Support for User Involvement in Data Cleaning

Nkadi Sebastião

nkadi.antonio@tecnico.ulisboa.pt

Abstract. Data cleaning is the process of removing errors and inconsistencies from data in order to improve its quality. The involvement of the user responsible for the execution of a data cleaning program is very important to fix data quality problems that cannot be automatically fixed. Nowadays there is a huge number of data cleaning frameworks due to the ever increasing need of obtaining data with quality. In this work, we focus on two data cleaning frameworks: Cleenex and Pentaho Data Integration (PDI). In this work we develop using Cleenex and Pentaho a data cleaning program aimed at fixing data quality problems contained in a data set describing synthetic australian citizens. From this program, we specified new requirements for Cleenex and conducted a comparative study between both frameworks regarding the support for user involvement capabilities during a data cleaning process. We concluded that the Quality Constraints and MDRs (Manual Data Repairs) are not completely integrated to the data cleaning graph. We also evaluate the quality of our data cleaning program written in Cleenex regarding the accuracy of data it outputs, and the user effort required. We concluded that user feedback during a data cleaning process is beneficial. Finally, we add new features to Cleenex regarding the support for the specification of MDRs on the input tables of the data cleaning graph.

1 Introduction

Data Cleaning is the process of removing errors and inconsistencies from data in order to improve its quality [Rahm and Do (2000)]. As such, this process of removing errors and inconsistencies can be done, both on a simple data set as when integrating multiples data sources.

There exists a huge number of contributions regarding data cleaning. In particular, ETL (Extraction, Transformation, Loading) tools have data cleaning features. Typically, data cleaning programs deal with a huge amount of input data, making it difficult the cleaning process. The quality of the output data of these programs is extremely important, as they are normally used in decision-making processes.

In general, data cleaning programs are modeled as workflows or graph of data transformations. Each of these data transformations encompasses a logical task in the entire process of removing data quality problems from the input data. They are composed, feeding the output of one transformation to the next.

The development of data cleaning programs is an iterative process. In each iteration, a human agent must inspect the output of the data cleaning program
and determine what data quality problems were not sufficiently removed, and refine the transformations in order to bring the input data closer to a clean state. However, in practise it is impossible to write data cleaning programs that outputs completely clean data, as there are data quality problems that cannot be automatically removed. That is the reason why the user feedback i.e., involvement is crucial.

When using ETL and data cleaning tools, intermediate results obtained after individual transformations are typically not available for user inspection or eventual manual correction – the output of a data transformation is directly pipelined into the input of the transformation that follows in the graph. In this work we develop a data cleaning program aimed at fixing data quality problems contained on a data set regarding synthetic australian citizens. We develop it on two different data cleaning frameworks: Cleenex [Galhardas et al., 2011] and Pentaho Data Integration (PDI) [Casters et al., 2010]. The big difference between is that Cleenex was specifically designed for supporting user feedback in data cleaning processes and PDI was not. From this we present the Cleenex weaknesses and new requirements for it. We also were able to conduct a comparative study between both frameworks.

The remainder of this paper is organized as follows. Section 2 presents an overview of Cleenex, and Section 3 presents an overview of PDI, both of them presenting some background for this work. Section 4 details how our case study was conducted and presents the main contributions of this work. Section 5 presents the data cleaning program evaluation, the new requirements for Cleenex and the comparative study. Section 6 discusses the most relevant related works. Section 7 presents our conclusions and future work.

2 Cleenex

The Cleenex data cleaning framework supports the specification of data cleaning programs through a declarative language very similar to SQL and inspired by it. Most of the concepts of this framework were inherited from Ajax [Galhardas et al., 2001, 2000] by adding to it two new constructs that permit the user involvement in data cleaning processes: Quality Constraints (or simply QCs) and Manual Data Repairs (or simply MDRs). A program is modeled as a Directed Acyclic Graph (DAG) of data transformations and relations.

There are two human roles on Cleenex: the Designer and the User. The designer uses the data cleaning graph declarative language to construct a graph of data transformation, and also identifies the relations of the graph where user feedback can be given by defining appropriate QCs and MDRs. Each transformation can have one or more input relations and one or more output relations. All intermediate data are saved on a RDBMS. There are five types of data transformations: Mapping, a one-to-may mapping between an input tuple and the corresponding output tuples, View, an SQL query augmented with some integrity checking on its result, Matching, an approximate join, Clustering, that groups the tuples of an input relation having similar properties into sets
of tuples using a given clustering algorithm, **Merging**, that partitions an input relation according to a given criteria and chooses a representative tuple for each partition.

The user is the responsible for executing data cleaning programs previously developed by a designer. Using the CLEENEX graphical interface, a user can inspect input and 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. Whenever the feedback is needed, the user can edit manually the data. Note that the refinements that the user can apply on the data are not arbitrary. The user can only refine or edit data previously allowed by the designer of the data cleaning program.

### 2.1 Quality Constraints

As the name itself suggests, Quality Constraints are conditions added to a data cleaning graph to identify the points of the graph, *i.e.*, relations where the intermediate result must obey certain conditions, and its main purpose is to signalize suspicious data, *i.e.*, data that where not automatically transformed as intended, and, therefore must be analysed by the user. CLEENEX features five types of QCs: i) **Unique**, 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 compose that relation, ii) **Functional Dependency**, used to enforce the existence of a functional dependency between two set of attributes in the target relation, *i.e.*, if two tuples have equal values for a set of attributes (called **determinant**) then they must also agree in the values assigned to another set of attributes (called **dependent**), iii) **Inclusion Dependency**, that 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, iv) **Predicate**, that enforces that each tuple of a relation obey a certain domain property, identified by a predicate that returns either “true” or “false”, v) **Candidate Key**, that enforces that a set of attributes in a relation be simultaneously **not null** and **unique**.

---

**QC:** not null (year)

<table>
<thead>
<tr>
<th>Title</th>
<th>Director</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>Terminal Velocity</td>
<td>Deran Sarafian</td>
<td>2013</td>
</tr>
<tr>
<td>Against All Odds</td>
<td>Taylor Hackford</td>
<td>1984</td>
</tr>
<tr>
<td>After Earth</td>
<td>M. Night Shyamalan</td>
<td>—</td>
</tr>
<tr>
<td>The Equalizer</td>
<td>Antoine Fuqua</td>
<td>2014</td>
</tr>
<tr>
<td>Another Earth</td>
<td>Mike Cahill</td>
<td></td>
</tr>
</tbody>
</table>

(a) Blamed tuples.

**DEFINE MANUAL DATA REPAIRS AS**

```markdown
Film_Releases {
  InsertMissingYear: INSERT USING
  SELECT *
  FROM BLAMED(QC)
  AS VIEW
}
```

(b) MDR definition.

---

Fig. 1: Example of QC definition, blamed tuples and MDR definition.

When defining a QC the designer must indicate the node of the DAG where it will be enforced, *i.e.*, any relation belonging to the data cleaning graph. He
must also indicate the condition(s) that the tuples must obey. A tuple violating at least one of these conditions is called a blamed tuple. Figure 1a shows a QC enforcing a condition that the Year must not be null and its corresponding blamed tuples.

2.2 Manual Data Repairs

A Manual Data Repair (MDR) is a mechanism used to limit and guide the user when providing feedback. As QCs, MDRs are added to the data cleaning graph by the designer as well.

A MDR is composed by an Updatable View and an Action. The purpose of the updatable view is to enable the designer to limit the quantity of information that will be shown to the user. This is achieved by projecting only the values of some attributes, or selecting only some tuples of a relation. For example, it is possible to limit the amount of information that is shown to the user by selecting merely the blamed tuples related to one or more QCs. Figure 1b provides an example of MDR definition. We define an MDR named InsertMissingYear that aims to allow the user to insert the missing years on the relation named Film_Releases, shown in Figure 1a. The updatable view is, obviously, the set of the blamed tuples related to the QC named QC, also presented in Figure 1a.

The MDR is merely a template for an action that can be applied to a relation on the data cleaning graph. The concrete actions are designated by MDR Instances. Each one of them represents a feedback unit that can be applied several times during the data cleaning process. They also can be persistently saved with the intermediate data so that the feedback is not lost if the data cleaning process needs to be interrupted and resumed later.

There are three types of actions user can perform over an updatable view: insert, update and delete. By restricting the choices users can make, it becomes much easier to provide feedback. Moreover, MDR Instances can be saved and reapplied on subsequent executions of the data cleaning program.

3 PDI

PDI stands for Pentaho Data Integration. Also called Kettle, it is the component of Pentaho – software developed for Business Intelligence, including ETL, OLAP and Data Mining capabilities – responsible for ETL processes. It has a set of features that makes it possible the transference and the transformation of data coming from heterogeneous data sources, typically databases or files on many formats. Furthermore, it is cross-platform and has a graphical interface that has “drag and drop” data cleaning operators capabilities, reducing as much as possible the designer effort, and consequently increasing productivity. We have chosen PDI to serve as a basis for comparison with respect to Cleenex for being an ETL tool that has been widely used, and because of its good knowledge base.

One of the most fundamental concepts in Kettle is transformation. It handles the manipulation of rows or data, and consists of one or more steps that perform
core ETL work such as reading data from files, filtering out rows, data cleaning, or loading data into a database.

The steps are the building blocks of a transformation and are represented as icons. Inside a transformation they are connected by transformation hops. The hops define a one-way channel that allows data to flow between the steps that are connected by the hop. In Kettle, the unit of data is the row, and a data flow is the movement of rows from one step to another step.

A transformation hop, represented by an arrow between two steps, defines the data path between the steps. The hop also represents a row buffer called a row set between two steps, which size can be defined in the transformation settings. When a row set is full, the step that writes rows halts if there is room. When a row set is empty, the step doing the reading will wait a bit until rows are available again.

Transformations can be combined using jobs. A job is a construction used to coordinate ETL activities, such as: i) Definition of the transformations execution sequence, ii) Checking the existence of files and database relations, iii) File management operations, such as sending, receiving, copying, and deleting, and iv) Sending success or failure notifications of an operation by email, etc.

The building blocks of jobs are the job entries. They make it possible the invocation of a transformation previously defined inside a job. Just like the transformation steps, they are also graphically represented as icons. Because they are executed sequentially, a starting point defined by the designer is required. This starting point represents a special job entry called Start. Job entries can have an execution path defined through job hops. Job hops are represented in the form of the link between two job entries.

4 Case Study

We developed a data cleaning program aimed at fixing data quality problems contained on a dataset about fictitious Australian citizens using both Cleenex and PDI frameworks. This dataset is composed by two tables: Citizens(id, name, street_number, address_1, address_2, suburb, postcode, state, date_of_birth, age, phone_number, soc_sec_id) and States(id, name, short_name, area, telephone_code, population, suburbs).

The attributes of the Citizens table have the following meanings: id (a unique identifier of a tuple), name (the name of the citizen), street_number (the number of the street), address_1 (the name of the street), address_2 (the number of apartment), suburb (the suburb), postcode (the postcode), state (the state), date_of_birth (the date of birth), age (the age), phone_number (the phone number), and soc_sec_id (the social security number).

With regard to the table States, the meaning of the attributes are the following: id (a unique identifier of a state), name (the full name of the state), short_name (the short name of the state), area (the area of the state), telephone_code (the telephone code), population (the population), and suburbs (the complete list of the existing suburbs and postcodes of a state).
The content of the table Citizens was automatically generated using FEBRL [Christen (2007)], a data cleaning framework with dirty data generation capabilities, such as duplicate data and misspellings. It contains 700 tuples distributed in three groups: i) 149 non-dirty tuples (21.2%); ii) 500 dirty tuples (duplicates); and iii) 51 dirty tuples (7.2%). The table States plays a role of a dictionary, where accurate information about states, suburbs and postcodes can be found.

![Fig. 2: Activities of the data cleaning program.](image)

The developed data cleaning program is depicted in Figure 2 using UML activities diagram. Its main objective is to remove the duplicate tuples and misspellings contained in the Citizens table, and select only citizens belonging to australians states named ‘Queensland’, ‘Victoria’ and ‘South Australia’. The activities between 1 and 6 are intended to data normalization, and their main goal is to maximize the effectiveness of the duplicate detection and removal process, materialized in Activity 7. Subsequently, in Activity 8, we proceed the soc_sec_id repair and data formatting.

4.1 Implementation on CLEENEX

In this Section we discuss how the citizen cleaning program was implemented on CLEENEX. For the sake of simplicity, we only illustrate the implementation of the Activity 1, depicted in Figure 3.

The Activity 1 aims at repairing state values. We divided the tuples of the table Citizens in two groups: i) those having not null value on the attribute state (table CitizensHavingNotNullStates), and ii) those having (table CitizensHavingNullStates). In the case of the tuples of the table CitizensHavingNotNullStates, we performed an approximate lookup in order to confirm that the state of a citizen really exists on dictionary. If a state of a citizen does not exist on dictionary, we request user feedback on the table HighestStateSimilaritySelected through a QC and a MDR.

The tuples of the table CitizensHavingNullStates were in its turn divided in two groups: i) those having not null value on the attribute phone_number (table CitizensHavingNotNullPhoneNumber), and ii) those having (table CitizensHavingNullPhoneNumber). In the case of the tuples of
the table CitizensHavingNotNullPhoneNumber, we extracted the telephone code from the phone_number and used it to infer the corresponding state. In the case of the tuples of the table CitizensHavingNullPhoneNumber, we simply present them to the user for feedback through MDR as we do not have enough information to infer the state. Finally, using a View transformation applied to the tables CorrectedStatesView and DiscardedNullStates, we
perform the union of the created groups obtaining the table RepairedStates. Note that we only present a tuple to the user for feedback when we do not have enough information to automatically infer a value.

4.2 Implementation on PDI

In order to keep the data cleaning program implementation on PDI similar to Cleenex, we stipulated that every PDI transformation might keep its results on a database table. Figure 4 presents the implementation of activity 1 on PDI. It starts with a special job entry Start and ends with another special job entry Success. In between, there are job entries either executing defined transformation or waiting for user feedback.

First of all, we create tables needed by the transformation to store the results, using the job entry SQL. The most important job entries are described below:

- **OpeningStateRepair**: It divides the tuples of the table Citizens in four groups materialized in database tables: i) those having null value of state and not null value of phone_number, and these were selected through the step table input named CitizensHavingNullStatesAnd-NotNullPhoneNumbers, ii) those having null value of both state and phone_number attributes, and these were selected through the step table input named CitizensHavingNullStateAndNullPhoneNumber. The table input named CitizensHavingNotNullStates just selects the tuples of the table Citizens having a null value of state. These tuples are passed to the step Filter rows which in its turn divides them in iii) tuples also having not null value of phone_number, and iv) tuples not having;

- **Wait for FeedbackOnBlamedStates**: It blocks the data cleaning program until a CSV file named FeedBackOnBlamedStates.csv with schema (id, state) is provided. Each filled data line of this file represents a unit of feedback about non-existing states. To provide feedback the user should see the tuples of the group iii);

- **BlamedStatesRepair**: It incorporates the feedback obtained in Wait for FeedbackOnBlamedStates job entry. The tuples of the group iii) are updated accordingly;

- **Wait for FeedbackOnNullStates**: It blocks the data cleaning program until a CSV file named FeedBackOnNullStates.csv with schema (id, state) is provided. Each filled data line of this file represents a unit of feedback about tuples having null values in state. To provide feedback the user should see the tuples of the group ii);

- **NullStatesRepair**: It incorporates the feedback obtained in Wait for FeedbackOnNullStates job entry. The tuples of the group ii) are updated accordingly;

- **RepairedStates**: It puts together the tuples of the four groups i), ii), iii) and iv) using the SQL union operator.
Note that PDI was not intentionally designed for supporting user feedback. Our methodology to incorporate user feedback goes through the division of the tuples in groups according to a given property. There are groups of tuples which data quality problems can be automatically fixed, or that do not have data.
quality problems at all. The groups of tuples which data quality problems cannot be automatically solved should be analysed by the user. When the feedback is needed, the cleaning program blocks and the user must provide a CSV file with feedback. In this CSV file, the user must specify the id of the tuples that the feedback is being given and the values to be inserted or modified. After that, the program incorporates the feedback by updating, inserting or deleting tuples on the table that has the data quality problems in question, and then can continue its execution.

Our approach for supporting user feedback on PDI has a big drawback. It lies in the fact that, for instance, when the goal of feedback is the update of tuples, we incorporate this feedback through the PDI step Insert/Update. As the name of the step suggests, it inserts a tuple if the value of the key exists on the target table and otherwise, it simply update the specified values. If the user makes mistake when identifying the id of the tuples that will benefit from feedback, erroneous information can be inserted, and consequently the data accuracy of the cleaning program output may get harmed.

5 Evaluation

We evaluated our data cleaning program designed on CLEENEX regarding the accuracy of output data and the user effort when providing feedback. We performed three tasks: i) Completely manual cleaning of the Citizens table, obtaining $T_1$, ii) Automatic cleaning of the Citizens table using the implementation of the cleaning program on CLEENEX without any kind of user intervention, obtaining $T_2$, iii) Automatic cleaning of the Citizens table using the implementation of the cleaning program on Cleenex incorporating user feedback with the creation of MDR Instances, obtaining $T_3$. Figure 5a) shows that, due to leveraging of the user feedback using QCs and MDRs, $T_3$ shows a 26% gain on Precision score in relation to $T_2$.

For measuring user feedback, we introduced two new tasks: $T_2'$ and $T_3'$. $T_2'$ and $T_3'$ involve the manual cleaning of the output of $T_2$ and $T_3$, respectively, until we obtain the same data present in $T_1$. Figure 5b) shows a reduction on user feedback effort. In particular, The total user effort on $T_3'$, both during and after the automated cleaning phase requires the inspection of 44% fewer characters, the insertion of 90% fewer characters and the deletion of 77% fewer characters in regard to $T_2'$. The 25% increase in updated characters in $T_3'$ with respect to $T_2'$ is explained by the fact that most of our MDRs were concerned to update tuples.

Figure 5 shows the MDR instances statistics on our cleaning program implemented on CLEENEX. Note that 86% of updated characters where performed on intermediate tables of the cleaning graph whereas we only insert and delete tuples on the output table of the graph, just after the duplicate detection and removal process.
Statistics of MDR instances:

<table>
<thead>
<tr>
<th>Activity</th>
<th>Action</th>
<th>Tups</th>
<th>Chars</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Insp.</td>
<td>29</td>
<td>217</td>
</tr>
<tr>
<td></td>
<td>Upd.</td>
<td>190</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Insp.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Upd.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Insp.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Upd.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>Insp.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Upd.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>Insp.</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Upd.</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

When creating MDR Instances

<table>
<thead>
<tr>
<th>Action</th>
<th>Incidence</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
</tbody>
</table>

User Effort

<table>
<thead>
<tr>
<th>Action</th>
<th>Incidence</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
</tbody>
</table>

TP 115 91 114
TN 585 561 585
FN 24 1
FP 0

Precision 0.7913
Recall 0.7913

(a) Accuracy.

Fig. 5: Accuracy and User effort results.

5.1 CLEENEX weaknesses

Due to the development of the data cleaning program depicted in Section 4, we found out the following weaknesses of CLEENEX:

Operators between relations
It is not possible to perform the union or intersection of the content of two tables.

Specification of output relations of a transformation
When specifying an output table of a given transformation, it is not possible to specify the data types of its attributes.

Logical operators precedence and SQL language operators
CLEENEX does not support logical operators precedente, most importantly on the WHERE clause. Also, although a View transformation is said to be a mere SQL query, it does not allow the use of SQL operators as like.

Machine learning
CLEENEX does not learn from QCs. A QC can give important information about what kind of values are acceptable to a given attribute. When the user provides a MDR instance, CLEENEX does not verify if the values of each attribute refered in the MDR instances are in harmony with the QCs that has an impact to them.

Lack of MDR Instance management features
It is not possible from the graphical interface of CLEENEX to see any information about existing MDR instances. With MDR management features, it would also be possible to delete and update existing MDR instances.
5.2 **CLEENEX and PDI – A Comparative Study**

This Section presents the results of the comparative study we have conducted between CLEENEX and PDI. We compared them according to the following parameters:

**Data cleaning program representation**

A data cleaning program in CLEENEX is written using a declarative language very similar to SQL, explained in Section 2. In PDI, as the designer specifies a program using the graphical interface, XML code is internally generated.

**Customization (Use of another programming languages)**

Both CLEENEX and PDI allow the use of Java code to extend the functionalities of a cleaning program. The main difference is that PDI does not require 100% Java [Meadows et al., (2013)], i.e., the programmer has not to write a full Java class declaration (only methods, constructors and imports) whereas in CLEENEX a full class declaration must be provided. Furthermore, PDI also allows the use of pure SQL code for getting data from a database through Table input step, whereas CLEENEX does not.

**User involvement**

CLEENEX was designed to support user feedback by defining appropriate QCs and MDRs. PDI was not, but by using the Wait for file job entry followed by a mechanism to read the content of the CSV file containing user feedback, it becomes possible as explained in Section 4.2.

**Usability**

CLEENEX has only a user interface for executing cleaning programs, not for designing them. PDI has a intuitive graphical interface with “drag and drop” capabilities that can be used for both developing and executing programs.

**Duplicate detection and removal process**

In CLEENEX this process is implemented through a Matching-Clustering-Merging sequence of transformations. The Matching transformation performs a cartesian product among tuples of the desired table and also calculates values of similarity for each tuple combination; the Clustering transformation groups the very similar tuples into the same cluster; finally, the Merging transformation chooses a representative tuple for each cluster generated by the clustering algorithm. Note that more than one attribute can be used in the computation of similarity as it is done through an external function. In PDI, this process is inflexible, as the step that performs this operation is black box. It only allows the selection of one attribute that will be used in the duplicate detection and removal process.

**Mapping between input and output tuples**

In CLEENEX it is done by the Mapping operator. In PDI, it can be done by one of these steps: i) Split fields (only one-to-one mapping between input and output tuples), ii) Split fields to rows (one-to-many mapping), and iii) Table input (a pure SQL query).

**SQL queries**

In CLEENEX it is done through the View transformation whereas on PDI one can use the Table input step.
Cartesian products between relations
In Cleenex it is implemented through the Matching transformation. In PDI, the steps Join rows or Table input can be used.

Use of clustering algorithms
In Cleenex, the Clustering transformation is flexible as it permits the use of any defined clustering algorithms. In PDI, there is the Closure generator step that only uses the transitive closure algorithm.

Merging groups of tuples having similar properties
In Cleenex it is implemented by the Merging operator whereas in PDI this operation is not available.

6 Related Work

In this section we present the most relevant work on data cleaning, information extraction and data integration that leverage user feedback.

Potter’s Wheel [Raman and Hellerstein (2001)] is a data cleaning interactive system. It automatically detects data quality problems and presents a sample data to the user. While the user applies transformations to the data, these are stored and the system produces a program. This program can be executed over the entire dataset. This work shows the importance of spreadsheet-like interfaces to display data and learning user intentions. It differs from Cleenex in that on Potter’s Wheel feedback is used to produce a data cleaning program whereas on Cleenex the feedback is used to improve the output results.

[Belhajjame et al., (2011, 2013)] proposes an approach that incrementally improve a dataspace based on user feedback. In this proposal, the feedback is given on the result of a query posed by a user. Based on this feedback the system constructs an internal representation of the user needs. This approach is similar to the one implemented on Q System [Talukdar et al., (2008)]. It makes it possible the system to learn the preferences of the users. Consequently, the system uses the information obtained from this feedback to better process queries posed by users to the system, so that its output can meet user requirements. In this proposal it is not possible to limit the amount of information seen by the user whereas on Cleenex, using MDR features, it is.

[Chai et al., (2009)] presents a framework in the domain of Information Extraction (IE), where it was identified that user feedback can improve automated processes, that uses rules in a declarative language, hlog, to specify IE programs and to present intermediate data to be inspected and modified by users. While having been developed for a different domain, this work is very similar to Cleenex. In it, it is also possible by the designer to limit on the actions of the users and the amount of data they can see.

7 Conclusions and Future Work

The overriding objective of this work was the development of a data cleaning program using both Cleenex and PDI frameworks. A particular emphasis was
given to the implementation of the program on Cleenex. This emphasis derived from the need to know its strengths and weaknesses, in order to infer new requirements for it. On the other hand, the development of that program in the both frameworks provided us a basis for the preparation of a comparative study between the two frameworks.

We presented new requirements for Cleenex. Among them, we mentioned the lack of operations of intersection and difference of relations, inexistent features that allow the specification of attributes data types in the output relation(s) of transformations, lack of precedence of logical operators and the lack of support for SQL operators such as `like`.

It would be interesting the development of a “drag and drop” graphical interface for Cleenex, so that it can help the designer on the development of data cleaning programs. It is supposed that this kind of interfaces internally generates an equivalent code of the cleaning program as the designer specifies the graph. This new interface will oblige the designer to focus more on the semantic of the transformations, QC's and MDRs than in the syntax of the program. It would also be interesting to study the impact of defining MDRs in the intermediary relations of the cleaning graph. It was only studied the impact of MDRs on the entire cleaning graph.

References


X. Chai, B.-Q. Vuong, A. Doan, and J. F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In SIGMOD Conference, pages 87–100, 2009.