Support for User Interaction in a Data Cleaning Process

João Lobato dos Santos Dias

joao.lobato@ist.utl.pt

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 [7], 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 implementation 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 [7]. Moreover, CLEENEX is instrumental for performing a thorough evaluation of this approach.

1 Introduction

The process that can be used to identify and repair the data quality problems of a dataset is called *data cleaning*. 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.

We realized the concepts of *data quality constraints* and *manual data repairs* that were proposed in proposed in [7] in order to involve the user in the data cleaning process in a framework called CLEENEX. We based our work on the existing data cleaning framework AJAX and developed a prototype implementing those concepts. We performed the evaluation of our prototype with regard to accuracy *vs* user effort, obtaining results showing a 34% gain in the precision of output data while decreasing total user effort in relation to the manual cleaning the output data of the same data cleaning process without QCs or MDRs. We also obtained results showing that the new constructs have a negligible impact in the total running time of a data cleaning process.

The rest of this paper is organized as follows: Section 2 presents some background of our work, providing an overview of AJAX and of the approach to user involvement in data cleaning in which our work relies. Section 3 details CLEENEX, the framework that realizes the support for user interaction in data cleaning. Section 4 details the prototype implementing CLEENEX. Section 5 discusses the experimental evaluation of our prototype. Section 6 discusses the most relevant related work. Section 7 presents our conclusions and future work.

2 Background

The AJAX framework [5, 6] 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 AJAX framework assigns two roles to the human operator: the *Designer* and the *User*. The designer specifies the DAG that constitutes the cleaning program using a declarative language inspired in SQL. The user runs the data cleaning process.

The designer uses the data cleaning graph specification language to construct a graph of data transformations. Each transformation can have one or more input relations and one or more output relations. All intermediate data is materialized in a RDBMS. There are five types of data transformation: **Map**, a one-to-many mapping between an input tuple and the corresponding output tuples, **Match**, an approximate join, **Cluster**, that groups the tuples of an input relation,*e.g.* pairs of duplicate records, into sets of tuples using a given clustering algorithm, **Merge**, that partitions an input relation according to a given criteria and chooses a representative for each partition and **View**, an SQL query.

The user executes the data cleaning process. The framework offers a Graphical User Interface (GUI). 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. 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.

While the GUI of AJAX allows the user to inspect and modify intermediate data in the data cleaning process, in practice it blurs the distinction between designer and user. It forces the user to require skills in data cleaning and knowledge about the domain of the program. This limits the number of data consumers whose knowledge can be leveraged to improve the results of the data cleaning process. Another drawback of AJAX is that any user feedback is lost if the data cleaning graph is re-executed.

[7] 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*.

Quality Constraints (QCs) are added to a data cleaning graph 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. A QC is composed by the relation in the data cleaning graph where it is enforced and the condition that the tuples of that relation should obey. The set of tuples that violate a QC is called set of *blamed tuples* of that QC.

Manual Data Repairs (MDRs) are associated to relations in the cleaning graph. They are used to limit and guide the actions that a user can perform during the data cleaning process. An MDR is composed of an updatable View and an Action. The *View* is used to limit the amount of information that the user must analyze, using projections or selections. In particular, the view of an MDR can be defined to show only the *blamed tuples* of a QC. The *Action* is either an insert, delete or update operation that can be applied to the 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 [10].

3 CLEENEX

In this section we present the realization of the data cleaning graph concept as proposed in [7], named CLEENEX. CLEENEX adds support for Quality Constraints (QCs) and Manual Data Repairs (MDRs), addressing the shortcomings of AJAX that we identified and extends the execution semantics of AJAX to incorporate these new constructs.

3.1 Support for Quality Constraints

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** – 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.
- **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*. This can be useful to identify errors in the data.
- **Inclusion Dependency** – 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.
- **Predicate** – Enforces that each tuple of a relation obey a certain domain property, identified by a predicate that returns either "true" or "false". A notable element of this family is the *not null* predicate, enforcing that the values of a set of attributes in a tuple be different than null.
- **Candidate Key** – Enforces that a set of attributes in a relation be simultaneously *not null* and *unique*.

We developed a language used to define the QCs, inspired by the AJAX specification language.

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.

In order to compute the set of *blamed tuples* for each QC, CLEENEX uses parameterized SQL queries. Note that these queries select those tuples of the relation that do **not** satisfy the constraint.

3.2 Support for Manual Data Repairs

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.

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. 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.

Figure 1 shows an example of the definition of QCs and MDRs in CLEENEX. There are two QCs and two MDRs defined over the relation *addresses*: $qc1$ marks the tuples without *state* information and $mdr1$ can be used to complete that information; $qc2$ marks the tuples that do not obey the dependency $(city, state) \rightarrow country$ and $mdr2$ can be used to resolve the conflict by deleting one or more of the blamed tuples.

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

```
Addresses {
  qc1: NOT NULL (state),
  qc2: FUNCTIONAL DEPENDENCY (city, state) DETERMINES (country)
}
Addresses {
  mdr1: UPDATE(state) USING SELECT * FROM blamed(qc1) AS VIEW,
  mdr2: DELETE USING SELECT city, state, country FROM blamed(qc2) AS VIEW
}
```

Fig. 1. QC and MDR definition in CLEENEX.

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.

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.

The views of MDRs are created in RDBMS 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.

More details about the QC and MDR definition languages can be found in [3].

3.3 Execution of a Data Cleaning Graph with QCs and MDRs

The execution semantics of AJAX were changed to incorporate the computation of blamed tuples of QCs and the application of MDR instances. 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 2. 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*.

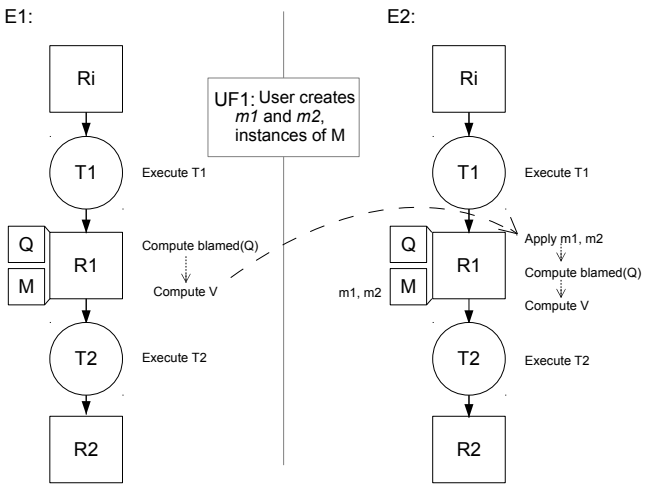

Fig. 2. Overview of execution of a data cleaning graph in CLEENEX

Figure 2 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.

4 Prototype

We developed a prototype implementing the proposed framework, based on the original AJAX prototype. We added two new components, *QC Manager* and *MDR Manager* that, respectively, encapsulate the support for QCs and MDRs. We also performed some modifications in existing components to accomodate the new constructs. In this document we give special emphasis to the changes in the GUI.

4.1 QC Manager

The new *QC Manager* is responsible for the support of QCs in the CLEENEX prototype. It includes the parser for the QC definition language. Internally, each type of QC is implemented by a subclass of the abstract class *Constraint*. Each of this subclasses implements a method called *generateCode* that, using templates, generates the SQL query to compute the set of blamed tuples of the QC. There are also the optional methods *setup* and *teardown* that can be used, for example, to construct indeces to speed up query execution. Instances of *Constraint* are stored in memory and the code to compute the blamed tuples of a QC associated to a given relations are included in the transformation that populates that relation.

4.2 MDR Manager

The new *MDR Manager* is responsible for the support of QCs in the CLEENEX prototype. It includes the parser for the MDR definition language and a subcomponent that manages the persistence of MDR instances. MDRs and MDR instances are represented internally has two similar class hierarchies that encompass each type of action available. Each instance of a subclass of MDR is responsible for the creation of MDR instances of the corresponding type, using the *Abstract Factory* design pattern. The implementation uses a similar code generation scheme to the one used for QCs. However, while the code for computing blamed tuples of QC depends only on the QC defined by the designer, the code that applies a given MDR instance depends on the MDR (known at compile time) and on the target tuple, defined by the user when she creates the MDR instance. Therefore, code generation is split between the instance of the MDR and the instance of the MDR instance.

4.3 GUI

We modified the original AJAX user interface to let the user inspect the data returned by views of MDRs and to create MDR instances. To display data returned by views, the GUI uses a spreadsheet-like interface with embedded widgets used to create MDR instances, *e.g.*, when a user inspects the data returned by the view of an MDR with action "delete", each row in the displayed table has a button that allows the creation of an MDR instance to delete the corresponding tuple. We added an additional mechanism to let the user manipulate how she views that data. The manipulations that the user can perform are: sort rows, filter rows by cell value and hide columns, which results in collapsing multiple rows with equal data into a single row, allowing the user to create multiple MDR instances with a single action.

5 Experimental Validation

We performed two experiments 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.

5.1 Accuracy *vs* User Effort

We conducted an experiment to evaluate the gain in accuracy of the output results of a data cleaning process that incorporates user feedback in relation to the effort required of the user that provides feedback. We developed a data cleaning program to clean a subset of the data used in CIDS project [2]. The goal of the process is to eliminate duplicate publications and present only publications authored by team members. We performed three tasks: *T1*, manual cleaning of the dataset, *T2*, automatic cleaning of the dataset using our program without QCs or MDRs and *T3*, automatic cleaning of the dataset incorporating user feedback. We added tasks *T2'* and *T3'*, that result of the manual cleaning of the outputs of *T2* and *T3*, respectively.

To measure the accuracy of results we treat each task has a classification problem, where a duplicate publication correctly identified as such is classified as a *True Positive* (TP), a publication present in the output but that was not authored by a team member is classified as a *False Negative* (TP), *etc.* We interpret the results using the metrics *Precision* and *Recall* [8].

We measure user effort by counting the tuples and characters that the user has to inspect as well as those that are inserted, deleted or updated, depending on user action. Figure 3 shows our results. Table (a) shows a 34% gain in the precision of output data of *T3* over *T2* while table (b) shows the total user effort, both during and after the automated cleaning phase in $T3'$ requires the inspection of 27% fewer characters, the insertion of 77% fewer characters and the deletion of 64% fewer characters in regard to $T2'$. We can conclude that the gain in accuracy of output data while requiring less user effort than simply cleaning the output data by hand.

(a)

	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

(b)

		T1	T2'	T3'
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

Fig. 3. Accuracy *vs* user effort in the publications experiment.

5.2 Impact of Computation of Blamed Tuples

We also conducted an experiment to evaluate the time spent by our prototype in the computation of blamed tuples for QCs. We constructed an artificial dataset that can be grown by tweaking an integer parameter and evaluated each possible type of data transformation and each possible type of QC using increasingly large volumes of input data.

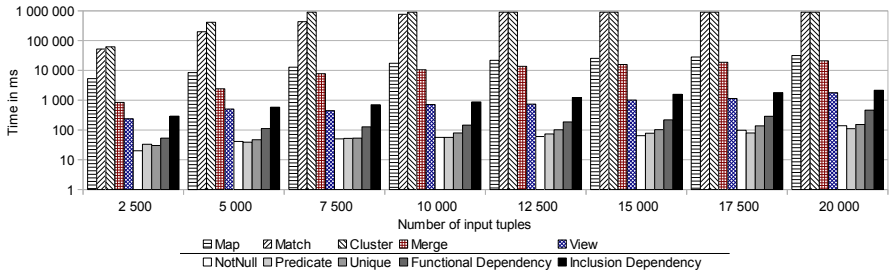

Fig. 4. Computation time for each type of transformation and QC.

Figure 4 presents our results. 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. Also note that we cut off the execution of *Cluster* and *Match* at the 15 minute mark (900000ms). The chart shows all transformations other than *View* take more time than any QC, especially *Clusters* and *Matches*. We can conclude that impact of computing blamed tuples in the total running time of a data cleaning program is negligible.

6 Related Work

Data cleaning is characterized by automated processes that produces output data that data must be appraised to determine if it possesses sufficient quality to serve the interests of the data consumers. This appraisal is sufficiently complicated

that it must be executed by human operators. In general, the user cannot provide feedback over intermediate data because the data process is treated like a black box.

Potter's Wheel [9] is data cleaning tool that also uses data transformations. It presents the user with sample data and as the user applies transformations to the sample data these are stored and system prepares a program. This program can then be run over the entire dataset. While the user cannot really provide feedback over the output of dataset, this work shows the importance of spreadsheet-like interfaces to display data and convey user intentions. 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.

Another data cleaning framework that leverages user feedback is Guided Data Repairs (GDR) [11]. Instead of using data transformations, GDR uses *conditional functional dependencies* [4], *i.e.*, functional dependencies that add constraints to the determinant or dependent sets attributes, to suggest repairs for the data that solve conflicts. The user is presented with those repairs and a user interface lets her accept them or fix them. Iteratively, the data quality problems are fixed. As with Potter's Wheel, the user is not presented with intermediate data – and something like duplicate elimination is impossible – but this frameworks tries to incorporate user feedback and uses constraints to identify data quality problems.

[1] 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 was very important for the development of CLEENEX. It also imposes limits on the actions of the users and reincorporates user feedback in subsequent executions. 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.

7 Conclusions and Future Work

In this paper we presented realization of the concepts of *Quality Constraints* and *Manual Data Repairs* in the context of CLEENEX. This constructs are used to limit the data that the user has to inspect and what can she modify, codifying her feedback into units that can be stored and reapplied in later executions.

We also presented our evaluation of the prototype that implements that framework, obtaining results that show that a 34% gain in the *Precision* of the output data with marginal gains in *Recall*, while requiring user effort during the data cleaning process inferior to what is necessary to clean the output data obtained without QCs or MDRs. Our results also show that the time spent

computing blamed tuples is negligible when compared to the execution of data transformations.

It would be interesting to evaluate the added effort on the part of the designer for adding QCs and MDRs to the data cleaning graph since it conditions how the graph is constructed. It would also be interesting to evaluate the impact of the GUI facilities that allow a user to manipulate how the data returned by an MDR view is displayed. Some limitations of MDRs also need to be addressed, like the limitation of a single action per view and the fact that the user can only see the view whose data she is modifying. At present, the user cannot use information from other relations in the graph to aid her when providing feedback.

References

1. Xiaoyong Chai, Ba-Quy Vuong, AnHai Doan, and Jeffrey F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. In *SIGMOD*, 2009.
2. Francisco Couto, Tiago Grego, Cátia Pesquita, and Paulo Veríssimo. Handling self-citations using google scholar. 13(2), 2009.
3. João Dias. Support for user interaction in a data cleaning process. Master's thesis, 2012.
4. Wenfei Fan, Floris Geerts, and Xibei Jia. Conditional functional dependencies for data cleaning. In *ICDE*, 2007.
5. Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. Ajax: An extensible data cleaning tool. In *SIGMOD Conference*, 2000.
6. Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative data cleaning: Language, model, and algorithms. In *VLDB*, 2001.
7. Helena Galhardas, Antónia Lopes, and Emanuel Santos. Support for user involvement in data cleaning. In *DaWaK*, 2011.
8. David L. Olson and Dursun Delen. *Advanced Data Mining Techniques*. Springer, 2008.
9. Vijayshankar Raman and Joseph M. Hellerstein. Potter'swheel: An interactive data cleaning system. In *VLDB*, 2001.
10. Emanuel Santos and Helena Galhardas. Using argumentation to support the user involvement in data cleaning. In *QBD*, 2011.
11. Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. Guided data repair. *PVLDB*, 4(5):279–289, 2011.