process. The accuracy of any decision is highly dependent on the quality of the data upon which it was based. Data with inconsistencies, errors, conflicts, redundancies, that is incomplete or outdated is called *dirty data*. Data can become dirty due to several reasons, even if we consider a single database. Misspellings can occur during data entry. Records can become incomplete if some information was unavailable when the data was inserted in the system. Data can become outdated if details of the real-world entity to which it refers change over time, *e.g.* a person moves to a different address. If we consider integration of data from multiple sources, then we can face other problems like different data models. There are several taxonomies in the literature that can be used to classify the types of data quality problems that can be found in relational databases [1, 11, 13, 14, 16].

The process that can be used to identify and repair the data quality problems of a dataset is called *data cleaning*. One can follow several methods when cleaning data. The simplest approach to data cleaning is to perform any necessary repairs manually. However, this approach is impractical when we consider large datasets, which is usually the norm. The other option is to write programs to automatically clean the data.

Typically, a data cleaning program is modeled as a graph of data transformations. Each of these data transformations encompasses a logical task in the process of removing quality problems from the data. These transformations are composed, feeding the output of one transformation to next. This results in a directed acyclic graph of transformations, where input dirty data is processed in turn, until the program outputs cleaned data. In practice, it is impossible to write a data cleaning program that outputs completely clean data in a single pass. Because it is not easy to discover all quality problems without effectively cleaning the data manually, the program will have to be run several times. Between each of these iterations, is is up to users to evaluate the output of the program and refine the data transformations

in order to bring the data closer to a clean state. The process of data cleaning is characterized by a *Debug-Refine-Clean* loop, that continues until the data is considered clean enough for the purposes of the data consumers. Note that, in most cases, it is necessary to perform some form of manual cleaning of the data because, as the number of iterations increases, it is easier to simply clean the data manually than refining the data transformations again.

In order to develop data cleaning programs, one can use Extraction-Transform-Load (ETL) tools (e.g., SQL Server Integration Services[1] or Pentaho[2]) or specialized data cleaning tools (e.g., those provided by Informatica[3]) that already include primitives for typical data cleaning tasks, like record normalization or identification and eliminations of duplicates. However, these tools usually treat the execution of data cleaning programs as a black box process. This means that the only feedback that they provide to the user is through the final results of the whole automated cleaning process. In case of failure, because the user only has the final data available, it can be hard to identify which data transformation did not accomplish its job.

In line with what was proposed in [9], we believe that a data cleaning tool should include provisions for the inclusion of human feedback during the data cleaning process. The designer should be able to specify points in the data cleaning graph where intermediate data is materialized and subject to certain criteria to evaluate its quality. By directing the user attention to certain stages where errors are more likely to be revealed, it is possible to involve the user in the data cleaning process without subjecting her to exceeding amounts of data. Furthermore, if the system accepts direct user feedback over intermediate data on those stages, it is possible to directly incorporate her knowledge in that intermediate data, instead of forcing her to refine transformations to deal with data quality problems that may occur sparingly in the input data. If the interface for user interaction is tuned to directly support the operations that a user is expected to perform on the intermediate data, her effort will not be exceedingly high.

1.1 Objectives

The main goal of our work is to propose a realization of the concept of data cleaning programs as proposed in [9]. In concrete, we consider that user involvement in the cleaning process requires the addition of two new types of constructs to data cleaning graphs: *data quality constraints*, used to draw the attention of the users to specific points in the graph that require their attention, and *manual data repairs*, that represent the actions that users can undertake to manually clean some data items. These constructs have been proposed in the context of a data cleaning framework where data cleaning programs are defined in a declarative language inspired in SQL, AJAX [7, 8].

[1] http://www.microsoft.com/sqlserver/en/us/solutions-technologies/business-intelligence/integration-services.aspx
[2] http://kettle.pentaho.com/
[3] http://www.informatica.com/

The realization proposed in this thesis, which we designate by CLEENEX, covers both the design-time and the run-time aspects of the approach to user involvement in data cleaning proposed in [9]. On one hand, support for the definition of *data quality constraints* and *manual data repairs* in data cleaning graphs requires the extension of the AJAX specification language and the partial redefinition of its operational semantics. On the other hand, it is also necessary to design the support for the user feedback during the data cleaning process, centered on the inspection of tuples that violate data quality constraints and incorporation of user feedback provided through manual data repairs.

The other goals of the work are the development of a prototypical implementation of the proposed solution, through the modification of the AJAX prototype, and the realization of some evaluation experiments that demonstrate the feasibility of the approach to user involvement in data cleaning proposed in [9] and provide some additional results for the evaluation of this approach.

1.2 Contributions

The main contributions of this thesis are:

- Definition of a language for specifying data quality constraints in a data cleaning program.
- Definition of a language for specifying manual data repairs in a data cleaning program.
- Partial redesign of the AJAX framework, to leverage the new constructs during the execution of data cleaning graphs.
- Partial redesign of the AJAX user interface, to facilitate the visualization of tuples that violate data quality constraints and the provision of user feedback through manual data repairs.
- Development of a prototype implementing the proposed framework, based on the original AJAX prototype.
- Evaluation of the impact of user feedback in a data cleaning process, measuring the accuracy gains while considering the increase in user effort.
- Evaluation of the impact of the new constructs in the runtime execution of data cleaning programs.

1.3 Structure of This Document

The remainder of the thesis is organized as follows: Chapter 2 presents some background of our work, providing an overview of AJAX and detailing the approach to user involvement in data cleaning in which our work relies. Chapter 3 presents CLEENEX and its prototype, the main contributions of this work.

Chapter 4 presents the experimental validation conducted to evaluate the CLEENEX prototype. Chapter 5 discusses the most relevant related work. Chapter 6 draws some conclusions about the work presented both in terms of results and contributions, and discusses some limitations and directions for future work.

Chapter 2

Background

In this chapter we present the background of the work developed in this thesis. First, we describe in Section 2.1 the AJAX data cleaning framework that uses a declarative language to define data cleaning programs. Then, in Section 2.2, we discuss some shortcomings of AJAX and present two constructs that were proposed to leverage user knowledge in data cleaning.

2.1 AJAX Framework

The AJAX framework [7, 8] proposes the definition of data cleaning programs using a declarative language inspired in SQL. A program is modeled as a Directed Acyclic Graph (DAG) of data transformations and relations. The usage of a declarative language to define the cleaning program permits a clear separation between the logical layer – which data transformations should be applied to the initial dataset – and the physical layer – how should those transformations be executed. This division is inspired by Relational Database Management Systems (RDBMS).

In order to understand why the separation between logical and physical layers is important, consider two different databases: one of them stores customer data for a retail company, while the other stores bibliographic information of scientific publications. The existence of duplicate records in a database is very common so virtually any data cleaning program must perform detection and merging of duplicates. This operation is both computationally heavy (as it usually means a Cartesian product between the whole set of records) and hampered by the possible existence of homonyms, *i.e.*, real world entities that share the same name. While a programmer tasked with cleaning the retail database would compare customer names using the Levenshtein Distance [12], she might choose to use a different similarity metric to compare two publications, *e.g.*, the sets of authors of each of them, using the Jaccard Index [10]. However, while the similarity metrics are different, it is possible to abstract the common mechanism

of applying such functions to pairs of records and clustering those records according to the results of those metrics. The separation of logic and physical layers allows the programmer to concentrate on choosing the appropriate similarity metric for the task at hand, rather than how to try to speed up the Cartesian product of a million tuples using the Levenshtein Distance metric.

In this section, we explain the AJAX framework, detail the declarative language used to define the data cleaning programs, the underlying exception mechanism and the component architecture of the prototype that implements the framework.

2.1.1 Cleaning Data with AJAX

Figure 2.1 shows a diagram that represents how the AJAX framework can be used to clean data. This framework assigns two roles to the human operator: the *Designer* and the *User*. The designer specifies the DAG that constitutes the cleaning program. The user runs the data cleaning process.

The designer uses the declarative language to define a data cleaning program. This language possesses five different operators corresponding to five different types of data transformation. She must also supply a configuration file and the implementation of the required external functions. The configuration file will not be discussed here. It is a simple property file with the required information to connect to the RDBMS, which program to execute, *etc*.

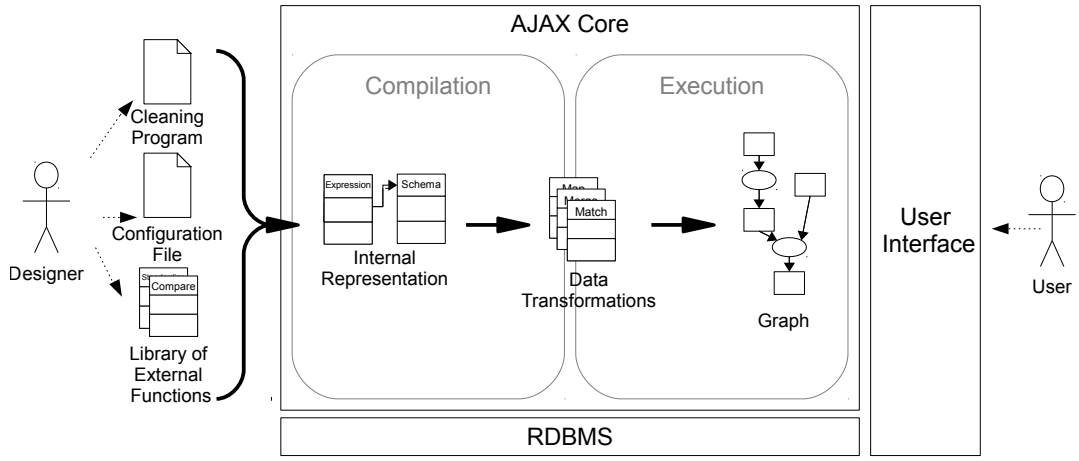


Figure 2.1: The AJAX framework.

The external functions are defined in the Java programming language as static methods, using classes as namespaces. These are functions that are invoked by the data transformations. Some functions are general utilities that are already distributed with AJAX, *e.g.*, type conversions or string splitting according to a Regular Expression. However, most of the required functions are domain-specific, like record comparison functions and data normalization routines. Due to the specificity of these operations and their coupling with the chosen transformations, the framework can be extended to support them. After being

implemented, they can be inserted in the library and become available to be reused in other cleaning programs.

Each transformation in the data cleaning program can have one or more input relations and one or more output relations. All intermediate data, *i.e.*, the output relation of a transformation and subsequent input of the next transformation, is materialized in tables stored in the RDBMS. Each data transformation can invoke any function defined in the external library provided.

The core operation of the AJAX framework is divided in two phases: a compilation phase and an execution phase. In the *Compilation Phase*, the cleaning program is parsed and the system constructs an internal representation of it. If the program is syntactically and semantically correct, the system generates a Java class implementing each data transformation, based on existing templates for each kind of operator in the declarative language. This Java class reads the tuples of each input relation, transforms them, and populates the output relations. These classes are then compiled and linked to the Java runtime.

After this compilation, the system enters the *Execution Phase*, where each transformation is executed according to the order defined in the DAG. The data cleaning graph is presented to the user through a Graphical User Interface (GUI). This interface is composed of two panes. The top pane shows the data cleaning graph and can be used to re-execute the program, either from start to end or only from an intermediate transformation to the end, and to select any transformation or relational table. The bottom pane is used to show program execution statistics, display the code of a transformation, display the contents of a selected table as well as permitting the user to insert, delete or edit any tuple in that table.

As such, using this interface, a user can inspect intermediate data and decide which refinements should be applied to the logic underlying the data transformations in the graph in order to improve the quality of the output data. The user can also edit the data manually if she determines that it is not practical to do so programmmatically. While the user visualizes the data through a spreadsheet interface, she may perform edits using a form interface. Since the concept of unit of user feedback is not in way realized in AJAX, the user intentions are only materialized in the modified data. If the contents of the underlying relation change, *e.g.*, because the transformation was re-evaluated, the feedback from the user is lost among any modifications.

2.1.2 Language and Operators

The programmer uses a declarative language based in SQL to specify the input relations, the signature of the required external functions, the composite data types used by these functions and the data cleaning graph of transformations.

The input relations are specified according to their schema. Initial data must be already loaded in the

```
Author
(
  firstname STRING,
  lastname STRING
);

Functions.extractNameComponents(STRING) RETURN Author
Functions.standardizeFirstName(STRING) RETURN STRING
Functions.standardizeLastName(STRING) RETURN STRING
Functions.generateAuthorId(INTEGER) RETURN INTEGER

CREATE MAPPING StandardAuthorsByPublication
FROM Authors A
LET author = extractNameComponents(A.name),
    fname = standardizeFirstName(author.firstname),
    lname = standardizeLastName(author.lastname),
    authorid = generateAuthorId(0)
{
  SELECT authorid,
         A.publicationId,
         A.fullname AS fullname,
         fname AS firstname,
         lname AS lastname
  KEY authorid
}
```

Listing 2.1: Example of a data transformation defined using the Map operator.

RDBMS. Similarly, the programmer must define the signature of any external function or composite data type used by at least one data transformation in the data cleaning graph. These artifacts must be loaded previously into the library in a predefined directory.

The specification of each transformation begins with a *create* clause that identifies the type of transformation, followed by a *from* clause that specifies the input relations. An optional *let* clause allows the assignment of local variables and the invocation of external functions to transform the input tuples. A *where* clause can be used to filter some of the input tuples and finally one or more *output* clauses specify the schema of each of the output relations as well as the integrity constraints imposed over each of them.

The framework supports five operators that can be specified with the language. Each of these operators corresponds to a type of data transformation with particular execution semantics. The *Map* establishes a one-to-many mapping between an input tuple and the corresponding output tuples; this operator extends SQL by allowing to generate more than one output tuple for each input tuple. The *Match* performs an approximate join and is implemented as a first-class operator to allow different physical algorithms. The *Cluster* groups the tuples of an input relation into sets of tuples using a given clustering algorithm. The *Merge* partitions an input relation according to a given criteria and chooses a representative for each partition. Finally, a *View* is a SQL query. This small palette of operators is considered sufficient to design an effective data cleaning process [8]. Note that each transformation should be tailored to the specific needs of the application using the functions defined in the external library.

Listing 2.1 provides an example of the usage of a Map operator. The purpose of the *StandardAuthors-ByPublication* transformation is to split the names of authors in first name and last name, standardize these subcomponents and assign them a unique identifier. This transformation receives as input the relation *Authors*, with schema $(publicationId, fullName)$. The function *extractNameComponents* splits the name into its subcomponents, returning an instance of the *Author* datatype. *standardizeFirstName* and *standardizeLastName* are external functions defined in Java that encapsulate the logic of the name normalization. *generateAuthorId* generates sequential integers that identify each author. We are imposing a primary key constraint on the output relation using the attribute *authorid*. In this case, this output relation has the same name of the transformation. This is usual when the transformation has a single output.

2.1.3 Exception Tuples

The execution of data transformations in the AJAX framework includes the generation of *exception tuples* to help the user identify problems encountered during the data cleaning process. A data transformation may be unable to process an input in two cases: (i) it is presented with data that it is not prepared to handle, which leads to one of the external functions invoked within it to signal an error, *e.g.*, a transformation that is supposed to choose a representative for a set of tuples in a cluster is unable to do so, or (ii) an integrity constraint imposed over one of the output relations of that transformation is violated. Either way, the input tuple in question is not processed any further by the data cleaning process and the corresponding output tuples are not inserted in the regular output relations. Instead, the corresponding exception tuple is generated, providing a description of the problem encountered and associating it with the identifier of the input tuple that caused the problem. To accommodate the exception tuples, the system creates an exception output relation for each input relation of a transformation. Note that integrity constraints are enforced by the underlying RDBMS.

Figure 2.2 provides an example of the generation of exception tuples. The input relation *ClusteredNames* stores clusters with different spellings of the same name. *LongestName* is a data transformation of type *Merge* that selects the longest name as the representative for each cluster. However, in the example, that criterion cannot be successfully applied to cluster 2 because the names *Alan Smithee* and *Alan Smithee* have the same length. In this case, the exception tuple $\langle 2,$ *Many different names with max length* $\rangle$ describing the problem is inserted into *ClusteredNames*^{exception}.

2.1.4 Component Architecture

Figure 2.3 shows the component architecture of the AJAX prototype. The *Parser* component is responsible to validate the syntax and interpret the data cleaning program supplied by the designer. This

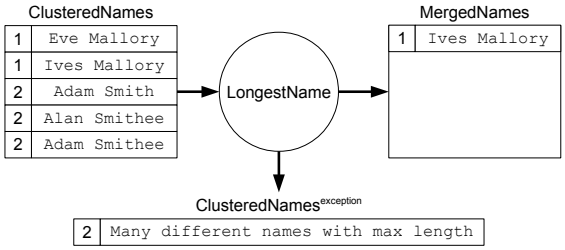


Figure 2.2: Example of the generation of an exception tuple.

component directs the *Catalog Manager* to construct an internal representation of the cleaning program.

The *Catalog Manager* constructs an internal representation of all the composite data types, functions and transformations. This component asks the *Database Manager* to create the necessary data structures to communicate with the RDBMS. The *Catalog Manager* also identifies any semantic problems of the cleaning program, *e.g.*, references to relations never created, and reports them back to the *Parser* to halt its execution. After all the program is successfully interpreted, the Catalog uses the *Optimizer* to select an appropriate implementation for each data transformation. Subsequently, it generates a Java class based on internal templates to execute the transformation logic. This code is then compiled and linked to the current Java Runtime.

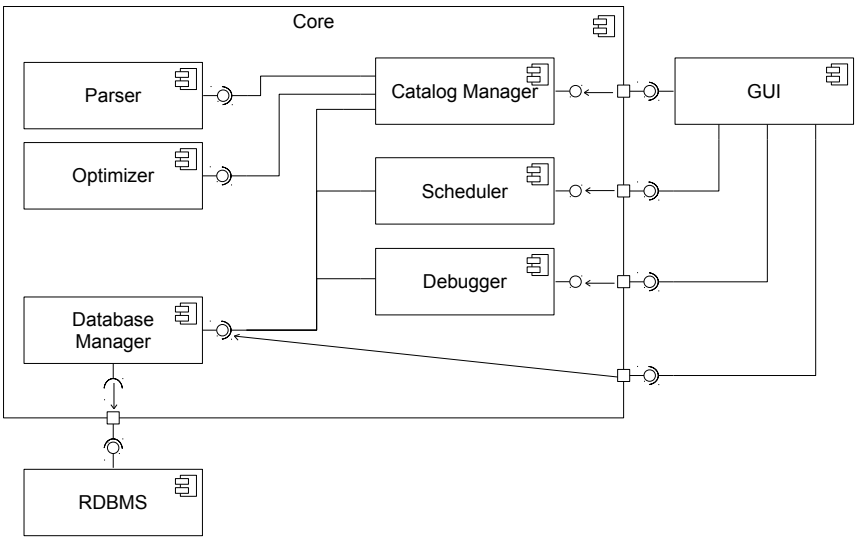


Figure 2.3: AJAX component architecture.

The *Optimizer* analyzes each data transformation and selects either execution in the Java Runtime or directly inside the RDBMS. It can be advantageous to let the RDBMS execute View operators, for example. On the other hand, if a Map generates more than one output tuple for each input tuple it must be executed purely in the Java because SQL queries do not support that operation.

The *Database Manager* component is responsible for all communication with the RDBMS. When requested by the Catalog during the compilation step, this component generates internal data structures to abstract differences between RDBMSs of different vendors, thus ensuring compatibility. During execution, it executes an SQL query, insert and delete commands as requested by the *Scheduler*, the *Debugger* or the *GUI*.

The *Scheduler* controls the execution phase of the data cleaning process. It executes compiled transformations according to the order defined in the data cleaning program, following commands from the *GUI* component.

The *GUI* component can be used to control the execution of the data cleaning program. This component contacts the *Catalog Manager* and constructs a graphical representation of the data cleaning graph. During execution, this component relays the user instructions to the Scheduler. It can also be used to analyze initial, intermediate and final data as well as support edition of data by the user, using the Database Manager. At the request of the user, the GUI uses the *Debugger* to trace the origin of a tuple, *i.e.*, identifies the input tuples of a transformation that originated a selected output tuple of that transformation.

2.2 Support for User Involvement in Data Cleaning

In Section 2.1, we detailed the two roles for the human operator that the AJAX framework supports. We also noted that both the designer and the user require skills in data cleaning and knowledge about the domain of the program. This stems from the fact that the user can visualize and alter absolutely any intermediate data in the process. In practice, this requirement that the users, *i.e.*, the data consumers, have intimate understanding of the data cleaning process is an important obstacle in the desire to leverage their knowledge about the domain. Even if there is a large group of people available to supply information to clean the data, only those who understand the cleaning program are capable of providing that information. If these two roles were more differentiated, it would open the possibility to leverage the knowledge of a wider range of users.

Another problem in the AJAX framework is the exception tuple mechanism. This mechanism can be used to identify some data quality problems not addressed automatically, in particular by defining integrity constraints over the data in the output relation of a data transformation. However, because these integrity constraints are those that the RDBMS enforces, it is possible that valuable information is lost. Consider a *Unique* constraint over one attribute of a relation. Because this constraint is enforced by the underlying RDBMS, the first tuple inserted with a certain value for that attribute is accepted and the subsequent tuples with the same value are rejected. Conceptually, all of those tuples are responsible for the violation of the constraint. Therefore, all of those tuples should be presented to the user in order to help her make

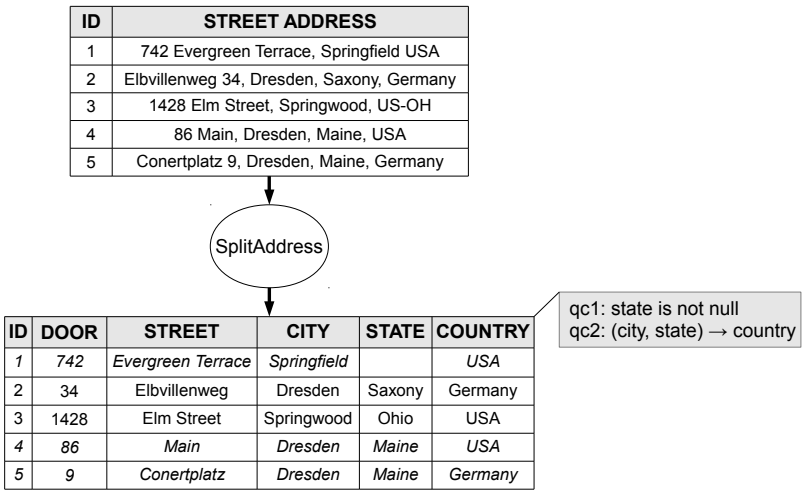
an informed decision about the right course of action.

[9] proposed two constructs to overcome the limitation of the integrity constraints and provide the means to limit the data that a User can view or alter in the data cleaning graph. It is argued that, by limiting the scope of the actions of a user and directing them to the points in the graph where they have more impact, her feedback is easier to provide, *i.e.*, requires less effort, and it is also easier to integrate. The proposed constructs are *Quality Constraints* and *Manual Data Repairs*. These constructs are added to the graph of data transformations by the designer and enforced during the execution of the data cleaning process. [9] also presents a sketch of the operational semantics to evaluate a data cleaning graph augmented with these constructs.

2.2.1 Quality Constraints

Quality Constraints (QCs) are added to a data cleaning process to pinpoint the stages in the graph where it is desired that intermediate data obey a certain criteria. These are prime locations for the inclusion of user feedback. Note that a tuple flagged due to violating these constraints is not removed from the output relation.

In order to define a QC it is necessary to identify the point of the graph where it should be enforced, *i.e.* any relation in the data cleaning graph, and the condition that the tuples of that relation should obey. There are not strict restrictions on the type of condition – it can either target each tuple of the relation or the relation as whole. A QC can be defined over the data of the relation where it should be enforced, but it can also reference other relations, as long as they appear upstream in the graph.



ID	STREET ADDRESS
1	742 Evergreen Terrace, Springfield USA
2	Elbvillenweg 34, Dresden, Saxony, Germany
3	1428 Elm Street, Springwood, US-OH
4	86 Main, Dresden, Maine, USA
5	Conertplatz 9, Dresden, Maine, Germany

ID	DOOR	STREET	CITY	STATE	COUNTRY
1	*742*	*Evergreen Terrace*	*Springfield*		*USA*
2	34	Elbvillenweg	Dresden	Saxony	Germany
3	1428	Elm Street	Springwood	Ohio	USA
4	*86*	*Main*	*Dresden*	*Maine*	*USA*
5	*9*	*Conertplatz*	*Dresden*	*Maine*	*Germany*

Figure 2.4: Example of the definition of QCs.

Figure 2.4 presents a scenario demonstrating the usage of QCs. The *SplitAddress* data transformation divides street addresses in their constituent parts. However, the data transformation cannot determine

if the information is wrong or complete. In order to detect these data quality problems, two QCs have been defined over the output tuple of the transformation: *qc1* that marks a tuple if it does not contain a value in the *state* attribute – in this case, tuple 1 – and *qc2*, that marks all tuples that do not respect the functional dependency that the country in the address should be determined by its city and state. In this case, both tuples 4 and 5 are marked for violating that constraint.

2.2.2 Manual Data Repairs

Manual Data Repairs (MDRs) are the mechanism proposed in [9] to limit and guide the actions that a user can perform during the data cleaning process. Like the QCs described in Section 2.2.1, MDRs are added by the designer to the cleaning graph. They can be defined over the output relation of a transformation in the graph.

A MDR defined over a relation is composed of a View and an Action. The *View* is an updatable view over that relation. The purpose of the View is to limit the amount of information that the user must analyze to determine the changes that she wants to perform on the data in order to clean it. This is achieved by projecting only some of the attributes of the relation or selecting only some of the tuples of the relation. In particular, it is possible to limit the tuples shown to those that violate a certain QC.

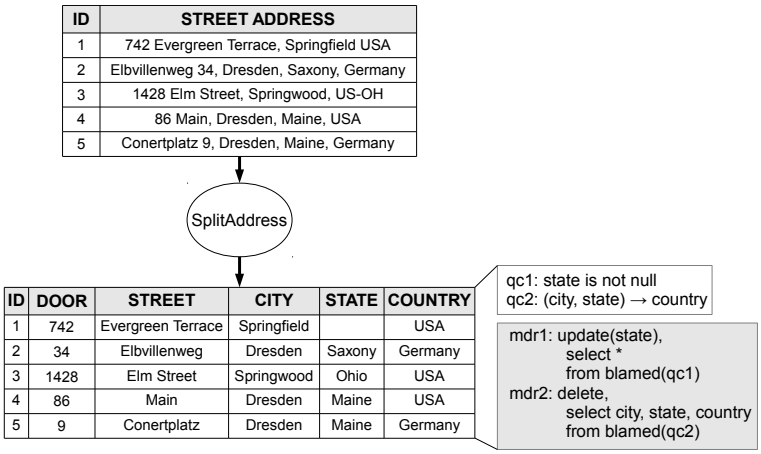


Figure 2.5: Example of the definition of MDRs.

The *Action* is either an insert, delete or update operation that can be applied to the updatable view. By constraining the choices that the user faces, it becomes easier to provide feedback. In addition, MDRs are template actions that can be saved and re-applied in subsequent runs of the data cleaning process. They can even form the groundwork to combine multiple units of feedback provided by different users [19].

Consider Figure 2.5. This figure shows the same example transformation from Figure 2.4 this time with the definition of two MDRs. Each View is defined using SQL. The notation $blamed(QC_i)$ denotes the

tuples in the target relation that violate the given QC. *mdr1* allows the user to add the missing information to tuple 1. *mdr2*, on the other hand, lets the user delete any tuple that violates *qc2*. Note that the view associated to *mdr2* only projects the *city*, *state* and *country* attributes. The values of these attributes are sufficient to help the user resolve the conflict that led to the violation of the constraint. After the user selects the appropriate tuples and the actions that should be performed on them, the provided feedback can be incorporated in data cleaning process, by applying the chosen operations to the target data.

2.2.3 Operational Semantics

In this section we present the operational semantics for a data cleaning graph enriched with QCs and MDRs that were proposed in [9]. However, we must first present the formal definition of a set of *blamed tuples* of a QC.

Definition 1. Let Q be a QC defined over a relation R and t the set of tuples of R. If $t \not\models Q$, the set $blamed(Q)$ is defined as the union of all subsets s of t such that $t \setminus s \models Q$ and $\nexists s' \subset s$ where $r \setminus s' \models Q$.

Intuitively, the blamed tuples of a QC defined over a given relation are all the possible combinations of tuples whose removal ensures that the remaining tuples of the relation no longer violate the QC.

An MDR is a template for an action that a user can undertake. That action is called an *MDR instance*. More rigorously, an MDR and an MDR instance are defined as follows.

Definition 2. An MDR M over a relation $R(A_1, ..., A_n)$ consists of a pair $\langle V, Ac \rangle$, where V is an updatable view over R projecting a subset of the attributes of R and Ac is an action that can be performed over V than can be one of $\{insert, delete, update\ A_1, ..., update\ A_n\}$ [9].

There are three types of MDR instances, each one reflecting the type of action associated with the MDR used to define the instance.

Definition 3. An MDR insert instance m_i is defined as the pair $\langle M_i, t_i \rangle$, where M_i is an MDR with action $insert$ and t_i is the tuple to be inserted.

An MDR delete instance m_d is defined as the pair $\langle M_d, t_d \rangle$, where M_d is an MDR with action $delete$ and t_d is the tuple to be deleted.

An MDR update instance m_u is defined as the triple $\langle M_u, t_u, v \rangle$, where M_u is an MDR with action $update\ A$, t_u is the tuple to be updated and v is the new value that will be assigned to attribute A.

Note that a tuple t in the instance of an MDR M must always obey the schema of the view of M.

In order to execute a data cleaning graph of transformations, each transformation is executed according to a partial order defined in the graph, generating the output relations of that transformation. Then,

the MDR instances defined over each relation are applied. Finally, the set of blamed tuples of each QC defined over those relations is computed. With the previous definitions we can now present the operational semantics of a data cleaning graph.

Definition 4. Let $\Gamma = \langle G, \langle Q, M \rangle \rangle$ be a data cleaning graph for a set $R_1, ..., R_n$ of input relations. Let $r_1, ..., r_n$ be instances of these relations and M be a a manual data repair state for Γ, *i.e.*, a function that assigns to every relation $R \in relations(G)$, a list of instances of manual data repairs over R. The notattion $^{\bullet}T$ and $T^{\bullet}$ denotes the sets of input and output relations of a transformation T, respectively. The result of executing Γ over $r_1, ..., r_n$ and M is $\{\langle tuples(R), tuples^{b}l(R) \rangle : R \in relations(G)\}$ calculated as follows:

for $i = 1$ **to** n **do**
  for each** $\iota \in M(R_i)$ **do**
    $s \leftarrow compute\_view(view(\iota), tuples(R_i))$
    $apply\_mdr(\iota, s)$
    $tuples(R_i) \leftarrow propagate(s)$
  end for
end for
for $i = 1$ **to** n **do**
  $tuples^{bl}(R_i) \leftarrow blamed(tuples(r_i))$
end for
for each* $T \in transformations(G)$ **do**
  let $\{R'_1, ..., R'_k\} = {}^{\bullet}T$
  $tuples(T^{\bullet}) \leftarrow T(tuples(R'_1, ..., tuples(R'_k))$
  for each** $\iota \in M(T^{\bullet})$ **do**
    $s \leftarrow compute\_view(view(\iota), tuples(R_i))$
    $apply\_mdr(\iota, s)$
    $tuples(T^{\bullet}) \leftarrow propagate(s)$
  end for
  $tuples^{bl}(T^{\bullet}) \leftarrow blamed(tuples(T^{\bullet}))$
end for

$*$ Assuming that the underlying iteration will traverse the set in ascending element order
$**$ Assuming that the underlying iteration will traverse the list in proper sequence

Procedure $compute\_view(view, setOfTuples)$ encodes the application of the view to the base relation consituted by the setOfTuples. $apply\_mdr(mdrInstances, s)$ encodes the result of the updates generated according to the type of each $mdtInstance$. Finally $propagate(view)$ encodes de propagation of the updates on the view to the base table.

Chapter 3

CLEENEX

In this chapter we present the realization of the data cleaning graph concept as proposed in [9], named CLEENEX, and describe the concretization details of a prototype implemented as an extension of the AJAX data cleaning framework. CLEENEX adds support for Quality Constraints (QCs) and Manual Data Repairs (MDRs), addressing the shortcomings of AJAX that were listed in Section 2.2.

In Section 3.1, we provide an overview of CLEENEX and how it differs from AJAX. In Section 3.2, we detail the support for QCs in CLEENEX, including the new *QC Manager* component. In Section 3.3, we describe the support for MDRs and the corresponding *MDR Manager* component. Finally, in Section 3.4.2, we present the modifications that were applied to the existing components of AJAX to accommodate the new constructs.

3.1 Overview of CLEENEX

The core tenet of AJAX, *i.e.*, the separation of logical and physical layers, was maintained. The designer has at her disposal two new languages to add QCs and MDRs to a data cleaning program. In order to facilitate the extension of AJAX, we decided to keep the definition of these new constructs separated from the data cleaning program. Therefore, the designer must supply two additional artifacts: the QC definition file and the MDR definition file. We chose to define new languages rather than extend the language in AJAX, but we strove to achieve a conceptual coherence with respect to the original language.

The data cleaning graph in CLEENEX is similar to the one in AJAX. However, the new constructs allow the program to be specifically tailored to incorporate user feedback. The exception tuple mechanism in AJAX was entirely replaced by the enforcement of QCs. This means that all the external functions must be reviewed and reimplemented so that they do not signal errors when facing input that cannot be processed. This breaks some of the existing working code. In practice, this makes the new functions

easier to understand, on account of having the identification of data quality problems declared in the form of QCs, instead of hidden in the body of the function.

The execution of the data cleaning process in CLEENEX is similar to the one in AJAX. However, each Java class generated to encapsulate the execution of a data transformation also includes the code to process the QCs and MDRs associated to the output relations of that transformation.

As noted in Section 2.2.1, the enforcement of QCs does not remove the tuples that violate those constraints from the target relation. These tuples, henceforth referred to as *blamed tuples*, are instead flagged for user inspection. In the developed solution, this flagging mechanism relies on the use of auxiliary relational tables in the RDBMS, called *blamed tables*. The system creates a blamed table for each QC and another relation that holds metadata for all QCs. This metadata includes the QC unique name, a condensed version of its definition and a unique numeric identifier. Blamed tables do not store blamed tuples directly. Instead, a blamed table for a given QC stores the key of each blamed tuple and the unique numeric identifier of the QC. This means that blamed tables establish relationships – using foreign keys – between blamed tuples and the QC.

An MDR is a template for an action that the user can undertake. The entity that represents the action itself is called an *MDR instance*. MDR instances are created using the GUI. Each relation has an associated container where the instances of its MDRs are stored. After the Java code for a data transformation is executed and its output relations are populated, all the instances of the MDRs associated to those relations are applied by order of creation. Thus, after the MDR instance is initially created, it will be applied each time its target relation is populated. MDR instances are also stored persistently at the moment of creation and reified during the compilation phase of the CLEENEX prototype.

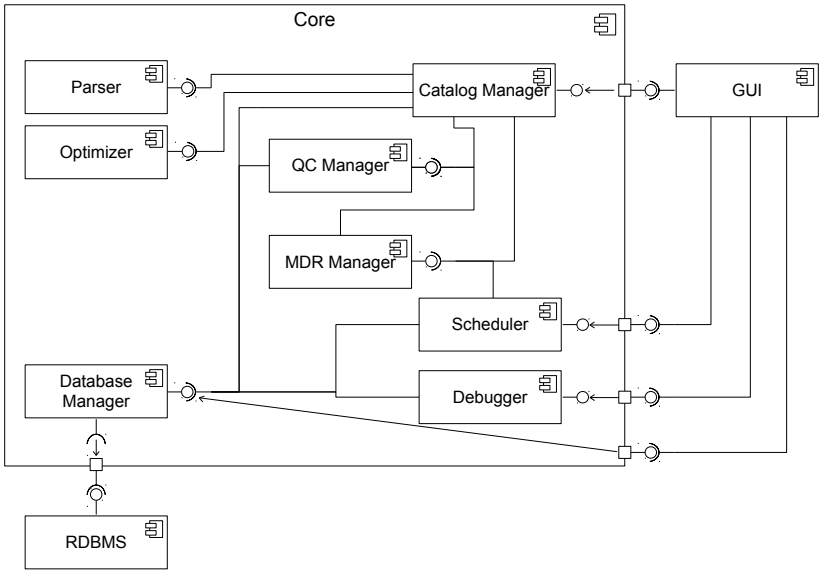


Figure 3.6: CLEENEX component architecture.

Figure 3.6 shows the component architecture of CLEENEX. In order to extend AJAX, we chose to add two new components: *QC Manager* and *MDR Manager*. The *QC Manager* includes the parser for the QC definition language and the mechanism used to generate the code to enforce the constraints. This code is included by the *Catalog Manager* in the generated code for data transformations. In addition, this component is responsible to create the blamed tables for each QC. The *QC Manager* must also expose QC information to the *MDR Manager*. Analogously, the *MDR Manager* includes the parser of the MDR definition language and encapsulates the mechanism to define MDRs and to construct and apply units of user feedback based on those MDRs. It was necessary to change the execution semantics in the *Scheduler* in order to incorporate MDRs in the execution of the data cleaning graph. Finally, we modified the AJAX *GUI* so that the user can analyze blamed tuples and create MDR instances. Moreover, we introduced a new data visualization mechanism at the GUI level for providing new visualization perspectives over the data records and thus assist the user on the decision to modify or insert data.

In order to illustrate the available types and usage of QCs and MDRs in CLEENEX we introduce part of a data cleaning graph that will be augmented with QCs and MDRs in Sections 3.2 and 3.3, respectively. In Figure 3.7, we present two relational tables that store intermediate data in an hypothetical process intended to obtain address data free of data quality problems. The relation *Addresses* with schema $(id, door, street, city, state, country)$ stores information about customer addresses that was divided into its constituent parts by the data transformation presented in Section 2.2.1. The relation *People* with schema $(id, name, residence)$ stores the names of persons and their place of residence. In both relations, the *id* attribute constitutes the primary key. The values of attribute *residence* reference tuples of *addresses*. This data may have some data quality problems: address information can be wrong or missing, the relationship between the two tables can be broken if there are tuples in *people* that reference locations not present in *addresses* and it is possible that the data in *people* is not completely normalized, referring more than one person per tuple.

Addresses:

ID	DOOR	STREET	CITY	STATE	COUNTRY
1	742	Evergreen Terrace	Springfield		USA
2	34	Elbvillenweg	Dresden	Saxony	Germany
3	1428	Elm Street	Springwood	Ohio	USA
4	86	Main	Dresden	Maine	USA
5	9	Conertplatz	Dresden	Maine	Germany

People:

ID	NAME	RESIDENCE
1	Marge Thompson	3
2	Bart Simpson	1
3	Homer and Marge Simpson	1
4	Erika Mustermann	5
5	Otto Normalverbraucher	6

Figure 3.7: Dirty relational data used in the example.

3.2 Support for Quality Constraints

Although [9] proposes the concept of QCs, it does not propose a specific constraint language to declare those constraints. It also does not offer a concrete proposal of the types of constraints that should be enforced. Finally, it does not offer a strategy on how to enforce the QCs. Throughout this section we address each of these points in turn. We also present the class diagram of the implementation of the *QC Manager* component.

3.2.1 QC Definition Language

We start by describing the types of QCs that we developed for this work. The types of QCs chosen are expressive enough to aid the discovery of data quality problems but not complicated enough that computing the set of blamed tuples for one of these QC impose significant delays in the remaining data cleaning process.

We considered the following five types of QCs:

- *Unique* – This type of constraint is used to enforce the condition that values of one or more attributes of the target relation should be unique in the set of tuples that constitute that relation. This QC can be used to identify possible errors in the data. In a relation r, with X a nonempty set of attributes of r and t_1 and t_2 tuples of r, the condition $Unique(X)$ holds iff

$$\forall t_1, t_2 \in r : t_1[X] = t_2[X] \Rightarrow t_1 = t_2 \tag{3.1}$$

- *Functional Dependency* – This type of constraint is used to enforce the existence of a functional dependency between two set of attributes in the target relation. This can be useful to identify errors in the data.

  According to [17], with r a relation, X and Y nonempty sets of attributes of r and t_1 and t_2 tuples of r, the functional dependency

$$X \rightarrow Y \tag{3.2}$$

  holds if

$$\forall t_1, t_2 \in r : t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y] \tag{3.3}$$

- *Inclusion Dependency* – This type of constraint enforces that the values of a set of attributes in the target relation be represented in the bag of values of a set of attributes with the same arity in

another relation. This can be used to identify erroneous values in the target relation.

Formally, with r and s relations, R a nonempty set of attributes of r, S a nonempty set of attributes of s, t_r a tuple of r and t_s a tuple of s, the inclusion dependency denoted as

$$R \subseteq S \tag{3.4}$$

holds if

$$\forall t_r \in r, \exists t_s \in s : t_r[R] = t_s[S] \tag{3.5}$$

– *Predicate* – This type of constraint enforces that each tuple t of a relation r obey a certain domain property. A predicate P identifying that property is defined as a function returning either $true$ or $false$ of the form

$$P(X, C) \tag{3.6}$$

where X is a nonempty set of attributes of r and C is set of constants. This constraint enforces that

$$\forall t \in r, P(t[X], C) = true \tag{3.7}$$

With C the empty set, a notable element of this family is the $not\,null$ predicate, enforcing that the values $t[X]$ be not null.

– *Candidate Key* – This type of constraint enforces that a nonempty set of attributes X of a relation r constitute a candidate key of t, *i.e.*, with t a tuple of r,

$$\forall t \in r, Unique(X) \wedge not\,null(t[X]) = true \tag{3.8}$$

Figure 3.8 extends the example introduced in Figure 3.7 with QCs to identify the data quality problems present. The violation of $qc1$ signals any tuple in the relation *Addresses* that lack a value for the *state* attribute. $qc2$ enforces the functional dependency that the values of the pair of attributes $\langle city, state \rangle$ determine the value of attribute *country* in the same relation. $qc3$ enforces the condition that all tuples in *People* have a different value for the attribute *name*. $qc4$ signals all tuples of that relation whose value for that attribute include the string "and" – this indicates that the data is not properly normalized. Finally, $qc5$ establishes the inclusion dependency between the attribute *residence* or *People* and the attribute *id* of *Addresses*.

Addresses:

ID	DOOR	STREET	CITY	STATE	COUNTRY
1	742	Evergreen Terrace	Springfield		USA
2	34	Elbvillenweg	Dresden	Saxony	Germany
3	1428	Elm Street	Springwood	Ohio	USA
4	86	Main	Dresden	Maine	USA
5	9	Conertplatz	Dresden	Maine	Germany

qc1: NotNull(state)
qc2: (city, state) → country

People:

ID	NAME	RESIDENCE
1	Marge Thompson	3
2	Bart Simpson	1
3	Homer and Marge Simpson	1
4	Erika Mustermann	5
5	Otto Normalverbraucher	6

qc3: Unique(name)
qc4: *notContains*(name, “and”)
qc5: residence ⊆ Addresses.id

Figure 3.8: Example of QCs in CLEENEX.

The language used to define the QCs is inspired by the AJAX specification language. However, it is more oriented to the task of data definition. Refer to Appendix A for the complete syntax of this language in extended BNF.

The QC definition file is divided in blocks. Each block refers to a relation in the data cleaning graph and lists all constraints that should be enforced over the data in that relation. Each constraint is composed of a unique name and a body. The body of a constraint identifies its type and can refer to attributes of the relation as well as embed literal numeric or string constants. In the case of an Inclusion Dependency Constraint it must also refer another relation in the graph. Listing 3.1 shows the QCs from the example in Figure 3.8.

3.2.2 Computation of Blamed Tuples

Each time a data transformation is executed, it populates its output relations. If there are QCs associated with one of those relations, then the corresponding blamed table must be populated as well. At that moment, the CLEENEX prototype executes SQL queries over the relations with associated QCs to compute blamed tuples. Note that these queries select those tuples of the relation that do **not** satisfy

```
Addresses{
  qc1: NOT NULL (state),
  qc2: FUNCTIONAL DEPENDENCY (city, state) DETERMINES (country)
}

People{
  qc3: UNIQUE (name),
  qc4: PREDICATE notContains(name, "and"),
  qc5: INCLUSION DEPENDENCY (residence) IN Addresses(id)
}
```

Listing 3.1: Example usage of the QC definition language.

the constraint.

Each type of QC has an associated SQL query template that is parameterized during the compilation phase to produce the query that will compute the blamed tuples at runtime. To check a relation for constraint violations, the SQL code is then injected in the body of the Java class that generates the data of that relation. Note that blamed tables do not store blamed tuples directly. Instead, the blamed table associated with a relation uses foreign keys to that relation and to the table that stores QC metadata to flag those tuples. Only the key attributes of the target relation are copied to the blamed table. Marking blamed tuples instead of copying them to new tables minimizes the amount of duplicate data required to identify data quality problems.

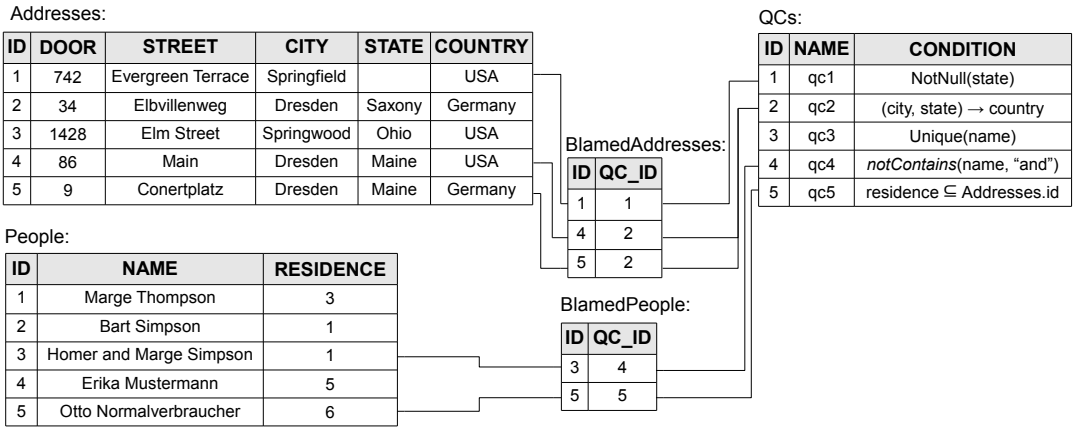


Figure 3.9: Blamed tables for the running example.

Figure 3.9 shows the blamed tables for the relations of the running example. *Addresses* and *People* are the relations introduced in Figure 3.7. *QCs* is the relation that stores QC metadata. *BlamedAddresses* and *BlamedPeople* are the blamed tables of *Addresses* and *People*, respectively. Tuples stored in the blamed tables establish relationships between their corresponding tables and the QCs that are violated. These relationships are used to mark the blamed tuples for user inspection. So, for example, tuples with ID 4 and 5 from *Addresses* are both blamed of violating $qc2$ and tuple 3 of *People* is blamed of violating $qc4$.

Listing 3.2 presents the SQL template queries that are used in CLEENEX to compute blamed tuples for three types of QC: functional dependency, inclusion dependency and predicate – refer to Appendix C for the templates of the remaining types of QC. These queries are responsible to populate blamed tables establishing relationships between the tuples that violate a given QC and the metadata of that QC in the global QC table. The terms in slanted font are replaced during the compilation phase with the appropriate expressions for each QC. Template queries are optimized for each kind of operation. *qc_id* is the unique identifier of the QC and is set when the code is generated.

The query to compute blamed tuples of a functional dependency QC uses a join rather than a subquery

```
--
-- template query for a functional dependency of the form
-- determinant → dependent
--
SELECT relation_pkey, qc_id
FROM relation
  NATURAL JOIN (
    SELECT determinant
    FROM relation
    GROUP BY determinant
    HAVING COUNT(DISTINCT dependent) > 1);

--
-- template query for a predicate QC
--
SELECT relation_pkey, qc_id
FROM relation
WHERE expression

--
-- template queries for an inclusion dependency of the form
-- relation.lhs ⊆ other_relation.rhs
--
CREATE INDEX index_name ON other_relation ( rhs );

SELECT relation_pkey, qc_id
FROM relation L
WHERE NOT EXISTS (
  SELECT R.rhs
  FROM other_relation R
  WHERE L.lhs = R.lhs );

DROP INDEX index_name;
```

Listing 3.2: Query templates used in CLEENEX.

as it is faster.

The query used to enforce an inclusion dependency uses an index. The index is created before the main query is executed to speed lookups and dropped afterwards.

The term $expression$ is replaced by an arbitrary expression for the *where* clause in the SQL query. The programmer that writes the predicate is responsible to ensure that the resulting SQL query is well formed. This subject is detailed in Section 3.2.3.

Listing 3.3 shows the SQL queries used to compute the blamed tuples of $qc2$, $qc4$ and $qc5$ of the example in Figure 3.9.

3.2.3 Implementation

In this section, we describe the implementation of the QC Manager component. The responsibilities of this component are:

- To parse the QC definition file.
- To manage the lifecycle of the QCs objects in memory.

```
--
-- query to enforce qc2 : (city, state) → country
--
SELECT id, 2
FROM Addresses
  NATURAL JOIN (
    SELECT city, state
    FROM Addresses
    GROUP BY city, state
    HAVING COUNT(DISTINCT country) > 1);

--
-- query to enforce qc4 : notContains(name, "and")
--
SELECT id, 4
FROM People
WHERE name ~ 'and'

--
-- query to enforce qc5 : residence ⊆ Addresses.id
--
CREATE INDEX temp_index5 ON Addresses ( id );

SELECT id, 5
FROM People L
WHERE NOT EXISTS (
  SELECT R.id
  FROM Addresses R
  WHERE L.residence = R.id );

DROP INDEX temp_index5;
```

Listing 3.3: SQL queries to compute blamed tuples for three of the example QCs.

- To create and populate the relation that stores QC metadata in the RDBMS.
- To generate SQL queries to compute blamed tuples.

Figure 3.10 shows the main classes in the implementation of this component, using a UML class diagram. The *QcManager* class exposes the public interface of this component. This class delegates the parsing of the QC definition file to *Parser*. The *QcManager* also creates and manages each QC, using a factory method for each type of QC. It creates the QC metadata table in the RDBMS as well. Each QC is represented by a subclass of *Constraint*. While the abstract method textitgenerateCode is used to generate at compile time the SQL query to find the set of blamed tuples for a QC, *setup* and *teardown* can be used to complement this code, *e.g.* they can be used to create and destroy indeces. *PredicateConstraint* is an example of a concrete subclass of *Constraint*. Each of these subclasses is responsible to generate the SQL queries to enforce a specific type of constraint, using the query templates presented in Listing 3.2.

The Predicate QC enforces a condition on each of the tuples of its target relation. This is an arbitrary condition, applied individually to each tuple, bound only by the domains of the attributes of that relation. In order to support different conditions, the subclass of Constraint that implements this type of QC – PredicateConstraint – is itself an abstract class that must be extended for each type of condition desired. While PredicateConstraint supplies the SQL query template allowing an arbitrary expression in

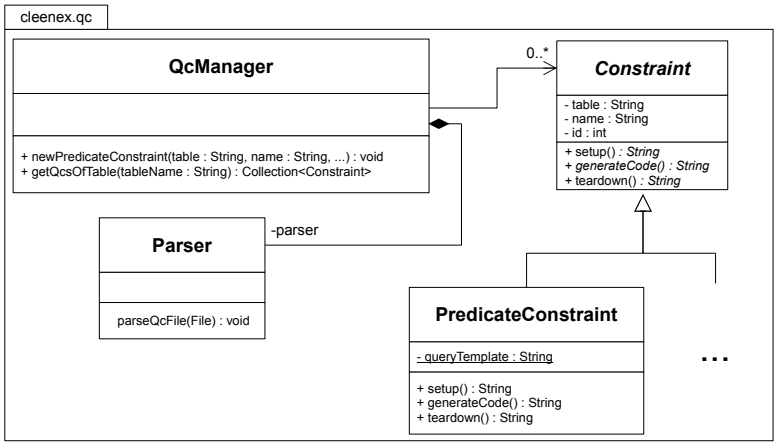


Figure 3.10: Excerpt of the class diagram of the qc.cleenex package.

the *where* clause, it is the responsibility of one of its subclasses to ensure that said expression is valid for the types of attributes involved. For example, in Figure 3.8, $qc4 : notContains(name, \text{“}and'')$ defined over the relation *People*, is enforced using a subclass of PredicateConstraint called *NotContainsConstraint* – the parser uses a simple naming convention to invoke the appropriate class using reflection. The *NotContainsConstraint* class ensures that the attribute *name* is of type string and generates an expression that marks the tuples of People that match the regular expression ”and”.

3.3 Support for Manual Data Repairs

In order to provide support for MDRs in CLEENEX, we developed an MDR definition language. Using this language, the designer can associate MDRs to relations in the data cleaning graph. Through these MDRs, the user can supply her feedback in the form of MDR instances as part of the data cleaning process.

In this section, we describe the support for MDRs in the CLEENEX prototype. We start by detailing the MDR definition language, then we describe the process of application of MDR instances to the data and we conclude the section by presenting the implementation of the *MDR Manager* component.

3.3.1 MDR Definition Language

Each MDR has an associated *Updatable View*. This view is used by the designer of the data cleaning program to limit the amount of data that the user must inspect in order to provide feedback. The designer accomplishes this by selecting tuples based on some condition or by projecting only relevant attributes in the relation. In addition, the designer can direct the user attention to the blamed tuples of an individual

QC associated to the target relation.

In CLEENEX, we support three types of MDRs:

- Insertion of a new tuple in the target view;
- Deletion of an existing tuple in the target view;
- Update of the value of a given attribute in an existing tuple in the target view.

Addresses:

ID	DOOR	STREET	CITY	STATE	COUNTRY
1	742	Evergreen Terrace	Springfield		USA
2	34	Elbvillenweg	Dresden	Saxony	Germany
3	1428	Elm Street	Springwood	Ohio	USA
4	86	Main	Dresden	Maine	USA
5	9	Conertplatz	Dresden	Maine	Germany

qc1: NotNull(state)
qc2: (city, state) → country

mdr1: update(state),
select *
from blamed(qc1)
mdr2: delete,
select city, state, country
from blamed(qc2)

People:

ID	NAME	RESIDENCE
1	Marge Thompson	3
2	Bart Simpson	1
3	Homer and Marge Simpson	1
4	Erika Mustermann	5
5	Otto Normalverbraucher	6

qc3: Unique(name)
qc4: *notContains*(name, “and”)
qc5: residence ⊆ Addresses.id

mdr3: insert,
select *
from People
mdr4: update(residence),
select *
from blamed(qc5)

Figure 3.11: Example of MDRs in CLEENEX.

Figure 3.11 presents an example that extends the example in Figure 3.11 with some MDRs. *mdr1* allows the completion of the tuples of the relation *Addresses* that do not have a value in attribute *state*, *i.e.*, those that are blamed of violating *qc1*. In this MDR, the designer chose to project all attributes of Addresses.

mdr2 is a delete MDR. If two or more tuple are flagged because they violate the functional dependenct enforced by *qc2* this MDR can be used to resolve the conflict, thorugh the deletion of some of those tuples. Note that this MDR projects only the attributes *city*, *state* and *country*, which are those involved in the functional dependency. *mdr3* allows the insertion of new tuples in the relation *People* and *mdr4* lets the user change the residence of a person that has an unrecognized current address.

To define MDRs we chose a language resembling the one used to define QCs. The various MDRs are aggregated by their target relation. They have a unique name and a body. The body of an MDR definition consists of an action – insert, delete or update of a single attribute – and an updatable view. The view is defined through an embedded SQL query. The *from* clause of this query can refer either to the target relation of the MDR or to the blamed tuples of a specific QC of that relation.

Listing 3.4 presents an example usage of the language for the MDRs presented in Figure 3.11.

The complete syntax of the MDR definition language can be found in Appendix B.

```
Addresses{
  mdr1: UPDATE(state) USING
      SELECT *
      FROM blamed(qc1)
    AS VIEW,
  mdr2: DELETE USING
      SELECT city, state, country
      FROM blamed(qc2)
    AS VIEW
}

People{
  mdr3: INSERT USING
      SELECT *
      FROM People
    AS VIEW,
  mdr4: UPDATE(residence)
      SELECT *
      FROM blamed(qc5)
    AS VIEW
}
```

Listing 3.4: Example usage of the MDR definition language.

3.3.2 Application of MDR Instances

An MDR is a template for an action that can be applied to a relation in the data cleaning graph. The concrete actions are called *MDR Instances*. Each of these instances encodes a unit of human feedback that can be re-applied multiple times during the data cleaning process. They can also be persistently stored with the intermediate data so that they are not lost if the data cleaning process needs to be interrupted and resumed at another time. In Section 2.2.3 we presented the formal definition of an MDR instance. In this section we describe how MDR instances are applied in the CLEENEX prototype.

The lifecycle of an MDR instance entails two different stages: creation and application. The creation of an MDR instance is triggered by a user, that selects an MDR and adds the required information to form an instance. Therefore, the system computes the views of each MDR during the execution of the data cleaning graph and presents the results of those views when the process is stopped waiting for user input.

In the current implementation of CLEENEX we use the views of the underlying RDBMS. These views are created during the execution of the data cleaning process. They are queried later when the user chooses to provide feedback with respect to a relation using the associated MDRs. The MDR instances that result from user feedback are stored in a in-memory FIFO container associated with that relation. When the user is satisfied with the new MDR instances, she re-runs the data cleaning process and all stored MDR instances are applied to the new data in the corresponding relation.

We opted for not using the support for updatable views of the underlying RDBMS. Even though the action encoded in an MDR instance is directed to the view of that MDR, we apply that MDR instance using a SQL query directed to the base relation referred by that view. This allows us to choose our

```
UPDATE People
SET residence = 2
WHERE ID = 5 AND
      NAME = 'Otto␣Normalverbraucher' AND
      RESIDENCE = 6
```

Listing 3.5: Example of SQL query generated to apply an MDR instance.

own definition of 'updatable view' regardless of the particular constraints of the underlying RDBMS. At the moment, the type of updatable views that are supported are those that target a single relation and perform only projections of attributes and selections of tuples.

Consider the example presented in Figure 3.11. Let us assume that the user chooses to repair the data in relation *People* using an instance of *mdr4*. According to the view of that MDR, the user is presented with a single tuple of the relation: $\langle 5, OttoNormalverbraucher, 6\rangle$, *i.e.*, the single tuple blamed of violating *qc5*. If we assume that the correct value for the attribute *residence* for that person is 2, then the user creates the MDR instance $\langle mdr4, \langle 5, Otto\ Normalverbraucher, 6\rangle, 2\rangle$. When the data cleaning process is re-executed and the relation *People* is populated again, the system will apply the user feedback, using the SQL query presented in Listing 3.5.

3.3.3 Implementation

In this section we describe the implementation of the *MDR Manager* component. The responsibilities of this component are:

- To parse the MDR definition file.
- To manage the lifecycle of the MDR objects in memory.
- To manage the lifecycle of the MDR instance objects in memory.
- To manage the persistent storage of MDR instance objects.
- To generate SQL queries to compute views of MDRs.
- To execute the queries that compute views of MDRs.
- To generate SQL queries to apply MDR instances.
- To execute the queries that apply MDR instances.

Figure 3.12 shows the main classes of the cleenex.mdr package that implements this component. The abstract class *PersistenceManager* stores and reifies MDR instances so that they are not lost during the program execution. *FlatFilePersistenceManager* is one of the three subclasses that currently extend PersistenceManager. Using the configuration file it is possible to select one of these three: one that disables persistent storage of MDR instances using the Null Object design pattern, one that uses a table in the RDBMS to store the instances and *FlatFilePersistenceManager*, that uses a flat file to store the instances.

The *Parser* class is responsible for the parsing of the MDR definition file. The creation of the defined MDRs is split between *MdrViewFactory*, that has one factory method for each subclass of *MdrView* (in Figure 3.12, we represented only *BlamedView* that covers the case where the target are the blamed tuples of a QC) and the abstract factory *ManualDataRepair*. *MdrInsert* is one of the concrete factories – there is one for each of the three type of MDR instance: **insert**, **delete** and **update**. Each of these subclasses of *ManualDataRepair* is also responsible to produce a query that is parameterized by the corresponding subclass of *MdrInstance* that it creates, so that the corresponding user action can be applied. We chose *MdrInstanceInsert* as an example for those subclasses.

The *MdrManager* exposes the public interface of the component. It can be used to create MDRs and their instances; its responsability is to delegate the execution of these tasks to the appropriate classes. It is however the responsibility of *MdrManager* to store the created objects, guaranteeing the association between MDRs and the corresponding relations. The *MdrManager* also computes views of MDRs and applies MDR instances using the respective generated SQL queries. Finally, the *ManualDataRepair* class hierarchy also implements an instance of the visitor design pattern that is exposed by the *MdrManager*. The *GUI* component has different classes that implement the *MdrVisitor* interface to gather information about the MDRs that are presented to the user.

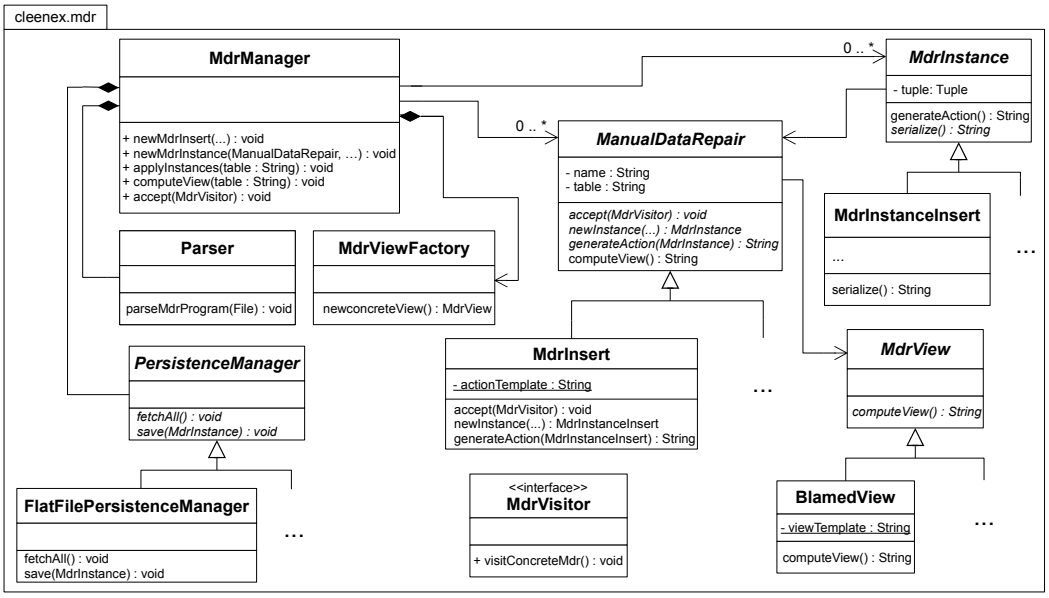


Figure 3.12: Excerpt of the class diagram of the cleenex.mdr package.

3.4 Cleaning Data with CLEENEX

In order to implement the execution semantics for a data cleaning graph with QCs and MDRs described in Section 2.2.3 we modified the AJAX *Catalog Manager* component and to the *Scheduler* component.

In addition, we altered the AJAX *GUI* to allow the user to inspect the data using the views of the MDRs and to create MDR instances.

3.4.1 CLEENEX Execution Semantics of a Data Cleaning Graph

The semantics of the execution of a data cleaning graph described in [9] imposes a circular dependency between MDRs and QCs of the same table. On one hand, since the view of an MDR can depend on the set of tuples that are blamed of violating a QC over that table, it is necessary to compute the blamed tuples of a table before applying MDR instances to the corresponding MDR view. On the other hand, if the goal of the MDR instances is to resolve data cleaning problems, and therefore ensure that the data does not have blamed tuples, it is necessary to apply MDR instances before computing the blamed tuples of a table and show these blamed tuples to the user.

The key point in understanding why this circular dependency is not a problem is to recognize that the data cleaning process is composed of multiple executions of the data cleaning graph interspersed with inspection stages where the user creates MDR instances. The data cleaning process starts in a state in which there are no MDR instances. Hence, in the initial state, the system is free to compute the blamed tuples taking into account only the output data returned by each transformation. The views are computed after finding the set of blamed tuples in order to present the data to the user during the inspection phase. After the user creates the new MDR instances and triggers the re-execution of the data cleaning graph, those MDR instances are applied to the view computed in the *previous* iteration.

In order to illustrate how the data cleaning process works, we use a simple example. Consider a data cleaning graph with one input relation *Ri* and two data transformations *T1* and *T2* sequentially composed as shown in Figure 3.13. The data cleaning graph also includes a QC *Q*, and an MDR *M* over the intermediate relation *R1*; the view *V* of *M* depends on the blamed tuples of *Q*.

Figure 3.13 illustrates what happens during two consecutive iterations of the data cleaning process. On the left side, it is represented the first execution of the data cleaning graph (*E1*) and, on the right side, a second execution of the graph (*E2*), after the user has provided her feedback (*UF1*) over the calculated data. In the example, this feedback is assumed to be *m1* and *m2*, two MDR instances of *M*.

These MDR instances are only applied after *T1* is executed in *E2*. The effects of the MDR instances change the contents of *R1*. At this point, *V* is computed again. On one hand, this means that *V* must persist between executions. On the other hand, it means that during each execution the appropriate order of operations for a relation *R* is: execute transformation generating *R* $\succ$ apply MDR instances $\succ$ compute blamed tuples $\succ$ compute views of MDRs.

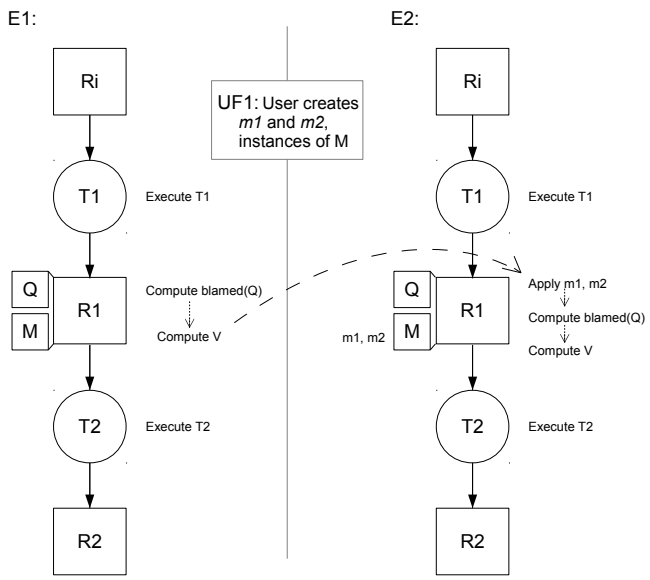


Figure 3.13: Overview of execution of a data cleaning graph in CLEENEX

3.4.2 Modifications Applied to Existing AJAX Components

There were three main modifications or novel functionalities added to the AJAX framework in order to support the execution semantics of a data cleaning program as proposed in [9]: *(i)* Code generation for each data transformation and execution of a data cleaning graph taking into account QCs and MDRs; *(ii)* graph traversal strategies; *(iii)* extension of the graphical user interface to support MDRs; and *(iv)* new GUI mechanisms to assist the user when providing feedback. We detail each of them in what follows.

In Section 2.1.4, when describing the AJAX components, we mentioned that the *Catalog* component generates Java classes that execute the logic of each data transformation. These classes are executed in the execution phase, ordered according to the data cleaning graph. During the implementation of the CLEENEX prototype, we chose to maintain this aspect of AJAX. However, it was necessary to tweak the process of code generation to include the processing of QCs and MDRs.

When generating the class that applies a data transfomation the first step is to apply the logic of the transformation to the input tuples. This creates the output data of the transformation. This block of code is exactly the same in AJAX and in CLEENEX. Next, the system applies all the instances of MDRs associated with the output relations. Notice that these instances are applied to views that were computed the previous time that this data transformation was executed, as explained in Section 3.4.1. The next step is to compute the blamed tuples for all QCs associated with the output relations of this data transformation. This will mark data cleaning problems that remain unresolved after applying MDR instances. Finally, the last step is to compute the views for each MDR associated with an output relation of this data transformation. These views filter the data that will be presented to the user so that she can produce new MDR instances. These views will also persist in the RDBMS until the next time that this

data transformation is run.

AJAX executes the data transformations in the same order that they were defined in the data cleaning program. Since the cleaning program constitutes a DAG, AJAX simply assumed that the designer had chosen one of the possible topological orderings. Although it was not strictly necessary, we chose to add a set of classes to the *Scheduler* component that allow the CLEENEX prototype to choose its own order for the execution of data cleaning transformations. The current implementation exposes a visitor interface to the internal components of this graph, improving the extensibility of the prototype.

We implemented two different graph traversal strategies. The first strategy uses the topological sort algorithm described in [4] to find an ordering to execute the transformations in the graph from start to finish. The second strategy is used to execute the graph from an intermediate transformation to the end. We use depth-first search to identify all transformations reachable from that intermediate transformation and perform the intersection of this set with the ordered set returned by the topological sort of the complete graph. By exposing the internals of the graph data structure, it is possible to add new traversal strategies. An interesting possibility is to implement a graph traversal strategy that constructs an ordering allowing parallel execution of independent branches of the data cleaning graph.

The *GUI* component of the AJAX prototype relies on a web application. Using the Java Servlet API, this application allows the user to interact with the data cleaning process, by inspecting data in intermediate relations and performing modifications in that data through manual data repairs. In the CLEENEX prototype it was necessary to add new servlets to list the MDRs of a given relation, show the data returned by MDR views and to let the user create MDR instances. The requirements of these new servlets motivated the inclusion of the visitor interface in the MDR *Manager* component, described in Section 3.3.3. This interface lets the programmer that implements each servlet have the power to extract the information that she needs, without requiring to change the implementation of the *MDR Manager* component to suit each new servlet.

To present the data returned by a view of an MDR, the appropriate servlet generates the required HTML code to render a table. Depending on the type of action that the MDR supports, different GUI widget are dynamically inserted in the web page. Each widget allows the user to create MDR instances in a different way. For example, to support an MDR of type delete, the GUI adds one button to each row displayed. Each of these buttons triggers a Javascript function that dynamically changes the colos the corresponding row to red, marking that row for deletion when the data cleaning process is restarted. This is accomplished by th creation of one or more MDR instances.

While MDR views are used to limit the amount of data that the user has to inspect, we decided to add an additional mechanism to let the user manipulate how she views that data. We believe this mechanism facilitates the task of providing feedback. The manipulations that the user can perform are:

– Sort rows – The header of each column is a button that, when pressed, sorts the rows according to

the ascending order of the values in that column. This way the user can view the tuples in whatever order she deems to be more useful.

– Filter rows – The user can mark cells in the table with the mouse. By pressing a button in the interface labeled “filter”, systems hides all rows that do not share the same values for the selected attributes. This allows the user to select which rows she wants to inspect, using the selected values as filtering condition.
– Hide columns – The user can mark one or more columns and press a button in the interface labeled “hide”. After removing one or more columns, two rows that have the exact same data are collapsed into a single row. That row represents more than one tuple in the original view. This allows the user to create multiple MDR instances with a single action.

(a)

CITY	STATE	COUNTRY	
Dresden	Maine	USA	delete
Dresden	Maine	Germany	delete

(b)

CITY	STATE	
Dresden	Maine	delete

Figure 3.14: View of an MDR of type delete.

Consider again the example of dirty address data from Figure 3.11. *Addresses* is a relation with schema $(id, door, street, city, state, country)$. Over *Addresses* are defined $qc2 : (city, state) \rightarrow country$ and $mdr2 : delete\, using\, SELECT\, city,\ state,\ country\, FROM\, blamed(qc)\, as\, view$. Figure 3.14 illustrates the action of hiding one column. Part (a) shows the data resulting from the view of $mdr2$: the user see two tuples that violate the constraint and can delete one or both of them using the corresponding buttons. Part (b) shows the same data with column *country* hidden. The tuple shown in (b) refers to both tuples shown in (a). This means that the action of deleting that tuple results in the creation of two MDR instances: $\langle mdr2, \langle Dresden, Maine, USA\rangle\rangle$ and $\langle mdr2, \langle Dresden, Maine, Germany\rangle\rangle$. Note that the mapping of shown rows to created MDR instances is the complete responsibility of the GUI. All three mechanisms can be combined by the user, effectivelly parameterizing the SQL query that shows the data without having to write a single character.

Chapter 4

Experimental Validation

In this chapter, we detail two experiments performed to validate and evaluate the CLEENEX prototype. We concentrate our tests on the trade-off between having more accurate results at the cost of increased user effort and on the impact of the computation of blamed tuples of QCs in the total runtime of a data cleaning process. Section 4.1 elaborates on how we evaluate the new constructs in CLEENEX. Section 4.2 presents the first experiment, dealing with user effort, and Section 4.3 describes the second experience, evaluating the costs of the computation of blamed tuples. Finally, Section 4.4 summarizes our findings.

4.1 Evaluation

In order to evaluate CLEENEX, we performed two different experiments. In CLEENEX, Quality Constraints and Manual Data Repairs for a data cleaning graph have to be defined by its designer. This imposes extra work on the side of the designer but aims at facilitating the incorporation of user feedback. As it was discussed in Chapter 1, it is not viable to design an automated data cleaning process whose output is a completely clean dataset. To do so would require so much fine-tuning of data transformations that, in the limit, one would need to write rules to clean each individual tuple. Therefore, it is assumed that the output of the automated data cleaning process must be reviewed manually by a human.

We judge the performance of the approach to data cleaning supported by CLEENEX based on:

– *Accuracy* – The use of the new constructs should result in higher accuracy of the output data of the automated cleaning process. In this context, “higher accuracy” means fewer data quality problems. Note that more accurate output data results in a smaller dataset that needs to be manually cleaned in a final passage.

– *User Effort* – We are leveraging the work of the user to increase the accuracy of the data cleaning process. Therefore, we must ensure that providing feedback during the data cleaning process requires less user effort than cleaning the resulting data manually, without that support.

– *Computational Impact* – The computation of the blamed tuples incurs a penalty on the execution time of the data cleaning process. It is necessary to ensure that the extra running of finding QC violations does not overshadow the execution time of data transformations.

4.2 Accuracy and User Effort

In order to judge the accuracy gains by incorporating the user feedback in the data cleaning process, we conducted an experiment similar to the one performed in [9]. However, any accuracy gains must take into account the user effort in providing feedback. Section 4.2.1 details the dataset used for the experiment. Section 4.2.2 describes the data cleaning program that we created for this experiment. Finally, in Section 4.2.3 we discuss our results.

4.2.1 Publications Dataset

Our real world dataset is comprised of publication data used in the CIDS project [5]. This dataset is composed of two tables: $team$ and $publications$. The table $team$ has schema $(name, email, citename)$, where $name$ is the full name of a team author, $email$ is the team author email and $citename$ is the short version of the name used in citation information, in the form "family name, initials of first and middle names". This table has 24 tuples. The tuples in table $publications$ were obtained through queries to Google Scholar[1] using team member's names. Out of the complete data we selected a sample of 432 publications, since we had to clean it by hand. The table $publications$ has schema $(title, authors, year, bibtex, link, cits, citsns)$, where $title$ is the title of the publication, $authors$ is the list of authors of the publications, each one in the same form described for $team.citename$, $year$ is the year of publication, $bibtex$ is the publication information in bibtex format, $link$ in an internal link returned by Google Scholar for that publication, $cits$ is the number of citations of that publication and $citsns$ is the number of citations that are not self-citations.

There are several possible data quality problems in the data in the $publications$ table:

– Duplicate publications.
– Publications authored by an homonym of a team member.
– Missing publication data – in particular, the year of publication.
– Same author referred by different names.

[1] http://scholar.google.com

– Typographical errors in the publication title.

The goal of the data cleaning process is to have in table $publications$ only one tuple for each publication authored by at least one team member, with accurate title, author list and year. In addition, every author that is a team member must be referred by his $citename$.

4.2.2 Publications Cleaning Program

In order to evaluate the accuracy gains and required user effort associated with QCs and MDRs, we designed a data cleaning program to clean the $publications$ table of the publications dataset described in Section 4.2.1. The data cleaning graph that models that program is composed of two input relations, thirty five data transformations – and the corresponding output relation of each transformation – four QCs and five MDRs.

The high number of data transformations precludes us from presenting each transformation in detail. Instead we provide a high level overview of the graph, dividing it in logical subgraphs that we refer to as "activities". Each of these activities can be said to accomplish a specific goal inside the data cleaning process, being composed of data transformations and their corresponding output relations. Some of these relations are used to link activities (note that there is no support in CLEENEX for activities; this is a concept we use here just to facilitate the explanation of how the program was designed and how it works).

Figure 4.15 shows the activities in the publications data cleaning program. Each activity is represented by a rectangle with rounded corners (borrowing the notation from UML activity diagrams). The small squares associated with activities are the linking relational tables. Rectangles are used to represent the input relations and final output relation of the data cleaning process. Each relation has small label with its schema. Note that these schemas are slightly different from those used in the program. In particular, the schemas presented here lack the identifiers used in relational joins in the program. These were omitted because their presence would complicate the diagram.

The schemas for the input relations were previously described in Section 4.2.1. In Activity1, we project the publication information that will be cleaned: its title, the list of its authors and its year of publication. In Activity2, we normalize the author list, extracting all unique author names. In Activity3, all known authors are approximately matched with the team members using only the similarity of names. Note that we have only the surname and initials of first and middle names of each author. We used the Jaro-Winkler similarity metric [21], assigning a higher weight to the initials. In our comparison we consider that a similarity of 0.97 indicates a match and a similarity of 0.93 indicates a possible match. In Activity4 we compute the coauthorship of all possible team members. We will use this information to eliminate false possible matches in Activity5.

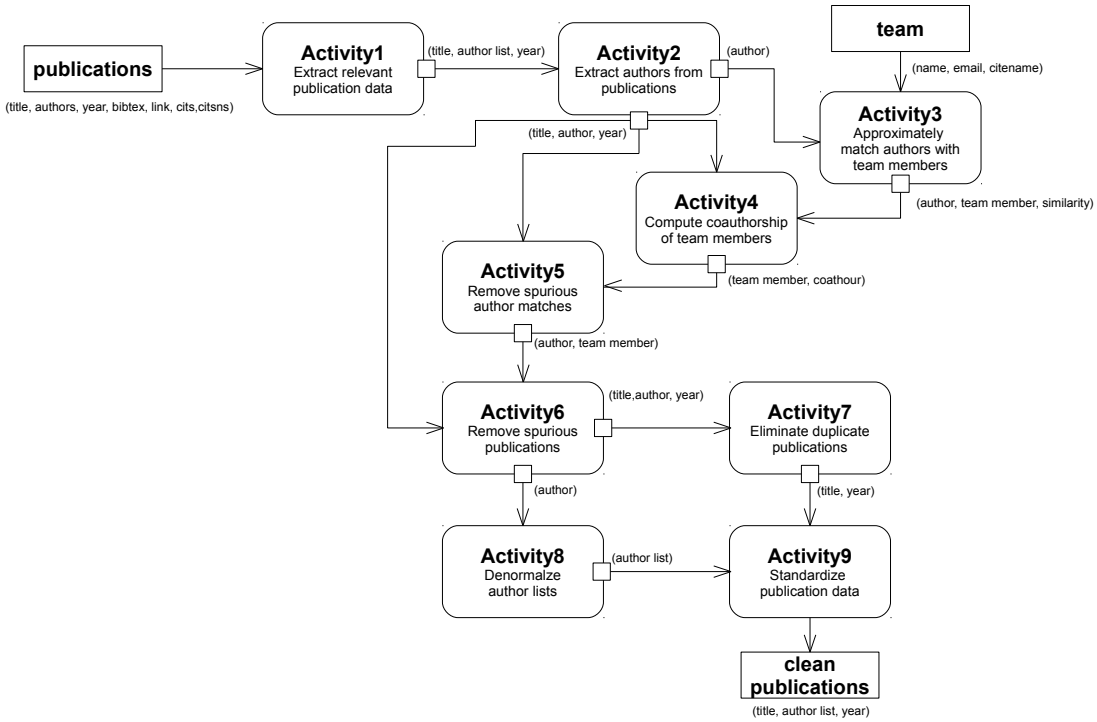


Figure 4.15: Activities in the publications data cleaning graph.

In Activity5, we use the similarity information from Activity3 to group all authors in clusters, using transitive closure. Each cluster contains all authors that are matches or possible matches of one team member. In order to eliminate false possible matches we use the coauthorship information of each author computed in Activity4. Each cluster is further divided into subclusters where authors in each subcluster have at least one common coauthor. We eliminate subclusters that only have possible matches with team members. Note that we cannot remove authors that are exact homonyms of team members. Activity5 is used to remove false positive matches that were introduced in Activity3.

In Activity6 we use the updated information associating known authors with team members from Activity5 to remove all publications that were not authored by at least one team member. In Activity7, we eliminate duplicate publications using a *Match*, a *Cluster* and *Merge* in sequence. We use the Levenshtein Distance to compare titles and the Jaccard Index to compare the lists of authors. In parallel, Activity8 denormalizes the list of authors of each publication. Finally, in Activity9 we join the author lists from Activity8 with the rest of the publication data from Activity7 and standardize the format of each tuple. The output of Activity9 is the clean publication data.

Listings 4.1 and Listing 4.2 show, respectively, the QCs and MDRs added to the data cleaning graph of the publications program to incorporate user feedback.

In Activity2 there is a *Map* data transformation that extracts authors from the author list of each publication. In this author list it is possible that some authors were omitted and referred only by the word

```
AuthorsByPublication
  NotNullAuthor: NOT NULL (fname, lname),
}

ApproximateJoinAuthorTeam{
  GreaterThan97Accuracy: PREDICATE greaterThan(sim, 0.97),
  AuthorMatchesOneTeamMember: FUNCTIONAL DEPENDENCY (authorid) DETERMINES (teamid)
}

MergeTitleMatches{
  NullYear: NOT NULL (year)
}
```

Listing 4.1: QCs used in the publications data cleaning program.

"others"; in this case the transformation creates a tuple with null firstname and lastname in its output relation: *AuthorsByPublication*. This relation has schema $(pubid, title, name, fname, lname)$, where $pubid$ identifies the publication, $title$ is the tile of the publication, $name$ is the author full name, $fname$ is the author first and middle initials and $lname$ is the author lastname. We added the QC *NotNullAuthor* to draw the attention of the user to *AuthorsByPublication* and the corresponding MDR *InsertAuthorName* to let insert the missing authors.

The core data transformation in Activity3 is the *Match* that performs the approximate join between author names and team member names. This transformation is called *ApproximateJoinAuthorTeam* and has a single output relation with the same name with schema $(teamid, teammember, authorid, authorname, similarity)$. In this transformation, the first case that warrants user inspection is the possibility that the same author is an approximate match to more than one team member. Tuples that fall in this category can be identified because they violate the functional dependency that an author should determine the matched team member. We added QC *AuthorMatchesOneTeamMember* to find the tuples in the relation *ApproximateJoinAuthorTeam* that match this case and the corresponding MDR *DeleteMultipleMatch* that allows the user to fix the problem by deleting one of the matches. We are also interested in drawing user attention to the results of the approximate matching itself. Remember that we defined as a possible match any match that has a similarity value between 0.93 and 0.97. QC *GreaterThan97Accuracy* alerts the user to tuples that are only possible matches. The corresponding MDR *UpdateAuthorSimilarity* allows the user to inspect the possible matches. By updating the similarity value, the user may reject those matches – setting the value to 0 – or confirm those matches – setting the value to 1.

The system computes the coauthorship of possible team members in Activity4 using a data transformation called *TeamCoauthorship*. This transformation stores its results in a relation with schema $authorid, authorname, coauthorid, coauthorname$. However, the resulting data is subject to the errors made when identifying team authors in Activity3. In particular, the system will always assume that two authors that are exact homonyms are the same person. In order to incorporate user feedback in this step, we created the MDR *DeleteTeamCoauthor*. Through this MDR, the user can remove author names

```
AuthorsByPublication {
  InsertAuthorName: INSERT USING
      select *
      from blamed(NotNullAuthor)
    AS VIEW
}

ApproximateJoinAuthorTeam {
  DeleteMultipleMatch: DELETE USING
      select teammember, authorname, similarity
      from blamed(AuthorMatchesOneTeamMember)
    AS VIEW,
  UpdateAuthorSimilarity: UPDATE(similarity) USING
      select teammember, authorname, similarity
      from blamed(GreaterThan97Accuracy)
    AS VIEW
}

TeamCoauthorship {
  DeleteTeamCoauthor: DELETE USING
      select authorname, coauthorname
      from TeamCoauthorship
    AS VIEW
}

MergeTitleMatches{
  UpdateYearTitleMatches: UPDATE(year) USING
      select title, year
      from blamed(NullYear)
    AS VIEW
}
```

Listing 4.2: MDRs used in the publications data cleaning program.

that never coauthored publications with a given team member. By removing that coauthorship association, the user is leading the system to the detection of homonyms of team members. This information will increase the system accuracy when removing spurious author matches in Activity5.

In Activity6 we perform the elimination of duplicate publications based on the similarity of their titles, lists of authors and year of publication. In this task there is a *Merge* data transformation called *MergePublicationMatches* that merges duplicate publications. Its output relation has the same name and schema $(pubid, title, year)$. If two publications considered duplicates have different years the system cannot choose the appropriate value. Instead, this data transformations output the publication tuple with a null year, expecting the user to provide the correct value. This accomplished using the QC *NullYear* to identify those tuples and the corresponding MDR *UpdateYearTitleMatches* through which the user can provide the missing information.

In order to further illustrate the association of QCs and MDRs to relations in this data cleaning program we detail Activities 3 and 4 in Figure 4.16. Relations are represented as white rectangles and data transformations are represented as ellipses annotated with the transformation type. Two relations have associated QCs and MDRs, represented as gray rectangles.

Activity3 is composed of two transformations and respective output relations: *TeamMembers* and *ApproximateJoinAuthorTeam*. *TeamMembers* assigns a unique identifier to each team member in the input

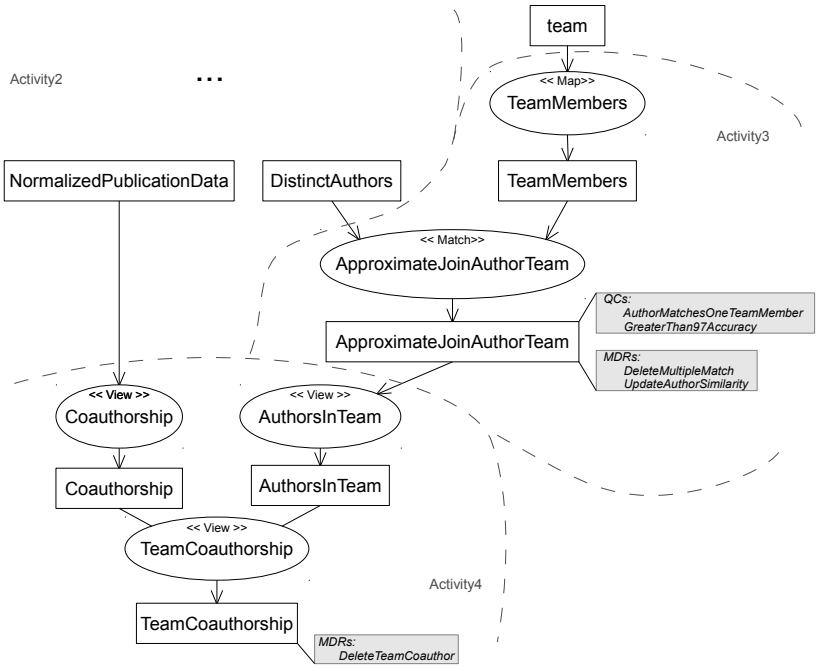


Figure 4.16: Detail of activities 3 and 4 of the data cleaning program.

relation *team*. *ApproximateJoinAuthorTeam* performs the approximate join between team members and all known authors, present in the relation *DistinctAuthors* computed in Activity2. The QCs and MDRs associated with the result of the approximate match can be used to inspect and repair possible errors in the operation.

The relation *NormalizedPublicationData* has schema $(authorid, authorname, publicationid, title, year)$ and stores all relevant publication data in first normal form, where each tuple associates one author with one of his publications, complete with the information of that publication. This data is used by the transformation *Coauthorship* to associate every author with known coauthors, using a self-join. The transformation *AuthorsInTeam* extracts the authors that are likely to be team members from *ApproximateJoinAuthorTeam*, *i.e.*, those with a name Levenshtein similarity higher than 0.93 . Finally, the transformation *TeamCoauthorship* selects only the coauthorship of possible team members, performing a join between the data in *Coauthorhsip* and *AuthorsInTeam*. Two authors that are exactly homonyms are identified as the same author. Through MDR *DeleteTeamCoauthor*, the user can identify authors that were spuriously assumed as coauthors of a given team member. The user feedback provided at this stage will be used later in Activity6 to remove publications authored by homonyms of team members. Note that the intricacies of the data cleaning graph is completely hidden from the user.

4.2.3 Results

By incorporating user feedback in the data cleaning process it is natural that the accuracy of results increases; we have already stated that, to have completely accurate results, it is necessary to have the user perform a final cleaning passage by hand. The goal of QCs and MDRs is to shift user effort from cleaning the final resulting data of the automated cleaning process to providing feedback in certain stages during the process, where smaller effort results in higher accuracy in the output data.

To evaluate the usefulness of the new constructs in real world data we performed an experiment using the publications dataset and the corresponding data cleaning program described in Section 4.2.2. We performed three tasks:

- *T1* – Manual cleaning of the *publications* table, obtaining *T1*^{cleanPublications}. This was accomplished by consulting the web pages of team members and consulting a copy of every publication listed in $publications$.
- *T2* – Automatic cleaning of the *publications*, obtaining *T2*^{cleanPublications} without any kind of user intervention during or after the data cleaning process.
- *T3* – Automatic cleaning of the *publications* incorporating user feedback with the creation of MDR instances, using all the capabilities of the CLEENEX prototype and obtaining *T3*^{cleanPublications}.

Note that these tasks were performed in order, by the same person. Since it was beyond the scope of our thesis to factor in the expertise of the user, we assume that in *T3* the feedback is provided by a user that is never wrong. When creating MDR instances, we based our feedback only in the data displayed and using only the action allowed in the MDR.

By definition, *T1* is completely accurate. The accuracy of *T2* is computed by comparing the contents of *T2*^{cleanPublications} with those of *T1*^{cleanPublications} (the process is analogous for *T3*). Note that we consider a tuple t in *T2*^{cleanPublications} as being equal to a tuple t' in *T1*^{cleanPublications} if it refers to the same publication and have the same id. For each tuple in *publications*:

- If the corresponding tuple (*i.e.* the tuple with the same id) is present both in *T2*^{cleanPublications} and *T1*^{cleanPublications}, it counts as a **True Positive (TP)** for *T2*.
- If the corresponding tuple is absent from both *T2*^{cleanPublications} and *T1*^{cleanPublications}, it counts as a **True Negative (TN)** for *T2*.
- If the corresponding tuple is present in *T2*^{cleanPublications} but absent from *T1*^{cleanPublications}, it counts as a **False Positive (FP)** for *T2*.
- If the corresponding tuple is absent from *T2*^{cleanPublications} but present in *T1*^{cleanPublications}, it counts as a **False Negative (FN)** for *T2*.

Based on the classification of each input tuple by *T2* and *T3*, we compare the performance of each task using the metrics of *Precision* and *Recall*. These metrics are defined as [15]:

$$Precision = \frac{TP}{TP + FP} \tag{4.9}$$

$$Recall = \frac{TP}{TP + FN} \tag{4.10}$$

We present in Table 4.1 the obtained results. The first four rows show the tuple count in each category. The last two rows show the Precision and Recall scores for each task. Each task is represented in one of the columns.

When cleaning the *publications* by hand, we removed all information other than title, list of authors and year of publication. The id used in *T1*^{cleanPublications} is a specific substring embedded in the $link$ attribute of the original *publications* relation – the same criteria used in the data cleaning program we created. Although we updated the contents of the tuples to reflect the exact information found in the copy of the corresponding publication, for the purposes of this component of the experiment, we are only interested in tuple deletions. We removed 197 tuples from the original 432 – classified as **True Negative**. These tuples were rejected because: (1) they were not authored by any team member or (2) they were duplicates records of a publication already present in the table. Both *T2* and *T3* show a high Recall score, *i.e*, the data cleaning program maintains most of the publications authored by team members. However, this is accomplished by sacrificing the precision of the results. If one considers the Precision score as the probability that a publication selected randomly from the output data was authored by a team member and does not have a duplicate in the list, then *T2*, while being better than a coin-toss still has a wide margin for improvement. A careful analysis of the data shows that this is caused by the presence of publications authored by exact homonyms of team members. Due to the leveraging of user feedback using QCs and MDRs, *T3* shows a 35% gain in the Precision score in relation to *T2*. The gain in the Recall score can be considered negligible.

	T1	**T2**	**T3**
TP	235	230	235
TN	197	94	181
FN	–	5	0
FP	–	103	16
Precision	1	0.691	0.936
Recall	1	0.979	1

Table 4.1: Accuracy of the results of each task.

Since the higher accuracy in *T3* is the result of direct user intervention, it is necessary to quantify the user effort required when providing feedback. Since the most basic approach to data cleaning is the manual approach, *T1* provides the baseline results for this experiment as well. However, because *T1* obtains

absolutely accurate results, we cannot compare it directly with *T2* or *T3*. Therefore, we introduced tasks *T2′* and *T3′*, where we take the output of the data cleaning program and clean it manually until we obtain the same data present in *T1*.

In order to judge user effort while cleaning relational data we consider both the number of tuples and characters that the user has to read and the number of tuples and characters that the user has to modify. When inserting or deleting a tuple we count the total number of characters in that tuple. When modifying a tuple we count only the absolute value of the difference between the original number of characters and the final number of characters in that tuple.

Table 4.2 shows the results obtained when measuring user effort. While *T1* and *T2′* only requires user intervention to perform the manual cleaning passage, *T3′* shifts some of that effort to the creation of MDR instances during the automated cleaning phase. *T2′* shows significant improvements over *T1*. Specifically, *T2′* requires the inspection of 24% fewer characters, the deletion of 48% fewer characters and the update of 48% fewer characters in relation to the control, *T1*.

The results show that the approach of *T3′*, using QCs and MDRs to include user feedback during the data cleaning process, results in a lower cumulative effort than simply manually cleaning the output data in *T2′*. The total user effort, both during and after the automated cleaning phase requires the inspection of 27% fewer characters, the insertion of 77% fewer characters and the deletion of 64% fewer characters in regard to *T2′*. The 9% increase in updated characters in *T3′* with respect to *T2′* is explained by the fact that some publications that are present in the output of *T3* can be updated whereas in *T2′* they must be inserted.

Since the 35% gain in the Precision scores of *T3* with respect to *T2*, shown in Table 4.1, is a direct consequence of the user effort in creating MDR instances and that effort is significantly lower than manually cleaning the output data, we conclude that there is a clear benefit in including the user feedback directly in the automated data cleaning process. The gain in the accuracy of results in *T3* in relation to *T2* directly correlates to the reduced user effort in the subsequent manual data cleaning.

It could be argued that, since the results with task *T3* already show very high Precision and Recall, one could dispense with the final manual cleaning phase altogether. In practice, however it is not generally possible to compute the accuracy of automatically cleaned data without having the same data already free of data quality problems to serve as benchmark. An interesting result is that the user inspects a high number of tuples in intermediate relations while creating MDR instances. In fact, she inspects more intermediate tuples than the number of tuples in the original dirty *publication* table. The difference is that those intermediates tuples have a much lower character count than the publication tuples in the final table. This happens because each intermediate table individually stores less data than the final clean table of publications. This effect is compounded by the selections and projections views used in the chosen MDRs.

			T1	T2'	T3'
User effort while creating MDR instances	Inspection	Tuples	–	–	552
		Characters	–	–	1 914
	Insertions	Tuples	–	–	2
		Characters	–	–	152
	Deletions	Tuples	–	–	143
		Characters	–	–	2 800
	Updates	Tuples	–	–	1
		Characters	–	–	13
Total user effort	Inspection	Tuples	432	333	803
		Characters	59 313	45 337	32 994
	Insertions	Tuples		5	2
		Characters		660	152
	Deletions	Tuples	197	103	159
		Characters	26 004	13 596	4 912
	Updates	Tuples	209	133	137
		Characters	1 333	690	758

Table 4.2: User effort required for each task.

4.3 Computational Impact of Blamed Tuples

Since we are introducing QCs, enforcing these new constructs is bound to have some impact in the execution time of data cleaning programs. Rather than testing the impact of computing blamed tuples for QCs in our publications program, due to the small fixed number of input tuples, we chose to generate an artificial dataset and evaluate the execution time of each type of data transformation and each type of QC individually. In Section 4.3.1 we describe that dataset and how it is generated and in Section 4.3.2 we discuss the obtained results.

4.3.1 Generating the Artificial Dataset

In order to conduct the experiment to evaluate the impact of the computation of blamed tuples in CLEENEX we chose not to use the dataset described in 4.2.1. We chose to construct an artificial dataset instead. By generating our own artificial dataset, it is easier to control how many input tuples we feed to each transformation. In particular, it becomes easier to subject each type of data transformation and each type of QC individually to an increasing number of input tuples.

We based our artificial dataset in an hypothetical database of a sales company. For our tests, we created a script that populates four input relations parameterizing the number of input tuples. We also defined one QC of each possible type over those relations:

– The *customers* relation, with schema $(id, name, country, phone)$ where:

  - id is an incremental numerical identifier.
  - $name$ is the customer name, chosen randomly from a pool of 200 synthetic names.
  - $country$ is the name of the country of the customer, one of the 27 countries in the European Union. We added the rule that one in ten generated tuples will have a null $country$.
  - $phone$ is the phone number of the customer. The generation of the values for this attribute includes the rule that 10% of the tuples will share a specific phone number.

  We defined two QCs over this relation:

  - A QC of type *Not Null*, defined over the attribute $country$.
  - A QC of type *Unique*, defined over the attribute $phone$.

- The *products* relation, with schema $(id, name)$, where id is a numerical identifier and $name$ is the product name, generated randomly.
- The *sales* relation, with schema $(id, customer, product, price)$, where:

  - id is a numerical identifier.
  - $customer$ is the name of the customer that bought the product. In order to test a QC of type *inclusion dependency*, 90% of the generated tuples will have values that appear in *customers*, while the remaining 10% will share the same customer name, guaranteed to be absent from that relation.
  - $product$ is the code for the product sold.
  - $price$ is the amount payed by the customer.

  The product code is guaranteed to be present in the *product* relation. In order to test a QC of type *Functional Dependency*, for 90% of the tuples the value for the $price$ attribute is 10 times the code in $product$. The remaining 10% share a value guaranteed to never be automatically generated. We defined three QCs over this relation:

  - A QC of type *Inclusion Dependency*, to flag tuples whose $name$ value is not contained in the possible values of the $name$ attribute of the *customers* table.
  - A QC of type *Functional Dependency*, to flag tuples that do not verify the functional dependency between the value of the $code$ attribute and the value of the $price$ attribute.
  - A QC of type *Predicate*, to flag tuples whose $price$ value is higher than 1000.

- The *clusteredCustomers* relation. This relation has schema $(clusterId, id, name)$. We use this table to test a data transformation of type *Merge*.

  - $clusterId$ is mandatory due to the underlying code generation and is a random positive integer where the maximum value is the number of tuples in the relation after it is generated. Tuples that share the same value for this attribute belong to the same cluster.

- id is an incremental numerical identifier and $name$ is chosen randomly from the pool of 200 synthetic names.

We tested one data transformation of each possible type:

- A transformation of type *Map*, that takes the *customers* relation as input and splits each customer name into first name and last name. It also standardizes the format of each component. It populates a relation with schema $(id, firstname, lastname)$, where id is the identifier of the customer in the input relation.
- A transformation of type *Match*. This transformation compares each possible pair of customer names, reading from the *customers* relation. In order to simulate the load of a typical approximate join operation, we compare names using the Jaro-Winkler distance. Because a *Match* with n the number of input tuples returns at most n^2 output tuples, rather than using than using the Jaro-Winkler distance value, we manipulate the matching condition to ensure that roughly only n pairs are inserted in the output relation. We use this hack because of the strict coupling between a *Match* and the subsequent *Cluster* imposed by the underlying implementation, inherited from AJAX.
- A transformation of type *Cluster*, that groups the pairs of customers returned by the preceding *Match* using transitive closure, so that the *Cluster* receives the same number of input tuples as the other transformations.
- A transformation of type *Merge*, that receives as input the *clusteredCustomers* relation, specifically crafted to test the *Merge* independent of the *Cluster*. This transformation elects a representative for each cluster, simply choosing the first alternative.
- A transformation of type *View*. Out of the possible types of SQL queries possible, we chose to perform a join between *products* and *sales*, using the id attribute of *products* and the $product$ attribute of *sales*. We populate an output relation with schema $(id, customerName, productName)$.

4.3.2 Results

In order to gauge the impact of the computation of blamed tuples in CLEENEX, we used our script to generate an increasingly larger number of tuples for each of the four input relations described in Section 4.3.1. Besides measuring the time spent computing blamed tuples for each type of QC, we also timed the execution of each of the five data transformations described in Section 4.3.1.

Our test environment consisted on an a laptop with an Intel Pentium M processor clocked at 2GHz, with 1GB RAM, running Windows 7 Professional 32 bits. The RDBMS used was PostgreSQL 9.0. In order to save on testing time, we enforced an upper bound of 15 minutes of execution for all data transformations. We made efforts to guarantee that this option would not affect our conclusions.

In Table 4.3 we present the results of out tests. Each column presents the execution times related to a certain number of input tuples. The top five rows present the execution times for each type of data

transformation, while the remaining five rows present the time spent computing the set of blamed tuples for each type of QC. All times are in milliseconds. The 900 000 value corresponds to the cut off at the 15 minute mark.

		2 500	5 000	7 500	10 000	12 500	15 000	17 500	20 000
Transformations	Map	5 283	8 477	12 776	17 340	21 839	25 458	28 177	31 466
	Match	52 102	198 308	434 891	763 010	900 000	900 000	900 000	900 000
	Cluster	61 850	413 021	900 000	900 000	900 000	900 000	900 000	900 000
	Merge	843	2 412	7 741	10 514	13 706	15 730	18 758	21 030
	View	239	503	447	715	735	1 008	1 141	1 778
Qcs	NotNull	20	41	50	56	60	64	98	139
	Predicate	33	39	52	56	73	78	79	110
	Unique	30	47	53	79	101	102	138	152
	Functional Dependency	53	111	127	145	186	219	288	461
	Inclusion Dependency	291	580	701	878	1 225	1 569	1 797	2 164

Table 4.3: Results obtained in the blamed tuples experiment.

The first conclusion that can be drawn from the data in Table 4.3 is that execution time is clearly dominated by the *Match* and *Cluster* data transformations. With 5000 input tuples, the highest value for which we have definite values for all data transformations and QCs, the computation time of the set of blamed tuples for the type of QC that takes the longest, *Inclusion Dependency* is less than 3% of the time it takes to execute the *Match* transformation and 1.4% of the time it takes to execute the *Cluster* transformation.

The other noteworthy result is that computing the set of blamed tuples for an *Inclusion Dependency* takes consistently more time than executing a data transformation of type *View* with a comparable number of input tuples. This is somewhat expected, because a *View* is just an SQL query executed in the RDBMS.

Figure 4.17 presents the sames values of Table 4.3 in bar chart form. Each transformation or QC is represented with a different pattern. On the horizontal axis we have the number of input tuples. On the vertical axis we have the execution time, in milliseconds. Note that we used a logarithmic scale in the vertical axis. The computation times for blamed tuples among each type of QC shows that *Predicate* and their subtype *Not Null* QCs take less time to enforce, followed by *Unique* and *Functional Dependency* and finally *Inclusion Dependency* takes the longest. The relative differences can be attributed to the type of SQL query used to enforce each type of QC: *Predicate* and *Not Null* use a simple query that translates into a linear scan of the target table, *Unique* and *Functional Dependency* use a self-join and *Inclusion Dependency* uses a nested subquery.

4.4 Discussion

In this chapter we reported and commented the results of our experiments. The experiment with the publications dataset, described in Section 4.2, showed a 35% gain in the Precision score and marginal gains in the Recall score of the results of a data cleaning program as a direct result of the incorporation

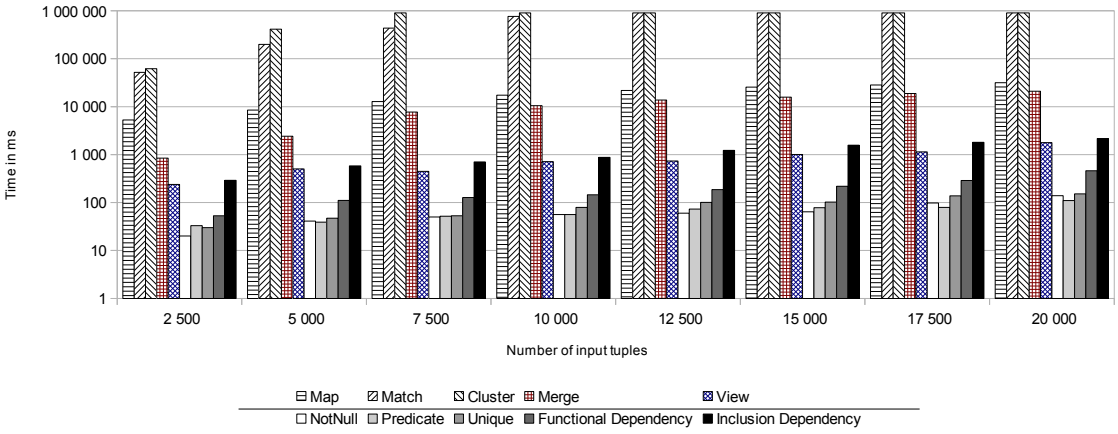


Figure 4.17: Computation time for each type of transformation and QC.

of user feedback through the use of QCs and MDRs. Using the metric of the number of characters visualized, providing her feedback required the user to examine only 3.2% of the complete dataset. Even when pitting the manual cleaning of the output of the automated data cleaning process without QCs or MDRs against performing the same manual cleaning after incorporating user feedback using QCs and MDRs, the user was required to inspect 27% fewer characters, insert 77% fewer characters and delete 64% fewer characters when using QCs and MDRs.

Note that we did not evaluate the relative designer effort to add QCs and MDRs to the graph. This must also take into account that adding these constructs forces the designer to effect some changes in the graph, in order to expose data that benefits from user inspection and feedback. We also did not evaluate how introducing new tuples in the input relations of the graph affects the application of MDR instances created in previous iterations. While this will benefit from the implementation of incremental execution of data transformations, *i.e.* processing only input tuples of a data transformation that were previously unknown, we foresee some difficulties in the process, related to the use of stateful generation of identifiers.

Regarding our experiment on the computational effort of enforcing QCs, presented in Section 4.3, our results show that enforcing *Inclusion Dependencies* takes more time that enforcing any other type of QC among those implemented – by an order of magnitude. Our results also show that the time of computation of blamed tuples of a *Inclusion Dependencies* is merely 1.4% of the execution time of a *Cluster* data transformation, the type that takes the longest. To put the relative commonness of each construct, out of thirty five data transformations in the publications data cleaning program, two were of the type *Cluster*; out of five QCs used, none of them was a *Inclusion Dependency*.

Chapter 5

Related Work

Data cleaning is characterized by lengthy, automated processes that iteratively executed over input dirty data. After each iteration, the data obtained must be analyzed to determine if it possesses sufficient quality to serve the interests of the data consumers. If it is determined that there are still too much data quality problems, the automated process must be fine-tuned and restarted. In general, the appraisal of data quality and especially the fine-tuning of the cleaning programs is left to human operators. Because it is not viable to completely eliminate human effort from the task of data cleaning, it is natural that data cleaning tools should provide facilities to help maximize the result of that effort. While the AJAX framework, described in Chapter 2, already showed some efforts in that regard namely the GUI and the Debug modules, it is only through the inclusion of QCs and MDRs in the context of this thesis that user intervention became a first-class citizen.

In this chapter, we present other data cleaning projects that somehow support user feedback during the execution of a data cleaning process. We also report on other approaches for incorporating human feedback in automated processes beyond the domain of data cleaning. We give a brief description of the architecture and operational semantics of each of those projects and discuss its relevance for the development of CLEENEX. While some of these projects follow a similar approach to user intervention in an automated process as the one in CLEENEX, others are included as a counterpoint to our work.

5.1 Potter's Wheel

We begin the presentation of other relevant academical works with Potter's Wheel. Potter's Wheel [18] is a data cleaning framework that also includes provisions for the incorporation of user feedback. This system aims at an interactive and semi-automatic design of the data cleaning program. The authors of this framework consider that the tasks of auditing to find discrepancies in data and designing data

transformations that need to be applied in order to correct the problems in the data should be integrated in the same tool. Moreover, in order to require less user effort, discrepancy detection is automatic. In the same vein, data transformations are specified through example or using form-like interfaces, rather than using a domain-specific language. The user is presented with a sample of the input data and she can apply several transformations, experimenting on that sample. When the user is satisfied with the results, the system compiles a data cleaning program that can be later applied to the whole dataset.

5.1.1 Architecture

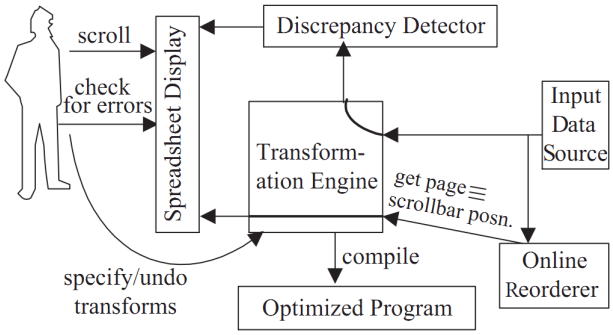


Figure 5.18: *Potter's Wheel Architecture [18].*

Figure 5.18 provides an overview of the architecture of Potter's Wheel. The *Input Data Source* reads a stream of records, the data to be cleaned. The *Online Reordered* fetches a portion of the records in the input data in order to present the user with a sample of the data. Meanwhile, the *Discrepancy Detector* analyses the input data, divides each record according to the domain of its recognized sub-components and applies discrepancy detection functions to each of those sub-components. This process will be further explained later in this section. Using the *Spreadsheet Display*, the user can check the detected discrepancies and choose which transformations the *Transformation Engine* should apply to the data. She can also scroll the window in order to obtain different samples of the dirty data. After the user has chosen the appropriate transformations, the *Transformation Engine* will compile a program to be applied to the dataset. Note that up to this point all modifications are performed on a temporary copy of the sample data.

The automatic discrepancy detection is based in domains. Besides the predefined standard domains (for example, *words* or *integers*), new domains can be defined by the user. To define a new domain, the user needs to specify the pattern to which values of that domain should conform. Consider an unstructured record ⟨*Homer Simpson, 388 years old, resident in 742 Evergreen Lane, Springfield*⟩. The system needs to infer that *Homer Simpson* is a name, *388* is an age, *etc.* before it can detect that *388* is not a valid age for a living person or that there is no *Evergreen Lane* in *Springfield*. However, because it is possible to define overlapping patterns, the system has to enumerate and rank all the possible

matching structures. In addition, the system needs to take into account that there are quality problems present in the data, which means that the optimal structure may not match all the records. Because of this, *optimal*, in this sense, is defined as the shortest structure that can match all input records in the sample with higher precision and recall. Considering the example again, $<$ *Address* $>$ is more concise than $<$ *Integer* $>$␣$<$ *Word* $>$␣$<$ *Word* $>$␣$<$ *Punctuation* $>$␣$<$ *Word* $>$ when trying to match *742 Evergreen Terrace, Springfield* and will therefore rank higher than the other alternatives.

5.1.2 Operators

Potter's Wheel provides a set of operators that can be applied to the data shown in the sliding window. A *Format* transforms each value of a given column. The applied function can be an arithmetic operation, a substitution using regular-expressions or a user defined function. *Add*, *Drop* and *Copy* are column operations with self-explanatory names. A *Merge* concatenates the values of two columns. A *Split* is the inverse operation. Using a *Divide*, the user can divide the values of a column into two new columns based on a predicate. *Unfold* aggregates several rows, rotating a column to use its values as new columns in the output. *Fold* does the inverse, collapsing a set of columns together copying the rest of the record into several rows. *Unfold* and *Fold* correspond to the PIVOT and UNPIVOT operators in current RDBMSes [1].

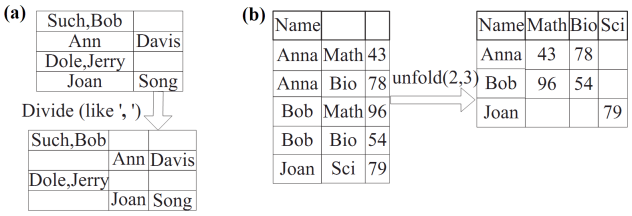


Figure 5.19: *Divide* and *Unfold* operators in Potter's Wheel (adapted from [18]).

Figure 5.19 provides examples of the operators *Divide* and *Unfold*. In (a) the columns of the first column were divided into two columns according to a pattern. In (b) the table was rotated, with the distinct values of the second column transformed in new attributes and the values in column 3 used to populate the new columns.

5.1.3 Considerations

Potter's Wheel takes an approach to data cleaning similar to the one followed in CLEENEX, using data transformations. However, rather than defining a language, this system offers a user interface where a user incrementally constructs a data cleaning program by example actions over a sample of the dirty

[1] http://msdn.microsoft.com/en-US/library/ms177410%28v=SQL.90%29.aspx

data. Note that this system offers absolutely no facilities for user interaction during the actual execution of those cleaning programs. However, Potter's Wheel shows the advantages of using tabular interfaces with user facilities to data manipulation. While there is a clear difference of goals associated with the corresponding user interfaces of Potter's Wheel and CLEENEX, both systems use tabular interfaces reminiscent of financial spreadsheets to convey user intentions to the system. The addition of facilities for data manipulation using simple widgets integrated in table-like interface was inspired by the operators used in Potter's Wheel.

5.2 Conditional Dependencies and Database Repair

Between AJAX and Potter's Wheel, we have reported two different approaches to data cleaning using the same basic idea: a user determines which transformations should be applied to the input data in order to clean it (using either a declarative language or through example) and a system processes that information, applying it during a lengthy automated process. However, other approaches are possible. Another possibility, rather than telling the system "how" to obtain clean data, is to provide constraints that clean data should obey, in a sense, showing "what" is clean data. An automated system can then determine what actions should be applied to the data so that it satisfies these constraints, *i.e.* to generate a repair of the database. However, an input database can have an unbounded number of valid repairs.

[6] introduces a new class of constraints that the authors claim to be better suited to the task of finding a database repair than traditional functional dependencies. In [3], the authors proposed a framework for data cleaning that uses this new class of constraints to generate database repairs and that leverages human feedback over a sample of the generated repair in order to measure the quality of the suggested updates. Another framework based on the concept of database repairs was proposed later in [22].

5.2.1 Conditional Functional Dependencies

Functional Dependencies can be used to identify errors in a data set. Consider a database of customers. If two records share the same address but have different zip codes, then one can deduce that there is a mistake in the data. However, the knowledge of the existence of this error is insufficient to determine which attribute of which record should be changed. Furthermore, if two records in a relation that stores addresses list "New York City" as the value of the attribute *city* and "Canada" as the value of the attribute *country*, enforcing the functional constraint $city \rightarrow country$ will not be enough to detect the errors present in the data.

In order to overcome these deficiencies in Functional Dependencies, the authors of [6] propose *Conditional Functional Dependencies* (CFDs). A CFD is pair $\langle R : X \rightarrow Y, T_b \rangle$, where $R : X \rightarrow Y$ is a traditional

functional dependency between two sets of attributes of a relation R and T_b is a table (a *pattern tableau*) that captures the conditions under which the functional dependency should be enforced.

(a) customers:

	first_name	last_name	city	state	country
t_1:	Jack	Smith	Dresden	Saxony	USA
t_2:	Anja	Meier	Frankfurt	Hesse	Germany
t_3:	Bernd	Weber	Dresden	Saxony	Germany

(b) φ_1=(customers: [city, state] → [country], T_1)

	city	state	country
T_1:	Dresden	-	-
	Dresden	Tennessee	USA
	Dresden	Saxony	Germany

Figure 5.20: Example of a CDF.

Figure 5.20 shows: (a) a relation *customers* that stores customer addresses and (b) an example of a CFD, φ_1. In the patterns of a tableau, the value of an attribute can either be a constant in the domain of that attribute or *"-"*, which identifies a variable. If the value is a constant, then the tuples must have exactly that value in the attribute. If the value is a variable, then tuples must have coinciding values for that attribute. Intuitively, tuples in the relation *customers* that have the value *Dresden* as the city and the same state must have the same country. The tableau provides two examples for pairs $\langle city, country \rangle$. If the state is *Saxony* then the country should be *Germany*. It also specifies that if the state is *Tennessee* then the country should be *USA*. Using φ_1, an automated system can detect an error in tuple t_1 and suggest changing the value of the state to *Tennessee*. It can also suggest changing the country to *Germany* instead.

5.2.2 SEMANDAQ

The authors of [3] propose a framework that makes use of CFDs to generate repairs for a database. Figure 5.21 shows the architecture of that framework, SEMANDAQ. Having a database with dirty data *D* and a set of CFDs Σ, the system can use either the *repairing module* or the *incremental module* to suggest a repair of *D*. Both these modules use the CFDs in Σ to suggest updates to the input tuples; the difference is that, while the *repair module* treats all of *D* as dirty data, the *incremental module* assumes a starting set of clean tuples, suggesting updates for the tuples on the dirty set one at a time. After a repair is automatically produced, it is submitted to the approval of the user. The *sampling module* selects a sample of the generated repair and presents it to the user so that it can be inspected and possibly modified.

The size of the sample is determined by ϵ, the percentage of wrong tuples that the user estimates exist in the original data. According to the number of changes the user performs on the data in the sample

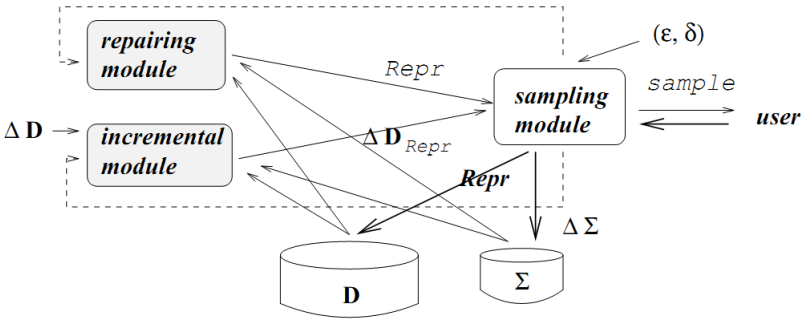


Figure 5.21: SEMANDAQ framework [3].

and δ, a confidence level, the system performs a statistical test to determine the quality of the generated repair. If the quality of the repair is deemed insufficient, the system applies the the updates that were confirmed manually to *D* and invites the user to add CFDs to Σ. It then initiates a new round in order to suggest a new repair. This process is repeated until a repair is accurate enough to pass the statistical test.

5.2.3 Guided Data Repairs

Guided Data Repairs (GDR) [22] is a framework for data cleaning based on the notion of using CFDs to generate data repairs similar to SEMANDAQ. However, there are some key differences between the two frameworks. Instead of proposing a comprehensive repair of the original database (that is, a modification of the original data that satisfies all constraints), GDR suggests a possible list of updates to the input tuples. In each of these updates, only the value of a single attribute is changed. These changes are then submitted to user approval and, if accepted, they are immediately applied to the dataset. This process is repeated until the data is cleaned.

Intuitively, while in SEMANDAQ the feedback of the user is used to rate a repair suggested by the system, in GDR the data is cleaned in an interactive way guided by each successive round of user interaction. The authors consider that this method of suggesting simpler updates that incrementally lead to cleaner data requires less user effort, thus speeding the data cleaning process.

Figure 5.22 show the architecture of the GDR framework. Like in SEMANDAQ, the input of the process is a database *D* of dirty data and a repository of CFDs. These two datasets are used to identify dirty tuples and suggest updates to these tuples. As it was previously stated, in each of these suggested updates only the value of one of the attributes is changed. This set of updates does not provide a comprehensive and coherent repair of *D*; each of the candidate updates brings the database closer to a clean state. Because there are multiple possible repairs for a single constraint violation, it is possible that some of the candidate updates are mutually exclusive. After computing the set of candidate updates, the

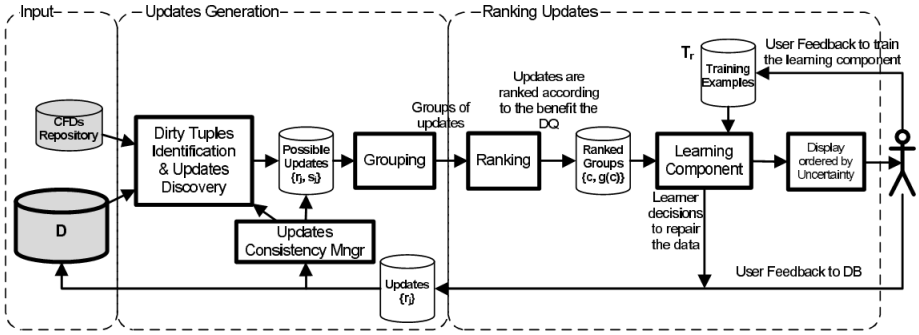


Figure 5.22: *GDR framework [22].*

systems assembles those updates in groups. The grouping criterion is simple: if two candidate updates suggest the assignment of the same value to the same attribute, they are inserted in the same group. In each iteration of the process, the user will provide feedback on the updates of one of these groups. In order to help the user selecting which group of candidate updates she wishes to evaluate, the system ranks the groups according to a criterion that will be explained later in this section. Because the selected group of candidate updates can be very large, only a sample is presented to the user. The feedback of the user over this sample of updates is used to train a machine learning module that will extrapolate the actions of the user on the sample to the rest of the candidate tuples in the selected group. Having determined, using both the human feedback and the decisions of the machine learning module, which updates should be applied to D and which should be rejected, the system updates D and generates a new set of candidate updates. It is also possible to reuse most of the groups of updates that were unused in the previous iteration. However, it is necessary to guarantee the consistency of these updates with the new state of D.

Because only a single group of candidate updates will be submitted to the feedback of the user, it is necessary to ensure that this feedback is applied to the group where it is the most valuable, *i.e.*, a user should provide feedback over the group of updates that, if accepted, cleans more input tuples. To each of the CFDs it is associated a weight that reflects how important that CFD is. In a similar fashion, every time that a candidate update is generated, the system assigns a degree of certainty to its suggestion. In a nutshell, if a group of updates resolves a higher number of more important violations with a higher degree of certainty, then that group of candidate updates will be ranked the highest among all the possible updates in the current iteration. Therefore, it should be a prime candidate to be submitted to user approval. Analogously to an hill-climbing process, if a user consistently provides feedback over the group of candidate updates most likely to provide clean data, the process will converge faster into a completely clean dataset.

5.2.4 Considerations

Both SEMANDAQ and GDR use CFDs to generate repairs for a dirty database. The first difference between these two frameworks is their approach to user feedback. In SEMANDAQ, the role of the human user in the process is clearly secondary. While the user is instrumental in recognizing which CFDs codify the rules for clean data, she is completely removed from the data cleaning process. Automatic reasoning using CFDs is used to identify and solve data quality problems. Only then is the user called upon to accept or reject the new data. If the user rejects it, the system tries to construct a new repairs for the input database. It should be noted that this does not guarantee that, as the system executes more and more iterations, the quality of its output increases. Furthermore, while constraints such as CFDs can be used to detect data quality problems and they can be used to suggest repairs, SEMANDAQ simply makes the direct association between data that conforms with constraints with error-free data.

GDR, on the other hand, assigns higher protagonism to the user. While CFDs are also used to identify tuples with errors, it is up to the user to make modifications based on the suggested updates, retaining complete control over how the data is modified. Because the data cleaning is directed by the user, this framework provide guarantees that the quality of the produced data increases with each iteration of the process. Of course this assumes that the user is providing accurate feedback. In CLEENEX, we also make use of constraints to identify data quality problems. However, we do not assume that any tuple flagged by a QC is automatically dirty and a tuple that was not flagged by a QC is clean. QCs are used to identify unexpected characteristics in a tuple that warrant user inspection. If there are data quality problems and how to solve them is completely up to the user.

While GDR associates progress of data cleaning process – defined as increase of accuracy with each iteration – to user involvement, her possible actions are restricted to modifying output data. The complete reliance in CFDs does not create intermediate steps where a user can inspect a small quantity of data leading to larger gains in accuracy in the end. By contrast, both SEMANDAQ and GDR attempt to limit the amount of user effort required by presenting only samples of the output data.

5.3 User Feedback in Information Extraction

Another field where an automated process can benefit from user feedback is Information Extraction. As is the case in data cleaning, Information Extraction (IE) automated processes tend to be lengthy and prone to errors. [2] proposes a declarative language for the development of IE programs that can incorporate user feedback to increase the quality of the results.

5.3.1 *hlog*

To specify the incorporation of user feedback in an IE program, the authors devised *hlog*, which builds upon their previous work [20] in languages based in *Datalog* to specify IE processes.

```
R1: webPages(p) :- dataSources(url, craw-depth, date),
                   crawl(url, crawl-depth, p)
R2: titles(title, p) :- webPages(p), _
                        extractTitle(p, title)
R3: abstracts(abstract, p):- webPages(p),     _
                             extractAbstract(p, abstract)
R4: talks(title, abstract) :- titles(title, p),
                               abstracts(abstract, p),
                               immBefore(title, abstract)
```

Figure 5.23: Information extraction rules in a hlog program (adapted from [2]).

A program written in *hlog* is divided in two sets of rules. The first set describes the semantics of the IE program and the system uses it to produce a DAG of information extraction and relational operators,which is called *execution plan*. Figure 5.23 presents an example of these rules. This IE program polls a set of data sources (listed in the input relation *dataSources*) and outputs a relation named *talks* that associates the title of a given talk and its abstract. *extractTitle* is an example of a procedural predicate that receives the contents of web pages and populates a table associating the text of a page and the titles extracted from it. The *immBefore* predicate returns a boolean value indicating if a given title immediately precedes a given abstract in a page.

```
R5: dataSourcesForUserFeedback(url, crawl-depth)#spreadsheet-UI
          :- dataSources(url, crawl-depth, date), date >= "01/01/2009"
R6: titlesForUserFeedback(title, p#no-edit)#form-UI
          :- titles(title, p)
R7: talksForUserFeedback(title, abstract)#form-UI
          :- talks(title, abstract)
```

Figure 5.24: User feedback rules in a hlog program (adapted from [2]).

The second set of rules, the *User Feedback Rules*, exemplified in Figure 5.24, specify views over input, intermediate and output data that should be exposed to human inspection and editing through a given user interface. These rules also allow the designer to limit the amount of data presented in each view and to constrain the modifications that a human can perform, *e.g.*, make certain attributes inspectable but not editable. *titlesForUserFeedback* exposes a view over the relation *titles* where a user can inspect both attributes using a form interface but cannot modify the attribute *p*. Using *dataSourcesForUserFeedback*, a user can access the data of the relation that defines the input sources, but only those tuples that have been inserted since *01/01/2009*.

5.3.2 Execution of *hlog* Programs

The execution of a *hlog* program is divided in two phases. The first is the generation of an execution plan and its subsequent evaluation. In this framework, an evaluation plan is a tree with relational tables as the root and leaves. The internal nodes can be either relational or information extraction operators. The edges represent data dependencies between nodes. Figure 5.25 shows the execution plan derived from the rules in Figure 5.23. The execution plan is evaluated from the leaves to the root like a relational execution plan. During this initial evaluation, the views that are mentioned in User Feedback Rules are materialized. For example, the output of the operator *extractTitle* invoked in rule 2 in Figure 5.23 will be materialized because the corresponding relation (*titles*) is mentioned in rule 6 in Figure 5.24.

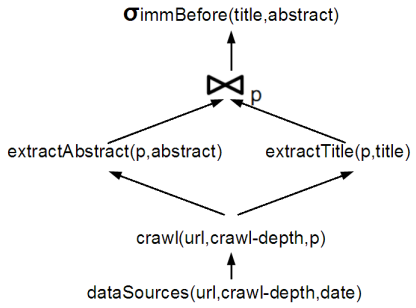


Figure 5.25: Evaluation plan of the example (adapted from [2]).

After this first evaluation, the system enters into a loop where it waits for user feedback. A user can insert, delete or update tuples of the materialized tables through one of the exposed interfaces. When a modification is submitted, the system applies that modification to the corresponding table and propagates it upwards, re-executing the operators until it reaches the root.

5.3.3 Reincorporating User Feedback

The semantics of the execution of a *hlog* program described in Section 5.3.1 shows that whenever a user provides feedback, part of the execution plan is re-evaluated. As such, it is required to store previous feedback in order to avoid erasing it and apply it again at a later time. Furthermore, it is possible to have contradicting user edits over the same data. This mechanism requires that the system finds and stores the provenance of every tuple, that is, for each operator store the information of which input tuples contributed to each output tuple. Note that this also happens in Ajax, although for different reasons. Another point in common is that all tuples must be uniquely identified by a key. This system, rather than propagating identifying information like in Ajax, uses auxiliary tables to associate the keys of the input tuples with the keys of the corresponding output tuples.

Whenever a user provides feedback on the output table of an operator the new data is directly incorporated in the contents of that table. A list that marks the tuples changed by that unit of user input is

associated to that operator. When an operator is reevaluated, both the previous and the current state of its output table are examined. All tuples in the new snapshot that share the same provenance as the marked tuples are deleted. All tuples marked in the old snapshot are copied to the new snapshot, since they represent all the human feedback up to that point.

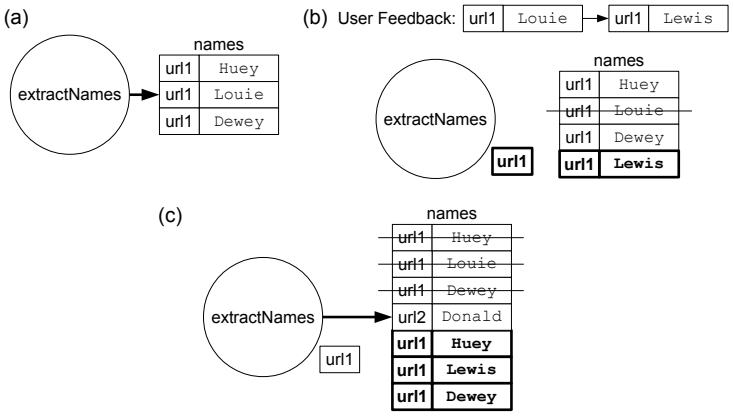


Figure 5.26: Reincorporation of past user feedback.

Figure 5.26 provides an example of the method used for reincorporating past user feedback. The *names* relation associates the url of a web page and names extracted from the text of that page. In (a), the *extractNames* operator recognized three names in *url1*: *Huey*, *Louie* and *Dewey*. In (b), a user determined that the name *Louie* should be *Lewis*. The system stores this new information and marks the tuples with provenance *url1* by storing a record with that value. In (c) the same operator is reevaluated, extracting the same names from *url1* and processing a new page as well. However, because the tuples with provenance *url1* were marked, the tuples in the new snapshot with that provenance are replaced by the corresponding ones from the old snapshot. This way, the user feedback stating that the tuple $\langle url1, Louie \rangle$ should in fact be $\langle url1, Lewis \rangle$ is reincorporated in the most recent version of the data, ensuring that the re-evaluation of the operator *extractNames* does not clobber the previous user feedback.

5.3.4 Considerations

This framework, while targeting the field of Information Extraction, is one of the most important projects for the development of the new user involvement constructs in CLEENEX. The use of a declarative language to specify the data that a user should inspect and to limit the range of actions that a user can undertake is directly inspired by *hlog*. Similarly, the integration of the incorporation of user feedback in the execution of operators was also inspired in this framework. However, we consider the use of MDR instances in CLEENEX to codify user actions as an improvement over the system used in this framework. By promoting user actions to first-class citizens, user feedback is not lost in the data. This turns MDR instances into the groundwork for future developments in CLEENEX, like combining and

reasoning about units of user feedback.

Chapter 6

Conclusions

The ability to write programs to clean dirty datasets is hindered by the typically large size of those datasets and the nature of the quality problems of the data contained therein. While the task of writing such data cleaning programs can be simplified by modeling them as a graph of composed data transformations, data cleaning processes are characterized by a *Debug-Refine-Clean* loop. In each iteration, a human must inspect the output data of the cleaning program, determine if it is sufficiently clean and, if that is not the case, refine the data transformations and restart the process. Eventually, as the number of iterations grows, it is easier to repair the remaining quality problems manually than refining the transformations again.

While the data cleaning cycle does not with dispense the intervention of the user, specialized data cleaning tools in general do not make intermediate data available for user inspection. Therefore, the user must wait for the completion of the execution of the cleaning program and receive as feedback only the final output data. In particular, by not letting the user modify intermediate data, the tool loses opportunities to leverage user feedback in certain intermediate stages where it would be easier for the user to inspect.

We realized the concepts of *Quality Constraints* (QCs) and *Manual Data Repairs* (MDRs) proposed in [9] and modified the existing AJAX framework to incorporate user feedback during the data cleaning process. The evaluation of our implementation of the new framework, called CLEENEX, shows that leveraging the user feedback in intermediate stages of the data cleaning process results in higher accuracy (with a 35% gain in the Precision score and marginal improvements in the Recall score) without requiring excessive user effort. In fact, by using QCs and MDRs during the data cleaning process and then performing manual cleaning of the output the data required the inspection of 27% fewer characters, the insertion of 77% fewer characters and deletion of 64% fewer characters, when compared against only performing the final manual cleaning after executing the same cleaning program without QCs or MDRs.

6.1 Summary of the Contributions

The main contributions of this work are as follows:

Definition of a language for specifying Quality Constraints

Presented in Section 3.2.1, this language is inspired by the AJAX specification language. It allows the designer to specify intermediate points in a data cleaning graph where tuples that do not verify a certain condition are flagged for user inspection.

Definition of a language for specifying Manual Data Repairs

Presented in Section 3.3.1, this language is also inspired by the AJAX specification language. This language allows the designer to associate to certain intermediate points in a data cleaning graph template actions that permit the inspection of data and creation of MDR instances by a user. These MDR instances are units of user feedback that are persisted across executions of the program. When applied, an MDR instance realizes the action specified by the user.

Partial redesign of the AJAX framework

The redesign of the AJAX framework was prompted by the need to incorporate the computation of blamed tuples of QCs and the application of MDR instances during the execution of a data cleaning graph. In Section 3.4.1, we presented the execution of data cleaning graphs over evolving stores of MDR instances.

Partial redesign of the AJAX user interface

In Section 3.4.2 we present the redesign of the AJAX user interface. While this redesign was necessary to incorporate the visualization of blamed tuples, MDR views and the creation of MDR instances, we chose to incorporate the visualization of data and creation of MDR instances in a unified spreadsheet-like interface. In addition, we introduced the ability for the user to perform simple manipulations on how to visualize the data, by sorting columns, filtering rows by cell values and hiding columns.

Development of a prototype implementing the proposed framework

We developed a prototype that implements the CLEENEX framework. In Section 3.2.2 and Section 3.2.3 we detail the implementation of the QC support in the prototype. In Section 3.3.2 and Section 3.3.3, we detail the implementation of the MDR support in the prototype.

Evaluation of the impact of user feedback in a data cleaning process

We conducted an experiment that shows that incorporating user feedback in intermediate stages of a data cleaning process leads to a gain of the accuracy of the output data, without requiring too much user effort. In fact, when considering a follow up manual data cleaning of the output data of a cleaning program, our experiment showed that the aggregate user effort when using QCs and

MDRs is smaller than when performing the same task without QCs or MDRs. This experiment is described in Section 4.2.

Evaluation of the impact for computing blamed tuples

We conducted a second experiment where we compared the time required to compute the set of blamed tuples for each type of QC with the execution of each type of data transformation. With input data of comparable size for each QC and data transformation, our experiment shows that the computation of blamed tuples has a negligible impact when pitted against the execution of data transformations. This experiment is described in Section 4.3.

6.2 Future Work

The current implementation of the prototype realizing the CLEENEX framework cannot perform the incremental implementation of data transformations. This means that, when a data transformation is executed, its logic is applied to all input tuples, rather then being restricted to new input tuples in the current iteration. Incremental execution of data transformations will certainly have an impact in the total runtime of data cleaning programs and should be implemented in agreement with computation of blamed tuples and application of MDR instances. Another limitation of the implementation is that it currently not possible to assign QCs or MDR to input relations of the graph. This is due to the fact that both the computation of blamed tuples and application of MDR instances are dependent on the execution of the data transformation that generates that relation as output.

Another aspect that should be considered in future developments is addressing some perceived limitations of MDRs. The first, simpler aspect that should be considered is decision of assigning a single possible action to each view. In certain conditions, it might be useful to let the user choose among more that one type of action. The second, more important aspect are the limitations inherent to the MDR views. Because a user can only visualize the data that she is editing, she cannot use information from other points in the graph. One possibility would be to extend the concept of updatable view used in MDRs to permit joins. However the user would have to edit the joined data, which might be difficult to understand and the user would be required to have some understanding of the relational model. A much simpler solution would be to let the user visualize more than one table. A viable option would be to specify which auxiliary tables could be inspected as part of the MDR.

While we evaluated the user effort when providing feedback through QCs and MDRs, we did not evaluate the designer effort when producing a data cleaning program that makes use of these constructs. This is important as the need to expose intermediate data for user inspection imposes conditions on the construction of a data cleaning graph. In addition, we also did not evaluate the impact on user effort of the GUI facilities that allow a user to manipulate how the data returned by an MDR view is displayed,

i.e., the ability to sort by column, filter rows by cell value or merge rows by hiding columns. Finally, we did not evaluate how the introduction of new input data between iterations of the data cleaning process affects the application of preexisting MDR instances.

Bibliography

[1] José Barateiro and Helena Galhardas. A survey of data quality tools. *Datenbank-Spektrum*, 14:15–21, 2005.

[2] Xiaoyong Chai, Ba-Quy Vuong, AnHai Doan, and Jeffrey F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *SIGMOD*, 2009.

[3] Gao Cong, Wenfei Fan, Floris Geerts, Xibei Jia, and Shuai Ma. Improving data quality: Consistency and accuracy. In *VLDB*, 2007.

[4] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[5] Francisco Couto, Tiago Grego, Cátia Pesquita, and Paulo Veríssimo. Handling self-citations using google scholar. 13(2), 2009.

[6] Wenfei Fan, Floris Geerts, and Xibei Jia. Conditional functional dependencies for data cleaning. In *ICDE*, 2007.

[7] Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. Ajax: An extensible data cleaning tool. In *SIGMOD Conference*, 2000.

[8] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, 2001.

[9] Helena Galhardas, Antónia Lopes, and Emanuel Santos. Support for user involvement in data cleaning. In *DaWaK*, 2011.

[10] Paul Jaccard. Distribution de la flore alpine dans le bassin des drouces et dans quelques regions voisines. *Bulletin del la Société Vaudoisedes Sciences Naturelles*, pages 241–272, 1901.

[11] Won Kim, Byoung-Ju Choi, Eui-Kyeong Hong, Soo-Kyung Kim, and Doheon Lee. A taxonomy of dirty data. *Data Mining Knowledge Discovery*, 7:81–99, January 2003.

[12] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. 1966.

[13] H. Müller and J.C. Freytag. *Problems, methods, and challenges in comprehensive data cleansing*. Informatik-Berichte // Institut für Informatik, Humboldt Universität zu Berlin. Inst. für Informatik, 2005.

[14] Paulo Oliveira, Fátima Rodrigues, Pedro Rangel Henriques, and Helena Galhardas. A taxonomy of data quality problems. In *2nd Int. Workshop on Data and Information Quality*, pages 219–233, Porto, Portugal, Jun 2005.

[15] David L. Olson and Dursun Delen. *Advanced Data Mining Techniques*. Springer, 2008.

[16] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[17] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 2002.

[18] Vijayshankar Raman and Joseph M. Hellerstein. Potter'swheel: An interactive data cleaning system. In *VLDB*, 2001.

[19] Emanuel Santos and Helena Galhardas. Using argumentation to support the user involvement in data cleaning. In *QBD*, 2011.

[20] Warren Shen, AnHai Doan, Jeffrey F. Naughton, and Raghu Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB*, 2007.

[21] William E. Winkler. The state of record linkage and current research problems. Technical report, Statistical Research Division, U.S. Census Bureau, 1999.

[22] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.

Appendix A

Complete Syntax of the QC Definition Language

⟨*qc-definition-file*⟩ ::= 'define' 'quality' 'contraints' 'as' ⟨*qc-definition*⟩ *

⟨*qc-definition*⟩ ::= ⟨*table-name*⟩ '{' ⟨*constraint-list*⟩ '}'

⟨*table-name*⟩ ::= ⟨*identifier*⟩

⟨*identifier*⟩ ::= ? alphabetic character ? (? alphabetic character ? | ? digit ?) *

⟨*constraint-list*⟩ ::= ⟨*constraint*⟩ ',' ⟨*constraint-list*⟩ | ⟨*constraint*⟩

⟨*constraint*⟩ ::= ⟨*constraint-name*⟩ ':' ⟨*constraint-body*⟩

⟨*constraint-name*⟩ ::= ⟨*identifier*⟩

⟨*constraint-body*⟩ ::= ⟨*not-null-constraint*⟩
| ⟨*unique-constraint*⟩
| ⟨*candidate-key-constraint*⟩
| ⟨*functional-dependency-constraint*⟩
| ⟨*inclusion-dependency-constraint*⟩
| ⟨*predicate-constraint*⟩

⟨*not-null-constraint*⟩ ::= 'not' 'null' '(' ⟨*attribute-list*⟩ ')'

⟨*unique-constraint*⟩ ::= 'unique' '(' ⟨*attribute-list*⟩ ')'

⟨*candidate-key-constraint*⟩ ::= 'candidate' 'key' '(' ⟨*attribute-list*⟩ ')'

⟨*functional-dependency-constraint*⟩ ::= 'functional' 'dependency' '(' ⟨*attribute-list*⟩ ')' 'determines' '(' ⟨*attribute-list*⟩ ')'

⟨*inclusion-dependency-constraint*⟩ ::= 'inclusion' 'dependency' '(' ⟨*attribute-list*⟩ ')' 'in' ⟨*table-name*⟩ '('
⟨*attribute-list*⟩ ')'

⟨*predicate-constraint*⟩ ::= 'predicate' ⟨*function-name*⟩ '(' ⟨*argument-list*⟩ ')'

⟨*constraint-name*⟩ ::= ⟨*identifier*⟩

⟨*attribute-list*⟩ ::= ⟨*attribute*⟩ ',' ⟨*attribute-list*⟩ | ⟨*attribute*⟩

⟨*attribute*⟩ ::= ⟨*identifier*⟩

⟨*function-name*⟩ ::= ⟨*identifier*⟩

⟨*argument-list*⟩ ::= ⟨*argument*⟩ ',' ⟨*argument-list*⟩ | ⟨*argument*⟩

⟨*argument*⟩ ::= ⟨*identifier*⟩
| ? string literal ?
| ? numeric literal ?

Appendix B

Complete Syntax of the MDR Definition Language

⟨*mdr-definition-file*⟩ ::= 'define' 'manual' 'data' 'repairs' 'as' ⟨*mdr-definition*⟩ *

⟨*mdr-definition*⟩ ::= ⟨*table-name*⟩ '{' ⟨*mdr-list*⟩ '}'

⟨*table-name*⟩ ::= ⟨*identifier*⟩

⟨*identifier*⟩ ::= ? alphabetic character ? (? alphabetic character ? | ? digit ?) *

⟨*mdr-list*⟩ ::= ⟨*mdr*⟩ ',' ⟨*mdr-list*⟩ | ⟨*mdr*⟩

⟨*mdr*⟩ ::= ⟨*mdr-name*⟩ ':' ⟨*mdr-body*⟩

⟨*mdr-name*⟩ ::= ⟨*identifier*⟩

⟨*mdr-body*⟩ ::= ⟨*insert-mdr*⟩
| ⟨*delete-mdr*⟩
| ⟨*update-mdr*⟩

⟨*insert-mdr*⟩ ::= 'insert' 'using' ⟨*sql*⟩ 'as' 'view'

⟨*delete-mdr*⟩ ::= 'delete' 'using' ⟨*sql*⟩ 'as' 'view'

⟨*update-mdr*⟩ ::= 'update' '(' ⟨*attribute*⟩ ')' 'using' ⟨*sql*⟩ 'as' 'view'

⟨*sql*⟩ ::= ? SQL query definition ?

⟨*attribute*⟩ ::= ⟨*identifier*⟩

Appendix C

Query Templates for all QC Types

```
--
-- template query for a Not Null QC
--
SELECT relation_pkey, qc_id
FROM relation
WHERE attributes IS NULL

--
-- template query for a Unique QC
--
SELECT relation_pkey, qc_id
FROM relation
  NATURAL JOIN (
    SELECT attributes
    FROM relation
    GROUP BY attributes
    HAVING COUNT(*) > 1 );

--
-- the template query for a Candidate Key QC is omitted because the system
-- substitutes a candidate key constraint over a set of attributes
-- for the conjunction of a not null constraint and a unique constraint
-- over the same attributes
--
```

```sql
--
-- template query for a functional dependency of the form
-- determinant → dependent
--
SELECT relation_pkey, qc_id
FROM relation
  NATURAL JOIN (
    SELECT determinant
    FROM relation
    GROUP BY determinant
    HAVING COUNT(DISTINCT dependent) > 1 );

--
-- template queries for an inclusion dependency of the form
-- relation.lhs ⊆ other_relation.rhs
--
CREATE INDEX index_name ON other_relation ( rhs );

SELECT relation_pkey, qc_id
FROM relation L
WHERE NOT EXISTS (
  SELECT R.rhs
  FROM other_relation R
  WHERE L.lhs = R.lhs );

DROP INDEX index_name;

--
-- template query for a Predicate QC
--
SELECT relation_pkey, qc_id
FROM relation
WHERE expression
```