CLEENEX - Debugger

Tiago André da Silva Francisco

tiago.silva.f@tecnico.ulisboa.pt

Instituto Superior Técnico

Abstract. Currently there are several data sources, with different representations. If it is necessary to use data, from multiple sources, that are represented differently, it is necessary to integrate these sources. The integration of these sources can originate several data quality problems, like the existence of approximate duplicate records. One of the solutions to correct data quality problems is to use data cleaning tools. These tools model the data cleaning process as a graph of data transformations. Typically, it is impossible to solve all data quality problems on the first graph execution. The data cleaning process is executed and refined iteratively. For the user to be able to refine the cleaning criteria (i.e., data transformations), it is necessary to understand why certain data quality problems were not solved on the previous iterations. This is why it is important to debug the data transformation graph. The debugging task involves the data derivation process, meaning, going backward and forward on the transformation graph. This document proposes the design and implementation of a debugging component based on a notion of data derivation for CLEENEX [1], a data cleaning prototype. In this component, data derivation is supported through data provenance. To identify the data provenance of a tuple, attributes from the input table are propagated to the respective output tables in the operators that compose the data transformation graph. To validate the debugger, we evaluated the performance of the data derivation component. The evaluation did not showed problems in performance for most of operators.

Keywords: Debugging, Data Provenance, Data Cleaning, Data Transformation.

1 Introduction

The process used to solve the data quality problems is called data cleaning. In the data cleaning tools, the data cleaning process is typically modelled as a data transformation graph which is applied to an input data set and produces a set of good quality output data. When the data that has to be cleaned has a great variety of data quality problems, the data transformation graph which corresponds to the cleaning process may have a higher complexity level. Therefore, it is hard to completely clean up a set of data on the first iteration because it is hard to define data transformations that considers all the possible quality problems of

a big data set. With this in mind, the data cleaning process is executed several times iteratively. Between each iteration, the user analyses the output data of the data cleaning process and can refine, if necessary, some of the cleaning criteria, so the next iteration can solve the problems the previous ones did not. For the user to change some of the criteria, he must first understand why the tuples that have the data quality problems have not been fixed yet. The reason may not be clear just by observing the data cleaning process output data. It is then necessary to execute a data transformation graph debugging task. The debugging process may involve going back and forward on the data transformation graph, transformation by transformation, in order to identify the input tuples that originated a specific output tuple (i.e, *data provenance* [2]), and the outputs tuples that come from a specific input tuple. The process where we go back and forward on the data transformation graph is called *backward data derivation* and *forward data derivation*. Therefore, the term data derivation means go through the data transformation graph. Data derivation allows to understand how a certain tuple with data quality problems was generated and which data cleaning criteria should be refined to deal with the problem. The data cleaning tools must provide a debugging mechanism with a data derivation component.

In this document we propose the implementation of a debugging component for the CLEENEX data cleaning prototype, which should implement the data derivation functionality. One way to support data derivation is through data provenance. To identify the data provenanece, some of the input table's attributes are propagated to their respective output tables, in every CLEENEX operator. To validate the debugging mechanism implemented we evaluated the data derivation component's performance.

This document is organized in five chapters. Chapter two describes the concept of debugger, the debugging features of actual data cleaning tools, the data provenance notion and the relevant researches and advances on the topic. Chapter three explains the debugging tool being developed. Chapter four talks about the validation experiences executed. Finally, chapter five concludes this document presenting the future work to be done in order to improve the CLEENEX debugging component.

2 Related Work

This section describes the *debugger* concept and the state of the art of the *debugging* functionality in the current data cleaning tools. It also describes the notion of *data provenance* and its most relevant related work.

2.1 Debugger

This section explains what a *debugger* and its utility. The term *program* refers to a set of instructions that form a task. Sometimes, the programs do not behave correctly during their execution due to programming errors, or when they do not produce the expected results, due to logical errors. The reasons of this behaviour

may not be clear. Depending on the complexity of the program, the errors (logical or programming) may become hard to find. A *debugger* is a software tool that helps the user understand the reason a certain program has errors and find their origin. In the context of data cleaning, there are specific *debugging* functionalities that are present in the current data cleaning tools. These functionalities are:

- Creation of *breakpoints* - Allows the user to specify the operator where the data cleaning graph execution should be suspended
- Value editing - Allows the user to edit data cleaning graph values
- Code analysis - Allows the user to view code lines that specify operators which constitute the data cleaning graph

2.2 Data Provenance

This subsection describes the notion of data provenance. Data Provenance describes the origins and the history of data in its life-cycle [2]. In the database community, the word provenance is used as synonym of the term *lineage* [3]. Moreover, it is referred as a metadata type that can describe all the data history, starting with their source. The provenance may be classified by its *granularity* [3]. Tan et al [3] consider two type of data provenance which differ by their *granularity*: *workflow provenance* (coarse-grained) and *data provenance* (fine-grained). The term *Workflow* refers to a set of computational procedures interconnected and sequentially executed. The *Workflow provenance* refers to the complete register of all processes and transformations which originated a certain workflow output. In contrast, the *data provenance* describes the origin of part of the data that is in the result of a transformation step. If the result of a certain transformation is a table, the data provenance refers the identification of the entry tuples which originated a certain output tuple. There are two main approaches to calculate data provenance:

- *Annotation approach*: In this approach, the data transformation propagates additional information (*annotations*) to its results. This information is, typically, metadata that describes the output data provenance.
- *Non-annotation approach*: In this approach, since no additional information is propagated, it is necessary to analyse the input and output data of a certain data transformation and relate them with the corresponding transformation to obtain the data provenance.

Besides the *granularity* classification, data provenance can be classified using the way the input and output data of a certain transformation are related. If the data provenance identify the input data that have been involved in the creation of certain output data is referred to as *Why-provenance*. The Why-provenance explains why an output tuple exists, identifying all of the input tuples that contributed to its creation. The *How-provenance*, in comparison to *Why-provenance*, consists in a more detailed explanation about the existence of certain output data of a transformation. Besides identifying input data, it

explains how an output tuple was produced. Besides the *Why-provenance* and *How-provenance*, there is the notion of *Where-provenance* [4], that describes the *location*, at the source database, of the input data that originated certain output data in a certain transformation. In the relational context, the *location* is the exact identification of the value of an attribute in its relation. [2].

2.3 Data Provenance Systems

There have been proposed approaches to determinate data provenance. There are two main approaches, which are, the ones that propagate metadata from the input tuples into the output tuples and the ones that do not propagate metadata. An hybrid approach will also be considered, adopted by some data provenance systems.

The systems analysed in this document that use metadata propagation to identify data provenance are: DBNotes system [5] and the CLEENEX [1] data cleaning prototype.

The DBNotes is a annotation system for relational data. In DBNotes, it is possible to attach zero or more annotations to every value in a relation. DBNotes propagates annotations that describe data provenance, so this information can be kept when the data is transformed. Consider that each input tuple attribute of a certain transformation is associated to an annotation that describes its origin. The resulting tuples of the same transformation will have, in each attribute, an annotation that describes its provenance.

CLEENEX is a data cleaning prototype, where the data cleaning process is modelled as a data transformation graph. In CLEENEX, data provenance is used to go backward on the data transformation graph. In this prototype, data provenance is obtained through key propagation, in the context of relacional data base, from the input tables into their respective output tables. Each operator produces output tuples with values that allow the association of the tuples with their provenance (i.e, the corresponding input tuples). CLEENEX operators are:

- *Map*: Establishes a one-to-many mapping, between an input tuple and the corresponding output tuples.
- *Match*: Applies an approximated *join* function. For each input tuple pair, this operator returns the similarity that exists between these tuples.
- *Cluster*: Groups the input tuples into set of tuples, applying a certain *clustering* algorithm.
- *Merge*: Groups the input tuples, using a certain criteria, and chooses a representative for each group.
- *View*: Corresponds to an arbitrary SQL query.

The systems analysed in this document which do not use metadata propagation to identify data provenance are: SPIDER [6], Orchestra [7], Trio [8] and Chimera [9].

Spider is a schema mapping debugger which was build to help the user in debugging and correcting schema mappings between an origin schema and the destiny schema. Orchestra is a collaborative system which supports data exchange

between several users linked among them. It is responsible for generating the schema mapping between all the data sources of their users. In the SPIDER and Orchestra systems, the schema mappings are used to obtain data provenance. In the schema mapping context, data provenance describes how certain data that follows a source schema is transformed so it can be stored in the destination schema. Trio is a system which extends the traditional relational databases. In Trio, each tuple of a certain relation is associated to a trust value between 0 and 1 and expression that determines its provenance. The expression that determines the provenance of a certain tuple corresponds to the inverse transformation of the transformation that originated the tuple. Chimera is a system where the user defines the data transformation to be applied on the data and the system saves the output data provenance of these transformations. It was created in the scientific context, so it would be possible to recreate certain experiments, from data provenance. Data provenance is composed by the register of all computational procedures that were applied to certain data.

The systems described in this thesis, that implement hybrid approaches to obtain the data provenance are WHIPS [10] and myGrid [11]. WHIPS is system to obtain the data provenance for data warehousing contexts. In this system, the procedures to obtain the data provenance vary depending on the data transformation properties. The myGrid system registers all procedures applied to data, as well as annotating their description.

3 Debugging Component

The goal of this thesis is to implement a *debugging* mechanism for the CLEENEX [1] data cleaning tool. This *debugging* mechanism should allow data derivation. To support data derivation we are going to use data provenance. To identify the data provenance, attributes from the input tables are propagated to the respective output tables. The *backward data derivation* mechanism is very similar to the *forward data derivation* mechanism. To simplify, in this section, we describe only the *backward data derivation* mechanism. The propagated attributes vary depending on the operator type. Table 1 contains the listing of the attributes that should be propagated, by type of operator.

The operators that receive one or more input tables and produce one or more output tables are *Map* (receives one input table), *Match* (receives two input tables) and *SPJU View (Select, Project, Join and Union)* (can receive several input tables). In these operators, the value of the attributes which are part of the key of the input tables are propagated into the output tables. Therefore, the values of the propagated attributes for a certain output tuple of a data transformation identify, in the input tables, the tuples that originated it. Figure 1 illustrates the key attributes propagation, from the input tables to their respective output tables, using the *Map* operator as an example. In this figure, the input table **t1** has the schema **(Id, Name)** and its registers regard the users of a certain service. The attribute **Id** is the key of the **t1** table. The purpose of this operator is, starting from the **Name** attribute, to obtain the name and surname of each

Operator	***Propagation***
Map	Key attributes
Match	Key attributes
ASPJU View	Attributes on which the aggregation was applied
SPJU View	Key attributes
Merge	Attributes on which the aggregation was applied
Cluster	Key attributes

Table 1: Attribute propagation through operators.

user. As we can see in the figure, the output operator schema contains, besides the **(Name, Surname)** attributes, the **Id'** attribute, propagated from the input table. In order to discover which tuple from *t1* originates the tuple (*1, Marcelo Costa*) from *t2*, it is enough to search, in the table t1 for the tuple with $Id = 1$.

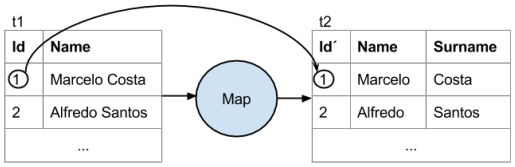

Fig. 1: Key Attribute propagation on *Map* operator.

The *Cluster* operator is a special case, since we know the schema of its inputs and outputs tables. The input schema is constituted by *A1,...An, B1,...Bn*, where *A1,...An* e *B1,...Bn* are input table attributes. The output schema always contains the attributes *(clustid, key)*, where **clustid** is the *Cluster* identifier and **key** corresponds to the key attributes of the input schema *A1,...An, B1,...Bn*, represented by **key1** and **key2**. As with the operators *Map*, *Match* e *SPJU View*, the key attributes of the input tables are propagated to their respective output table. Figure 2 illustrates the propagation of key attributes in the *Cluster* operator. This operator groups into the same *cluster* the tuples that were detected as approximated duplicates by the *Match* operator. The keys of the entry table shown in Figure 2, correspond to the attributes **Id_t1** and **Id_t2**, that have been propagated to the output of the match operator from each of its input tables.

The operators *M*erge and *A*SPJU View (*SPJU With Aggregate*) receive an input table and produce an output table. In these operators, each output tuple was obtained from a set of input tuples. In the operator *ASPJU View*, the input tuples are grouped by the value of a certain attribute or attributes, which value serves as aggregation criterion. In the case of *M*erge operator, the input tuples are grouped by the value of a certain attribute. The attributes which are

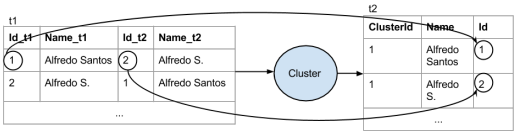

Fig. 2: Key Attribute propagation on *Cluster* operator.

propagated into the output tuples are ones that were used as an aggregation criterion or to an underlying consolidation operation to *Merge*.

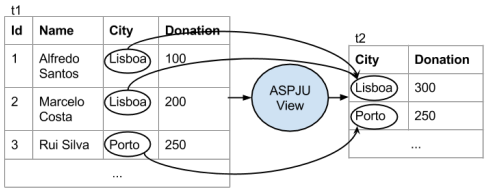

Fig. 3: Attribute propagation in the *ASPJU View* operator.

Figure 3 illustrates the attribute propagation in the *ASPJU View* operator. The input table **t1** of the stated operator has the schema **(Id, Name, City, Donation)**, where the **Id** attribute is the key of this table. The attributes **Name**, **City** and **Donation** represent, respectively, the name, the city of residence and the donated quantity of a certain user. The operator has the function to get the total of donations, by city of residence of users. As we can see in the Figure 3, the propagated attribute to the operator output, in this case, is the **City** attribute, which served as grouping criterion.

Now, we describe the *backward data derivation* queries. Similarly to what happens in attribute propagation, the SQL queries used to support data derivation also vary depending on the operator type. The Tables 2 and 3 contain, respectively, the SQL queries used in the *backward data derivation* and *forward data derivation* processes for each type of operator.

The queries used in the *backward data derivation* process for the *Map*, *Match* and *SPJU View* operators, return all the tuples whose input table key attributes values match the values of the attributes that were propagated to the output. The SQL query used in *backward data derivation* for the *Map*, *Match* and *SPJU View* operators is: `SELECT * FROM  FROM I WHERE I.key = t.key`, where I represents the input table of the *Map* operator and key represents the key attributed propagated.

In the *ASPJU View* and *Merge* operators, the SQL queries returns all the input table tuples , where the attribute value which served as aggregation criterion or for the underlying consolidation to *Merge*, correspond to the attribute

Operator	***SQL Query***
Map	*SELECT * FROM I WHERE I.key = t.key*
Match	*SELECT * FROM I WHERE I.key = t.key*
ASPJU View	*SELECT * FROM I WHERE I.Att = t.Att*
SPJU View	*SELECT * FROM I WHERE I.key = t.key*
Merge	*SELECT * FROM I WHERE I.Att = t.Att*
Cluster	*SELECT * FROM I WHERE I.key1 = t.key OR I.key2 = t.key*

Table 2: *Backward data derivation* SQL Queries.

Operator	***SQL Query***
Map	*SELECT * FROM O WHERE O.key = t.key*
Match	*SELECT * FROM O WHERE O.key = t.key*
ASPJU View	*SELECT * FROM O WHERE O.Att = t.Att*
SPJU View	*SELECT * FROM O WHERE O.key = t.key*
Merge	*SELECT * FROM O WHERE O.Att = t.Att*
Cluster	*SELECT * FROM O WHERE O.key = t.key1 OR O.key = t.key2*

Table 3: *Forward data derivation* SQL Queries.

value that was propagated to the output table. Therefore, the SQL query for these type of operators is `SELECT * FROM I WHERE I.Att = t.Att`, where *Att* corresponds to the propagated attribute.

In the *Cluster* operator, the SQL query returns tuples where the propagated output value has a match with the key attributes of the input table. The correspondent SQL interrogation is `SELECT * FROM I WHERE I.key1 = t.key OR I.key2 = t.key`.

4 Experimental Validation

To validate the *debugging* component, the SQL queries' performance was evaluated. The execution times for all the queries related to CLEENEX operators were measured, varying the number of input tuples for each operator, in order to know the correlation between the execution time and the number of input tuples. The entry tables of CLEENEX operators were populated with different quantity of tuples, generated synthetically, varying between 1.000 and 100.000.000.

The results obtained for the operators *Map*, *Match*, *Merge* e *SPJU View* were similar. The execution time was approximately constant because the SQL query execution plan considered the use of indexes on the primary key of the input table. Figure 4 shows the graphic relative to the *Map* operator.

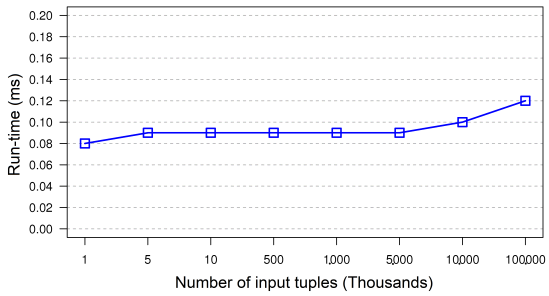

Fig. 4: *Map* Operator.

In the case of the *Cluster* operator, the use of an expression with 'OR' results in an inefficient execution plan. The performance decline becomes more accentuated starting around ten million tuples, as we can see on the Figure 5. In order to increase the performance of the *Cluster* operator, extra indexes were created in the database. Once these indexes were created, the performance was similar to the *Map*, *Match*, *Merge* and *SPJU View* operators.

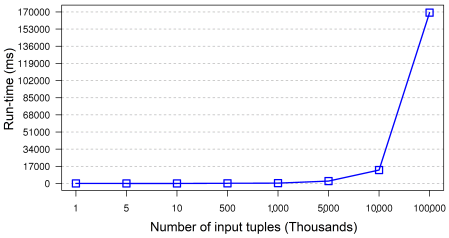

Fig. 5: *Cluster* Operator.

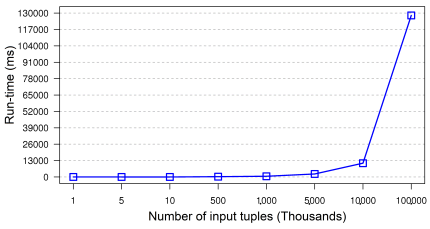

Fig. 6: *ASPJU View* Operator.

As for the *ASPJU View* operator, the performance also decreases as the number of tuples increases, as we can confirm on the Figure 6. The queries relative to this operator return, typically, all the tuples from the input table. To solve this performance issue we suggest an implementation of a mechanism that allows the user control the number of tuples at the exit of this operator.

5 Conclusion

In this document we proposed the design and implementation of debugger mechanism for the CLEENEX data cleaning tool . This mechanism makes use of data provenance to support data derivation, in order to refine the data cleaning pro-

cess. To identify the data provenance, the proposed approach was the propagation of key attributes of the entry tables into their respective output tables. This document also includes a description of a debugger, the debugging features of actual data cleaning tools and the notion of data provenace. We also mentioned the different data provenance classifications which are considered in the literature and several researches related the topic. These researches describe different approaches to obtain the data provenance. This document also describes the procedures that were used to validate the implemented debugger. As future work, we propose the creation of mechanisms to improve the queries' performance on the operators *Cluster* and *ASPJU View*.

References

1. J. L. d. S. Dias, "Support for user interaction in a data cleaning process," Master's thesis, Instituto Superior Técnico, 2012.
2. J. Cheney, L. Chiticariu, and W.-C. Tan, "Provenance in databases: Why, how, and where," *Found. Trends databases*, pp. 379–474, 2009.
3. W. C. Tan *et al.*, "Provenance in databases: Past, current, and future.," *IEEE Data Eng. Bull.*, vol. 30, no. 4, pp. 3–12, 2007.
4. P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and where: A characterization of data provenance," in *ICDT*, pp. 316–330, Springer, 2001.
5. L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, "Dbnotes: a post-it system for relational databases based on provenance," in *SIGMOD*, pp. 942–944, ACM, 2005.
6. B. Alexe, L. Chiticariu, and W.-C. Tan, "Spider: a schema mapping debugger," in *VLDB*, pp. 1179–1182, 2006.
7. T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen, "Update exchange with mappings and provenance," in *VLDB*, pp. 675–686, 2007.
8. J. Widom, "Trio: A system for data, uncertainty, and lineage," in *Managing and Mining Uncertain Data*, Springer, 2008.
9. I. Foster, J. Vockler, M. Wilde, and Y. Zhao, "Chimera: A virtual data system for representing, querying, and automating data derivation," in *SSDM*, pp. 37–46, IEEE, 2002.
10. Y. Cui and J. Widom, "Lineage tracing for general data warehouse transformations," *The VLDB Journal*, 2003.
11. M. Greenwood, C. Goble, R. D. Stevens, J. Zhao, M. Addis, D. Marvin, L. Moreau, and T. Oinn, "Provenance of e-science experiments-experience from bioinformatics," in *Proceedings of UK e-Science All Hands Meeting 2003*, pp. 223–226, 2003.