Improving Data Cleaning Quality using a Data Lineage Facility

Helena Galhardas*
INRIA Rocquencourt
Helena.Galhardas@inria.fr

Daniela Florescu
Propel
Daniela.Florescu@propel.com

Dennis Shasha
Courant Institute, NYU,
shasha@cs.nyu.edu

Eric Simon
INRIA Rocquencourt
Eric.Simon@inria.fr

Cristian-Augustin Saita
INRIA Rocquencourt
Cristian-Augustin.Saita@inria.fr

Abstract

The problem of data cleaning, which consists of removing inconsistencies and errors from original data sets, is well known in the area of decision support systems and data warehouses. However, for some applications, existing ETL (Extraction Transformation Loading) and data cleaning tools for writing data cleaning programs are insufficient. One important challenge with them is the design of a data flow graph that effectively generates clean data. A generalized difficulty is the lack of explanation of cleaning results and user interaction facilities to tune a data cleaning program. This paper presents a solution to handle this problem by enabling users to express user interactions declaratively and tune data cleaning programs.

1 Introduction

The development of Internet services often requires the integration of heterogeneous sources of data. Often the sources are unstructured whereas the intended service requires structured data. The main challenge is to provide consistent and error-free data (aka clean data).

Founded by "Instituto Superior Técnico" - Technical University of Lisbon and by a JNICT fellowship of Program PRAXIS XXI (Portugal)

Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW'2001)
Interlaken, Switzerland, June 4, 2001

(D. Theodoratos, J. Hammer, M. Jeusfeld, M. Staudt, eds.)

http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-39/

To illustrate the difficulty of data cleaning applied to sources of unstructured data, we first introduce a concrete running example. The Citeseer Web site (see [Ins]) collects all the bibliographic references in Computer Science that appear in documents (reports, publications, etc) available on the Web in the form of postscript, or pdf files. Using these data, Citeseer enables Web clients to browse through citations in order to find out for instance, how many times a given paper is referenced. The data used to construct the Citeseer site is a large set of string records. The next two records belong to this data set:

```
[QGMW96] Dallan Quass, Ashish Gupta, Inderphal
Singh Mumick, and Jennifer Widom. Making Views
Self-Maintainable for Data Warehousing. In
Proceedings of the Conference on Parallel and
Distributed Information Systems. Miami Beach,
Florida, USA, 1996. Available via WWW at
www-db.stanford.edu as pub/papers/self-maint.ps.

[12] D. Quass, A. Gupta, I. Mumick, J. Widom,
Making views self-maintanable for data
warehousing, PDIS'95
```

Establishing that these are the same paper is a challenge. First, there is no universal record key that could establish their identity. Second, there are several syntactic and formatting differences between the records. Authors are written in different formats (e.g. "Dallan Quass" and "D. Quass"), and the name of the conference appears abbreviated ("PDIS") or in full text ("Conference on Parallel ..."). Third, data can be inconsistent, such as years of publication ("1996" and "1995"). Fourth, data can be erroneous due to misspelling or errors introducing during the automatic processing of postscript or pdf files, as in the title of the second record ("maintanable" instead of "maintainable"). Finally, records may hold different information, e.g., city and country are missing in the second record.

The problem of data cleaning is well known for decision support systems and data warehouses (see [CD97]). Ex-

traction, Transformation and Loading (ETL) tools and data reengineering tools provide powerful software platforms to implement a large data transformation chain, which can extract data flows from arbitrary data sources and progressively combine these flows through a variety of data transformation operations until clean and appropriately formatted data flows are obtained [RD00]. The resulting data can then be loaded into some database. Some tools are comprehensive but offer only limited cleaning functionality (e.g., Sagent [Sag], ETI [Int], Informatica [Inf]), while others are dedicated to data cleaning (e.g., Integrity [Val]).

It is important to realize that the more "dirty" the data are, the more difficult it is to automate their cleaning with a fixed set of transformations. In the Citeseer example, when the years of publication are different in two records that apparently refer to the same publication, there is no obvious criteria to decide which date to use; hence the user must be explicitly consulted. In existing tools, there is no specific support for user consultation except to write the data to a specific file to be later analyzed by the user. In this case, the integration of that data, after correction, into the data cleaning program is not properly handled. Finally, in existing tools, the process of data cleaning is unidirectional in the sense that once the operators are executed, the only way to analyze what was done is to inspect log files. This is an impediment to the stepwise refinement of a data cleaning program.

This paper presents a mechanism based on exceptions offered by a data cleaning *framework* to support the refinement of data cleaning criteria. The contributions we will emphasize are:

- The explicit specification of user interaction based on exceptions that are automatically generated by the execution of data transformation operations.
- A data lineage facility that enables the user to interactively inspect intermediate data and exceptions; backtrack the cleaning process in the graph of data transformations; investigate the result of each data transformation operation; and modify the input/output of a data transformation interactively.

We illustrate these contributions with a data cleaning application for the Citeseer set of bibliographic textual references. The paper is organized as follows. Section 2 gives an overview of the proposed framework. Section 3 presents the way exceptions are specified and generated. The fourth section explains the data lineage mechanism. Section 5 shows some results and section 6 concludes.

2 Data Cleaning Framework

The proposed framework supports the specification of a data cleaning application as a graph of high-level transformations. Suppose we wish to migrate the Citeseer data set (which is a set of strings corresponding to textual bibliographic references) into four sets of structured and clean data, modeled as database relations: *Authors*, identified by a key and a name; *Events*, identified by a key and a name; *Publications*, identified by a key, a title, a year, an event key, a volume, etc; and the correspondence between publications and authors, *Publications-Authors*, identified by a publication key and an author key.

A partial and high-level view of the data cleaning strategy that we used is the following:

1. Add a key to every input record.
2. Extract from each input record, and output into four different flows the information relative to: names of authors, titles of publications, names of events and the association between titles and authors.
3. Extract from each input record, and output into a publication data flow the information relative to the volume, number, country, city, pages, year and url of each publication. Use auxiliary dictionaries for extracting city and country from each bibliographic reference. These dictionaries store the correspondences between standard city/country names and their synonyms that can be recognized.
4. Eliminate duplicates from the flows of author names, titles and events.
5. Aggregate the duplicate-free flow of titles with the flow of publications.

The definition of each transformation consists in determining "quality" heuristics that automatically lead to the best accuracy (level of cleaning quality) of the results. For considerable amounts of data with a reasonable degree of dirtiness, it is usually impossible to apply the automatic criteria to every record of data. To illustrate this problem, consider the first Extract operation (step 2) in the strategy above. The separation between the author list and the title is assumed to be done by one of the two punctuation marks: ;.". However, some citations use a comma between these two informations, so it is not clear to detect where does the author list finish and the title start. Another example concerns step 4. The two titles presented in the motivating example (starting by "Making Views...") are considered duplicates and need to be merged into a single title (the correctly written instance in this case). Suppose the consolidation phase used an automatic criteria that chooses the longest title among duplicates. Then, it could not be decided which is the correct one among these two. In such situations, it is important to define interaction points in the cleaning process that indicate operations which are not executed automatically and their corresponding input records. The user can then analyze these records and execute some interactive procedure to handle them.

Our framework [GFS+01b] permits to model a data cleaning program satisfying the strategy above as a data flow graph where nodes are logical operators of the following types: mapping, view, matching, clustering, and merging, and the input and output data flows of operators are

logically modeled as database relations. More specifically, the mapping operator takes a single relation and produces several relations as output; it can express any one-to-many data mapping due to its use of external functions. The view operator, which essentially corresponds to an SQL select statement, takes several relations as input and returns a single relation as output; it can express limited many-to-one mappings. The matching operator takes two input relations and produces a single output relation; it computes the similarity between any two input records using an arbitrary distance function. The clustering operator transforms the result of a matching operation into a nested relation where each nested set is a cluster of records from the input relation; the clustering of records is arbitrary. Finally, the merging operator takes a relation representing a set of clusters and apply an arbitrary data mapping to the elements of each cluster.

Example 2.1: The above data cleaning strategy is mapped into the data flow graph of Figure 1. The numbering beside each data cleaning operation corresponds to a step in the strategy. For each output data flow of Step 2, the duplicate elimination is mapped into a sequence of three operations of matching, clustering, and merging. Every other step is mapped into a single operator.

Each logical operator can make use of externally defined functions that implement domain-specific treatments such as the normalization of strings, the extraction of substrings from a string, the computation of the distance between two values, etc. For each input, its execution creates one or more regular output data flows and possibly one exceptional data flow. Regular data flows contain records automatically cleaned using the criteria specified in the operator. However, the processing of the input records may throw an exception specified by the operator, thereby entailing the insertion of the corresponding input record into an exceptional output data flow. The names of all the exceptions that were raised for each input record are attached to each exceptional record.

At any stage of execution of a data cleaning program, a data lineage mechanism enables users to browse into exceptions, analyze their provenance in the data flow graph and interactively apply some corrections. As a first user action, the cleaning criteria can be corrected. For instance, the aforementioned criteria used to separate author names and titles could be modified in order to include the following heuristic: "if a word in lower case is recognized, it is assumed the title has started just after the punctuation mark that precedes it" (in the example above, the lower case word is "views" and the punctuation mark that precedes it is the "," before "Making" so the beginning of the title would be correctly recognized). The corrected logical operation is then re-executed. As a second user action, the data that led to the generation of exceptions may be interactively corrected. For example, the user may interactively separate a list of authors from a title that do not have any punctuation mark between them. The corrected data can then be re-integrated into the data flow graph and the logical operations that take them as input can then be re-executed. This functionality proved to be essential in our experiments with Citeseer reported in Section 5.

3 Management of Exceptions

This section describes how the generation of exceptions can be explicitly specified within the data cleaning operators. The semantics and syntax of each logical operator are presented by example. A formal description of our declarative data cleaning language and the BNF grammar for its syntax can be found in [GFS+01a].

3.1 Mapping Operator

The following mapping operator transforms the relation DirtyData{paper} into a "target" relation KeyDirtyData{paperkey, paper}; this corresponds to Step 1 of Example 2.1.

```
CREATE MAPPING AddKeytoDirtyData
FROM DirtyData
LET Key = generateKey(DirtyData.paper)
{ SELECT Key.generateKey AS paperKey,
  DirtyData.paper AS paper INTO KeyDirtyData }
```

The **create** clause indicates the name of the operation. The **from** clause is a standard SQL from-clause that specifies the name of the input relation of the mapping operator. Then, the **let** keyword introduces a let-clause as a sequence of one or more assignment statements.

In each assignment statement, a relation can be assigned a functional-expression which is an expression that involves the invocation of one or more external functions (that have been registered to the library of functions of the system). If the functional-expression returns a value, it is named *atomic assignment statement*. The let-clause in the example contains an atomic assignment statement that constructs a relation Key using an external (atomic) function generateKey that takes as argument a variable DirtyData.paper ranging over attribute paper of DirtyData. Relation Key is constructed as follows. For every tuple DirtyData(a) in DirtyData[1], if generateKey(a) does not return an exception value exc, then a tuple Key(a, generateKey(a)) is added to relation Key. Otherwise, a tuple $\mathsf{DirtyData}^{exc}(\mathsf{a})$ is added to relation $\mathsf{DirtyData}^{exc}$. We shall say that this statement "defines" a relation Key{paper, generateKey}[2].

Finally, the schema of the target relation is specified by the "{ SELECT key.generateKey AS ...}" clause. It indicates that the schema of KeyDirtyData is built using the attributes of Key and DirtyData.

The mapping command below transforms KeyDirtyData defined above into four target relations, whose

[1] Where a is a string representing a paper.

[2] For convenience, we shall assume that the name of the attribute holding the result of the function is the same as the name of the function.

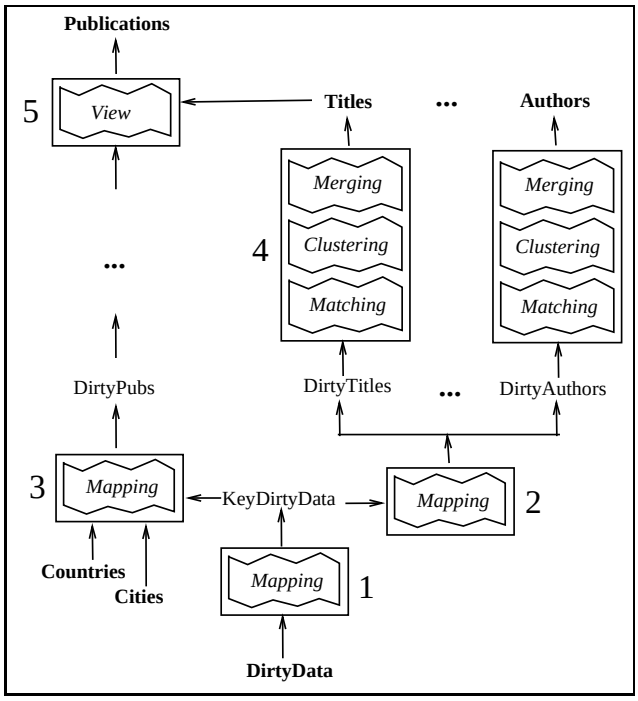

Figure 1: Framework for the bibliographic references

schemas are specified elsewhere in the sections of the data cleaning program that declare the externally defined functions. The schemas of the relations returned by table functions extractAuthorTitleEvent and extractAuthors are {authorlist, title, event} and {id, name} respectively. This operation corresponds to Step 2 in figure 1.

```
CREATE MAPPING Extraction
FROM KeyDirtyData kdd
LET AuthorTitleEvent = extractAuthorTitleEvent(kdd.paper),
      AuthId = SELECT id, name
               FROM extractAuthors(AuthTitleEvent.authorlist)
WHERE length(kdd.paper) > 10
{ SELECT kdd.paperKey AS pubKey, AuthorTitleEvent.title AS title,
  kdd.paperKey AS eventKey INTO DirtyTitles }
{ SELECT kdd.paperKey AS eventKey,
  AuthorTitleEvent.event AS event INTO DirtyEvents
CONSTRAINT NOT NULL event}
{ SELECT AuthId.id AS authorKey,
  AuthId.name AS name INTO DirtyAuthors
CONSTRAINT NOT NULL name }
{ SELECT AuthId.id AS authorKey,
  kdd.paperKey AS pubKey INTO DirtyTitlesDirtyAuthors}
```

The assignment statements in the let-clause are named *table assignment statements* since the corresponding functional-expressions return tables. A relation can also be assigned the result of an SQL **select from where** expression that can make use of previously defined relations.

The **where** keyword introduces a filter expressed as a conjunctive normal form in a syntax similar to an SQL where-clause.

The syntax of the output-clause consists of one or more **select into** expressions that specify the schema of each target relation, and the constraints associated with each target relation. Constraints can be of the following kinds: **not null**, **unique**, **foreign key** and **check**. Their syntax is the same as SQL assertions, but their meaning is different due to the management of exceptions when constraints are violated. If one of the output constraints is violated, the execution of the mapping does not stop and the corresponding input tuple is added to the relation $\mathsf{Extraction}^{exc}$.

We now explain the semantics of the mapping operator. The assignment statements that compose a let-clause are evaluated in their order of appearance in the let-clause for each tuple of the input relation. The syntax of an assignment statement also allows to assign a relation using an **if then else** control structure, in order to expose within the operator the logic of the assignment. If the evaluation of the let-clause does not throw any exception, the filter specified by the where-clause is checked. If it returns true and the output constraints are satisfied, the corresponding tuples are added to the regular output relations of the mapping.

Exceptions may arise during the evaluation of the let-clause or when output constraints are violated. The exceptions can be classified as *non-anticipated* or *anticipated*, depending on the place where they occur. Non-anticipated exceptions are not explicitly declared. They are thrown by the external functions called within the let-clause. The implementation of the let-clause is able to handle any exception thrown within it. If the evaluation of the let-clause for an input relation *S* returns an exception value, then the corresponding input tuple is added to a special output table named S^{exc}. The external functions *extractAuthorTi-*

tleEvent() or *extractAuthors()* called in the Extraction mapping presented may generate non-anticipated exceptions. Consider the second citation given as example in the introduction. As suggested before, *extractAuthorTitleEvent()* may not be able to recognize the end of the list of authors since there are only commas to separate the different elements of the citation. Therefore, an exception is thrown by this function notifying there is no title extracted.

Anticipated exceptions are declared and may be either *explicit* or *implicit*. Explicit exceptions arise when a *throw* clause is specified in the let-clause. There is the possibility to explicitly throw an exception, introduced by the **throw** keyword, in an if-then-else expression. Implicit exceptions occur when some output constraint (**constraint** keyword) is violated. An implicit exception is thrown in the Extraction operation whenever one of the output constraints is violated (e.g. a null event is extracted from one citation).

Next, we briefly present the matching, clustering and merging operations that model the duplicate elimination of author names represented by step 4 and the view operation that model step 5 in figure 1. The semantics, syntax and exceptions for each operator are presented by example and differences with respect to the mapping operator are highlighted.

3.2 Matching Operator

The (self-)matching operator that follows takes as input the relation DirtyAuthors{authorKey, name} twice. Its intention is to find possible duplicates within DirtyAuthors. The let-clause has the same meaning as before with the additional constraint that it *must* define a relation, named distance, within an atomic function assignment. Here, distance is defined using an atomic function editDistanceAuthors computing an integer distance value between two author names. The let-clause produces a relation distance{authorKey1, name1, authorKey2, name2, editDistanceAuthors} that contains one tuple for every possible pair of tuples taken from DirtyAuthors. The WHERE clause filters out the tuples of distance for which editDistanceAuthors returned a value greater than a value computed (by maxDist) as 15% of the maximal length of the names compared. Finally, the INTO clause specifies the name of the target relation (here, MatchAuthors) whose schema is the same as distance.

```
CREATE MATCHING MatchDirtyAuthors
FROM DirtyAuthors a1, DirtyAuthors a2
LET distance = editDistanceAuthors(a1.name, a2.name)
WHERE distance < maxDist(a1.name, a2.name, 15)
INTO MatchAuthors
```

The syntax for the components of the operator have already been presented before. The only subtlety in the SQL-like syntax of the matching operator is the use of the symbol "+" after an input predicate in the **from** clause (it would be "a1 +" in the above example). It indicates that a relation containing all the tuples of DirtyAuthors that did not match (called $\mathsf{DirtyAuthors}^{no-match}$), must be returned by the operator.

The matching operator cannot generate implicit exceptions since there is no explicit output/constraint-clauses. Non-anticipated or explicit exceptions may be thrown in the let-clause. Nevertheless, the general application of a matching operator is required to be completely automatic and no user interaction is usually required.

3.3 Clustering Operator

Consider the relation MatchAuthors generated by the matching operation above presented. The purpose of a clustering operation over MatchAuthors is to produce a set of clusters, each consisting of a set of DirtyAuthors tuples that are sufficiently close to each other and probably correspond to the same author. One possible clustering method is to view each tuple of MatchAuthors as a binary relationship between DirtyAuthors tuples, and group in the same cluster all tuples that are transitively connected. The result of the clustering operation is a relation that has one attribute, clust_id, and as many attributes as there are input relations to the matching operation that defined MatchAuthors, each of which holds the identifier of a tuple in the corresponding input relation. Thus, in our example, the output of the clustering operation over MatchAuthors is formally a relation with three attributes, one for each of the two DirtyAuthors relations, and one clust_id attribute. Suppose that we apply the clustering operation using a transitive closure to the following tuples of MatchAuthors:

MatchAuthors: 1 | D Quass | 6 | Dallan Quass | 1
1 | D Quass | 7 | Quass | 1
2 | A Gupta | 10 | H Gupta | 1

Then, assuming that o_1, o_2, o_3, o_4, and o_5 are the identifiers for authors 1, 2, 6, 7, 10, we would have the following tuples in the output relation, say clusterAuthors:

clusterAuthors: 1 | o_1 | null
1 | o_3 | null
1 | o_4 | null
1 | null | o_1
1 | null | o_3
2 | o_2 | null
2 | o_5 | null
2 | null | o_2
2 | null | o_5

The following is a specification of the clustering operation by transitive closure; and the schema of the target relation clusterAuthors is:
{cluster_id,DirtyAuthors_key1,DirtyAuthors_key2}.

```
CREATE CLUSTERING clusterAuthorsByTranstiveClosure
FROM MatchAuthors
BY METHOD transitive closure
INTO clusterAuthors
```

The clustering operator does not generate any exceptions. Its semantics consists on applying an automatic clustering method to the bi-dimensional space generated by a matching operation.

3.4 Merging Operator

Consider the clusterAuthors relation obtained in the previous example. Each cluster contains a set of names. For each cluster, a possible merging strategy is to generate a tuple composed of a key value, e.g., generated using a generateKey function, and a name obtained by taking the longest author name among all the author names belonging to the same cluster. Thus, the format of the output relation of the merging operation would be a relation, say Authors, of schema {authorKey, name}.

The **using** clause is similar to a **from** clause with the following differences. A **let** clause is always defined wrt the relation(s) indicated in the **from**. In the case of merging, the **let** clause is defined wrt to the relation and attributes of that relation indicated in the **using** clause. In this case we are interested to merge each cluster into a single tuple so the clusterAuthors relation and the cluster_id attribute are specified in the **using** clause. Essentially, each assignment statement is evaluated by iterating over the clusters of the input relation.

The **let** clause is used to construct the attribute values that will compose each tuple over the target relation. A specific notation is introduced to ease the access to the attribute values of the elements of a cluster, which are identifiers. Suppose that the identifier's attributes of the input relation, say P, are associated with relations S_1, S_2 (inputs of the matching). Let A be an attribute of S_1. Then, if p is a variable ranging over the attribute domain clust_id of P, the expression $S_1(p).A$ refers to the set of tuples: $\{x.A \mid x$ is a tuple over S_1 and the identifier of x belongs to cluster p$\}$.

The specification that follows describes the merging operation introduced intuitively above.

```
CREATE MERGING MergeAuthors
USING clusterAuthors(cluster_id) ca
LET name = getLongestAuthorName(DirtyAuthors(ca).name)
      key = generateKey()
{ SELECT key AS authorKey,
  name AS name INTO Authors }
```

In this example, ca is a variable ranging over the clust_id attribute of clusterAuthors. Therefore, expression (DirtyAuthors(ca).name) refers to the set of author names associated with all the DirtyAuthors identifiers of cluster ca. This set is passed to the function getLongestAuthorName that throws an exception if there is more than one author with maximum length belonging to the same cluster. This is a non-anticipated exception and the input tuple corresponding to the cluster that caused it is added to the relation $MergeAuthors^{exc}$. In general, the merging operator can also throw explicit and implicit exceptions if a throw-clause is specified in the let-clause and a constraint clause is indicated, respectively.

3.5 View operator

The last logical operator corresponds to an SQL query augmented with some integrity checking over its result. The interpretation of this clause is first to compute the result of the SQL select statement formed from the **select-into clause**, the **from** clause, and the **where** clause. Then, the set of constraints is evaluated against this result. If a constraint is violated, exceptions are generated.

The following example specifies an SQL join that aggregates together the informations that result from extraction of volume, number, year, etc of each citation with the corresponding titles and event information free of duplicates.

```
CREATE VIEW viewPublications
FROM DirtyPubs p, Titles t
WHERE p.pubKey = t.pubKey
{SELECT p.pubkey AS pubKey, t.title AS title,
  t.eventKey AS eventKey, p.volume AS volume,
  p.number AS number, p.country AS country, p.city AS city,
  p.pages AS pages, p.year AS year,
  p.url AS url INTO Publications
CONSTRAINT NOT NULL title }
```

4 Data Lineage Facility

In this section, we present the functionality supplied to the user for correcting exceptions and the methodology to effectively interact during the execution of the cleaning program. Furthermore, we present algorithms for the incremental execution of a cleaning program. They ensure the consistent integration of interactively modified data into the data flow of transformations.

4.1 Data lineage definition

We first introduce a useful definition of tuple data lineage taken from [CW00].

Definition 4.1: (Tuple Lineage for an Operator). Given a logical operator Op and its output relations $O_1, \ldots, O_n$, $O_i = Op(I_1, \ldots, I_m)$, where $I_1, \ldots, I_m$ are the operator input relations. For a tuple $t \in O_i$, *tuple t lineage for Op in* $I_1, \ldots, I_m$ is defined as $Op^{-1}_{<I_1,\ldots,I_m>}(t) =< I_1^*, \ldots I_m^* >$, where $I_1^*, \ldots, I_m^*$ are maximal subsets of $I_1, \ldots, I_m$ such that:
(i) $Op(I_1^*, \ldots, I_m^*) = t$;
(ii) $\forall I_i^*, \forall t^* \in I_i^*, Op(I_1^*, \ldots, \{t^*\}, \ldots, I_m^*) \neq \phi$
We say in that case that *tuple* $t^* \in I_i^*$ *contributes to t according to Op.*

This definition tells that the lineage tuple sets, given by I_i^*'s, derive exactly tuple t, and each tuple t^* in the lineage sets does in fact contribute to t. In our framework, the input relations of an operator are those specified in the from-clause. Relations that are used in the let-clause, consumed by external functions called within the let-clause, or by clustering algorithms are named *external input relations* and are not considered for data lineage. Given the above definition, the following two propositions apply to our cleaning operators.

Proposition 4.1: If Op is a *matching, mapping, clustering,* or a *SPJ(Select-Project-Join)* view:
$\forall I_1, \ldots, I_m, Op^{-1}_{<I_1,\ldots,I_m>}(t) =< t_1^*, \ldots, t_m^* >$

Our data lineage mechanism builds on this proposition. For every tuple $t \in O_i = Op(I_1,...I_m)$, we keep the identifiers of the tuples $t_1^*,...,t_n^*$ that are such that $Op^{-1}(t) =< t_1^*,...t_m^* >$. These identifiers permit to obtain the lineage of any output tuple. For instance, in the Extraction mapping specified in Section 3, the schema of the output relations DirtyAuthors, DirtyEvents, etc contains the identifier of the input relation (paperKey).

In the case of a merging operator, we have:

Proposition 4.2: If Op is a *merging*, its input relation I has the schema $(A_1,...,A_p,B_1,...,B_k)$, in which $A_1,...,A_p$ represents the attributes with respect to which the merging operation is defined, then $\forall t, merge_I^{-1}(t) = I^*$, where I^* is a partition[3] of I according to $A_1,...,A_p$.

Thus, it is sufficient to keep, for every tuple in the output of a merging, the identifier of the partition from which it is generated. This identifier is given by a tuple of values over $A_1,...,A_p$. In the merging operator MergeAuthors specified in section 3, which is applied after a clustering operation, attributes $A_1,...,A_p$ correspond to the *clust_id* attribute.

This technique does not work in the case of an aggregate-select-project-join (ASPJ) view, because the set of tuples of an input relation that contribute to a given output tuple t cannot be summarized by a compound identifier. However, additional techniques such as keeping auxiliary views could be used to support the tuple lineage of ASPJ views, as suggested in [CW00].

We shall say that the mapping, matching, clustering, merging, and SPJ views are *traceable*, which means that a tuple lineage for these operators can be supported by propagating the record identifiers through operators. For each traceable operator Op, we can define a *lineage tracing query* that returns the lineage of each output tuple t, according to the operator Op:

Proposition 4.3: If Op is a *mapping, matching, clustering, or a SPJ view*, with input relations $I(Id_1,...,Id_k,A_1,...,A_n)$ and output relation $O(B1,...,Bp,Id_1,...,Id_k)$, where $Id_1,...,Id_k$ are the attributes that uniquely identify each input tuple, $LineageQuery(O,Op,I)$ is given by:

$\pi_{(I.*)}[\sigma_{I.Id_1=O.Id_1,...,I.Id_k=O.Id_k}(I \times O)]$

Proposition 4.4: If Op is a *merging* with input relation $I(A_1,...,A_p,B_1,...,B_k)$, in which $A_1,...,A_p$ represent the attributes with respect to which the merging is defined, and output relation $O(E_1,...,E_l,A_1,...,A_p)$, $LineageQuery(O,Op,I)$ is given by:

$\pi_{(I.*)}[\sigma_{I.A_1=O.A_1,...,I.A_p=O.A_p}(I \times O)]$

We have defined the notion of lineage of a tuple according to a single traceable operator. Now, we introduce the definition of lineage of a tuple according to a sequence of traceable operators, i.e. a traceable data cleaning program. Again, our definition is adapted from [CW00].

[3] that is the result of a group-by operation on attributes $A_1,...,A_p$

Definition 4.2: (Tuple Lineage for a Traceable Data Cleaning Program). Let D be a set of tables $R_1,...R_n$, and let $O \subseteq p(D)$ be an output table of a data cleaning program p over D.
(i) if p = identity mapping of R_i then every tuple of R_i *contributes itself according to p*;
(ii) if $p = Op(O_1,...,O_k)$, where each $O_i \subseteq p_i$, where $1 \leq i \leq k$, for some program p_i over D, then suppose that $t' \in O_i$ contributes to some $t \in O$ according to Op and $t^* \in R_i$ contributes to t' according to program p_i, then t^* *contributes to t according to p*.
Then, the *lineage of t according to p* is $p_D^{-1}(t) =< R_1^*,....R_n^* >$, where $R_1^*,....R_n^*$ are subsets of $R_1,...R_n$, such that $t^* \in R_i^*$ iff t^* contributes to t according to p, for $i = 1,...,n$. In that case, R_i^* is the *lineage of t in R_i according to p* and is denoted by $p_{R_i}^{-1}(t)$.

We are now able to describe the algorithm, in Figure 2, to compute the lineage of any relation O according to a traceable cleaning program p.

```
Lineage(O, p, D) {
    if p = Identity(O) then return < O >
    /* else p = Op(I_1,...,I_k) */
    /* I_i = p_i(D) is the result of a */
    /* cleaning (sub-)program p_i */
    < I_1^*,...,I_k^* > ← LineageQuery(O, Op, {I_1,...,I_k})
    D^* ← φ;
    for (i := 1 to k) do
        /* Concatenates the lineage of each */
        /* cleaning (sub-)program */
        D^* ← D^* ∘ Lineage(I_i^*, p_i, D);
    return D^*; }
```

Figure 2: Algorithm for data lineage

4.2 Tuning data cleaning programs

During the execution of a data cleaning program, the data lineage facility offers the following functionality: (i) the user may inspect the set of exceptional tuples, (ii) backtrack in the data flow graph and discover how the exceptional tuples were generated, and (iii) modify attribute values of tuples, insert or delete a tuple of any relation of the data flow graph in order to remedy the exceptions.

The use of this functionality permits to tune a data cleaning program. We describe the method for tuning a program through a sequence of steps modeled by the state-transition diagram of Figure 3. This diagram represents the data cleaning process which corresponds to the execution of the whole cleaning program specified or the execution of any (sub-)program that compose it. A node represents a state of the data cleaning process. Arrows correspond to transitions with a label of the form: *event[condition].action*, which indicates that the *action* is performed whenever *event* occurs and *condition* is satisfied.

The **run** action corresponds to the execution of any data cleaning program. When the execution is finished, the process is in the state **transformations executed**. Three situations may then arise. First, if no exceptions were

thrown during the execution and no more data transformations need to be executed, the cleaning process halts and reaches its **final state**. Second, if there were no exceptions thrown and there are other transformations to be executed, their execution is triggered by the **run** action. The third situation corresponds to the occurrence of exceptions during the execution of the cleaning program and is handled as follows.

When exceptional tuples exist, the user is allowed to **inspect** the exceptional and regular output relations that were generated. In order to discover the tuples that contributed to any exceptional tuple, according to Definition 4.1, the user may obtain the lineage of an exceptional tuple according to the lineage algorithm presented in Figure 2. In this case, we say the user *backtracks* an exceptional tuple. Once the inspection of exceptions is finished, represented by the **data inspected** state in the diagram, the user is able to decide among two procedures to correct the exceptional situations.

The first procedure consists in refining the logic of some operators that were wrongly specified; rewrite incorrect code of external functions (for example, refine the extractAuthorTitleEvent() to properly separate authors from title in the citation: "D. Quass, A. Gupta, I. Mumick, J. Widom, Making views ...") or clustering algorithms; or add entries to auxiliary dictionaries that are incomplete. When the refinement needed is concluded (**code/dictionaries refined** state), the operators modified must be re-executed, as well as those operators in the data cleaning program whose input relations are affected by the output of the refined operators. The **run** action is thus re-triggered for the data cleaning program that encloses the sub-graph constituted by those data transformation operators.

The second procedure for correcting exceptional tuples takes place once exceptions were thrown during the execution of a cleaning program, and no refining actions can help to correct them. This situation occurs when the process is in the **data inspected state** and refinement is not useful. The user interactive **data modification** is then triggered (**modify** action). In this phase, the user may update any relation generated by the cleaning program in order to disambiguate an exceptional situation that cannot be automated (for example, decide the correct title among the two equally sized and similar titles in our motivating example). The user modifications can be insertion, deletion or updating of a tuple. Each tuple interactively modified usually contributes, according to Definition 4.1, to one or more tuples of arbitrary output relations in the graph of data transformations that compose the cleaning program. The operators that produce those output relations must then be re-executed and the **run** action is thus re-triggered. As we will detail next, the re-execution of data cleaning operators after interactive data modifications should and can, sometimes, be performed in incremental mode.

In addition to the methodology above described, the inspection of exceptions should obey to the following principle in order to prevent the user from doing redundant data analysis and correction: *the order by which the user analyzes and corrects exceptions should always be the order determined by the graph of transformations that model the data cleaning program*. This means that exceptions of an operator whose regular output relation(s) contribute to the output tuples of other operators should always be analyzed and corrected before the exceptional tuples occurring for those operators. Considering the Citeseer example and the sequence of logical operations represented in figure 1, the correction of exceptions thrown during the mappings that implement extractions (steps 2 and 3 of the strategy) should precede the correction of exceptions occurring during the merging operation that composes duplicate eliminations in step 4.

All cleaning programs considered in the rest of the paper are considered to be traceable.

4.3 Incremental execution

There are two possible modes of execution for a data cleaning program. First, when the current state of the cleaning process in Figure 3, is the *initial* state or the *code/dictionaries refined* state, the *run* action must materialize the output of all operators that compose the cleaning program being runned. This materialization is required since the execution of these operators potentially produces new values for all output tuples. The operators are then said to be executed in **non-incremental mode**. The second mode of execution may be possible and advisable after tuples have been interactively inserted, deleted or updated by the user (this corresponds to the *data modified* state in Figure 3). If a given modified tuple t' contributes to the output of a cleaning program p, according to Definition 4.2, then p should be only applied to t'. The operators that constitute p are then said to be executed in **incremental** mode. The advantage of running operators in incremental mode is to prevent useless computations for those input tuples $\{t\}$ in the operator input relation I that did not undergo any modification, i.e. $\{t\} = I - \{t'\}$.

Now, let us define the condition an operator must satisfy to be executed in incremental mode. A logical operator can be executed in incremental mode if the *additivity property on its input wrt union and difference* is satisfied. This is specified by definition 4.3 which follows the incremental view maintenance approach [CW91] [AGS93].

Definition 4.3: (Execution in Incremental Mode). A logical operator Op can be executed in *incremental mode* wrt input I_1 if it satisfies the property: $Op((I_1 - I_1^-) \cup I_1^+, I_2) = (Op(I_1, I_2) - Op(I_1^-, I_2)) \cup Op(I_1^+, I_2)$ where I_1, I_2 are the input relations of Op and I_1^- and I_1^+ contain the tuples deleted from and inserted to I_1, respectively.

A generalization of this definition to both inputs is straightforward. A further generalization to any kind of tu-

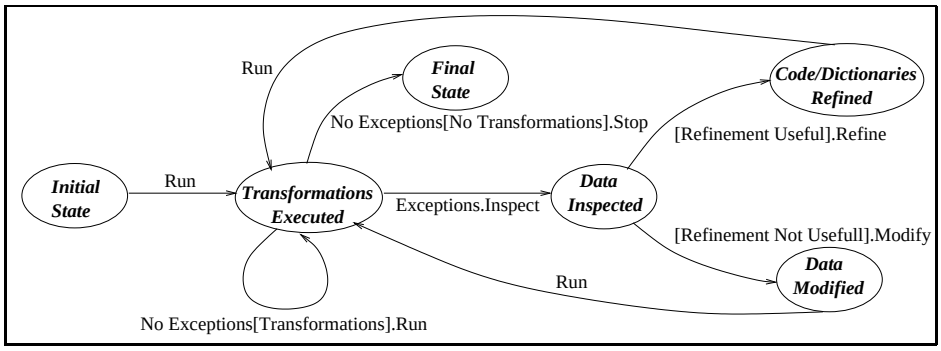

Figure 3: Methodology for cleaning data using the data lineage facility

ple modification can be stated as follows. Any tuple update can be modeled as the deletion of a tuple, whose attributes are assigned the old values, followed by the insertion of a new tuple, whose attributes are assigned the new values, and keeping the same tuple identifier.

As a consequence of Definition 4.3, we have that whenever an operator Op, such that $O = Op(I)$ where I is the input relation of Op, runs in *non-incremental mode*, all the operators of the cleaning programs whose output tuples' lineage belongs to relation O must be executed in non-incremental mode. This means the non-incremental execution of an operator forces the non-incremental execution of the operators whose inputs are affected by its output relations. We also would like to remark that the incremental mode is not possible for operators wrt to their external input relations, as defined before. This is the case of the mapping operator that extracts year, volume, city, etc represented by step 3 in Figure 1, and whose let-clause invokes external functions that accept dictionaries of cities and countries as input. If these dictionaries entries are modified, the mapping must be re-applied to the entire set of input tuples.

We will now present the types of operators for which an incremental execution is possible and those that must be executed in non-incremental mode. Clustering and SPJ views with an order-by clauses do not support the incremental mode, since they do not satisfy the property presented in Definition 4.3, i.e. their results depend on the whole set of input tuples, and a modification on a subset of the input relation potentially modifies the whole output relation[4]. The logical operators that support the execution in incremental mode are: mappings, matchings, mergings, and remaining SPJ views.

Consider a tuple $t(id_1,...,id_p,a_1,...,a_k)$ that belongs to a relation I with schema $I(Id_1,...,Id_p,A_1,...,A_k)$, where attributes $Id_1,...,Id_p$ uniquely identify each tuple $t \in I$. Now, suppose the modified tuple $t'(id_1,...,id_p,a'_1,...,a'_k)$, where $t' \in I$ and an operator $Op(I)$ whose output relation O has the schema $O(B_1,...,B_l,Id_1,...,Id_p)$. Recall that each operator satisfies Propositions 4.1 and 4.2 and is thus traceable. Based on this principle, each incremental operator is able to determine the output tuples that need to be re-computed as being those tuples $t_o \in O : t_o.Id_1 = t'_i.Id1,...,t_o.Id_p = t'_i.Id_p, \forall t'_i \in I$.

The mapping operator iterates over each input tuple and produces one or more tuples per output relation. For a a single tuple modification ($t \rightarrow t'$) in the input relation, its incremental execution deletes $mapping(t)$ and then inserts $mapping(t')$. The remaining output tuples will be the same. The incremental execution of the matching operator over input relations I_1 and I_2 where one of the input relations, let us say I_1, contains the modified tuple t', consists in deleting $matching(t,t_j)$, where $1 \leq j \leq cardinality(I_2)$ and inserting $matching(t',t_j)$. An analogous reasoning is valid for SPJ views. The merging operator collapses the tuples of the input relation that have the same value of a given attribute, let us say A_i. Suppose $t.A_i = a_i$. If $t'.A_i = a'_i$, the incremental execution of the merging consists in the following two steps. First, $merging(t_j)$, where $\{t_j.A_i = a_i \wedge t_j.A_i = a'_i)\}$ and $1 \leq j \leq cardinality(I)$, is deleted from the output relation O. Second, $merging(t_j)$, where $\{t_j.A_i = a_i \wedge t_j.A_i = a'_i)\}$ and $1 \leq j \leq cardinality(I)$, is recomputed and inserted in the merging output relation. Figure 4 intuitively illustrates the incremental execution for a merging operator assuming the user replaces tuple t that belongs to its input relation I by tuple t'.

4.4 Data modification

Let us now analyze the behaviour of an operator that can be executed in incremental mode. The following conflict situation (see Figure 5) may arise. Suppose Op_1, Op_2, Op_3, Op_4 are operators that can be executed in incremental mode. They are all executed, non-incrementally, a first time so their outputs are fully materialized. Suppose output tuples of Op_1 are $t_1,...,t_i,...t_n$ and for simplicity, assume they are propagated, and adequately modified, to the other three operators outputs. Suppose operator Op_2 generates exceptions. In order to correct them, the user interacts by deleting tuple t_i from O_2. Consequently, the incremental execution of Op_3 and Op_4 that

[4] In fact, for some classes of clustering algorithms and views, an incremental execution algorithm could be envisaged. This is object of future work.

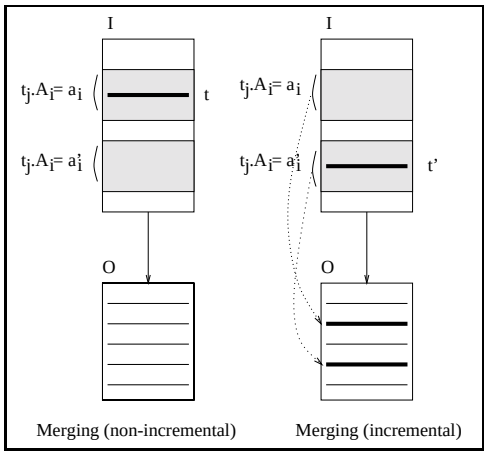

Figure 4: Execution in incremental mode of merging

follows removes t_i from their output relations, according to Definition 4.3. Now, imagine that, when inspecting the exceptional tuples generated by operator Op_4, the user decides to correct one of its exceptional tuples by inserting tuple t_i in the output of operator Op_1. The incremental execution of Op_2 that follows will then re-generate tuple t_i thus invalidating the previous user action that had deleted it. Some special handling must then be provided to avoid these possible conflicting situations.

```
Mapping, Merging, SPJ Views:
delete(Op,t,Output) {
    Output⁻ = Output⁻ ∪ {t} }

Matching:
delete(Op,t,O,I1^no-match) {
    O⁻ = O⁻ ∪ {t}
    if (∄t' ∈ O such that t'.Id1 = t.Id1) then
        I1^no-match = I1^no-match ∪ {t.Id1}
    if (∄t' ∈ O such that t'.Id2 = t.Id2) then
        I1^no-match = I1^no-match ∪ {t.Id2} }

Mapping, Merging, SPJ Views:
insertion(Op,t,O) {
    O⁺ = O⁺ ∪ {t} }

Matching:
insertion(Op,t,O,I1^no-match) {
    O⁺ = Output⁺ ∪ {t}
    I1^no-match = I1^no-match − {t.Id1}
    I1^no-match = I1^no-match − {t.Id2} }
```

Figure 6: Algorithms for logging interactive tuple deletion or insertion

The execution of an operator in incremental mode that takes into account previous user actions applied to its output is now sketched out. Three possible user actions are distinguished: tuple deletion, insertion or updating. As already mentioned, a tuple update corresponds to the deletion followed by the insertion of a tuple. We will thus explain just the deletion and insertion user actions. If an output tuple t is deleted by the user, the operator Op should, in principle, never again generate t in its output relation. The operator must then memorize that tuple t has been deleted from its output, by an user action, and recall this information during further re-executions. If a later user action regenerates the previously deleted tuple t in the Op input relation, Op should automatically recognize the possible existence of a conflict. Analogously, when an output tuple t is inserted as result of a user action, Op should log the user insertion. When further user actions direct or indirectly regenerate tuple t, the possibility of conflict should also be notified.

Before we detail the new execution algorithm, we introduce some auxiliary relations and define the notion of conflict. Any operator Op, executing in incremental mode, must take into account the user modifications applied to its input relation I, by definition of incremental mode. These user mofications are inserted in relations I^+ and I^- that have the same schema as relation I. In addition, Op must be able to consistently integrate user tuple modifications previously applied to its output O. These user-modified tuples are stored in relations O^+, for tuple insertions, and O^- for tuple deletions, where O^+ and O^- have the same schema as O. A *conflict* situation may arise during the integration of the different types of user modifications if the user modifications applied to input I concern the same tuple as the user modifications applied to output O. More concretely, a conflict exist for any two tuples $t \in I^+, (t' \in O^- \vee t' \in O^+) \wedge Op(t) = t'$, where $=$ means the two tuples have the same values for all attributes except the attributes that compose their identifiers. This means the incremental execution of Op re-inserts a tuple in output O that was already deleted or inserted by the user.

We now describe the components of the execution algorithm in incremental mode for an operator Op such that it consistently integrates the user modifications applied to its output relation O. This algorithm is an extension of the incremental execution presented in Definition 4.3.

First, two algorithms that permit to record user insertions and deletions in the output relation of an operator are presented in figure 6. We consider tuple t is deleted from

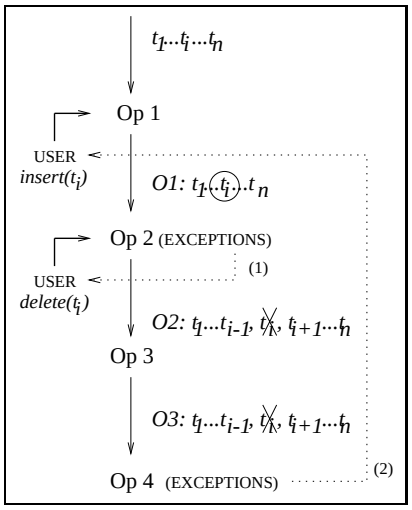

Figure 5: Conflict situation

the output relation O of operator Op or is inserted in same relation O. In the case of matching, we consider two input relations, I_1 and I_2 and the additional output relation $I_1^{no-match}$ that stores the tuples of the input relation I_1 that do not match with any tuple of the input relation I_2, as introduced in section 3. For each operator whose output relation tuples may be modified by the user, two additional output relations are updated: O^- that records tuples the user has deleted; and O^+ that records tuples the user has inserted. The matching operator needs a supplementary automatic procedure that updates the relation $I_i^{no-match}$ according to the user modifications.

Second, the execution algorithm of an operator Op in incremental mode that copes with data modifications in the input relation, represented by I^+ and I^-, and data modifications applied to the output relation recorded in O^+ and O^-, is presented in Figure 7. The input modifications result from the incremental execution of the operators that produce the input relation I of Op. The output modifications correspond to user actions in order to correct exceptions occurred during the execution of Op. The algorithm manages conflicts that may exist when integrating the input and output tuple modifications. We consider the simplest case of an operator accepting a single input relation I and returning a simple output relation O. Besides updating output relation O, according to data modifications, the execution of operator Op in incremental mode generates an additional table of exceptions named $UserOp^{exc}$. Whenever there exists a conflict between two user modifications reflected in the same operator output, the tuples that generate this conflict are added to this exceptional table. For input table $I^{+/-}(Id, A_1, \ldots, A_k)$ and output table $O^{+/-}(Id, B_1, \ldots, B_l, IId)$, where Id is the identifier of each table, the exceptional table $UserOp^{exc}$ has the schema $(O.Id, O.B_1, \ldots, O.B_l, O.IId, I.Id, conflictDescription)$. This additional exceptional relation stores the user modified tuples and the identifier of the user-modified input table that generate the conflict as well as a textual description of the conflict.

$IncOpWithUser(Op, I, I^+, I^-, O, O^+, O^-)$ {
  /* If there are conflicts, insert into $UserOp^{exc}$ */
  if($t \in I^+ \wedge t' \in O^+ \wedge Op(t) = t'$)
    $UserOp^{exc} = UserOp^{exc} \cup \{t, t'\}$
  else if ($t \in I^+ \wedge t' \in O^- \wedge Op(t) = t'$)
    $UserOp^{exc} = UserOp^{exc} \cup \{t, t'\}$
  else { /* there are no conflicts */
    /* executes incrementally for data modifications
    applied to previous operators */
    $O = (O - Op(I^-)) \cup Op(I^+)$ }}

Figure 7: Algorithm for incremental execution

The following proposition states the validity of this algorithm:

Proposition 4.5: Given any operator Op that can be executed in incremental mode, with input relations I, I^+, and I^-, and output relations O, O^+ and O^-, and considering any cleaning program $p = Op(I)$, the algorithm for incremental execution:

$IncOpWithUser(Op, I, I^+, I^-, O, O^+, O^-)$

guarantees that all conflicts are detected whatever is the order by which the user applies data modifications interactively.

5 Experiments

In this section, we show how exceptions are used in the Citeseer data cleaning application and report results of applying the methodology for cleaning data, proposed in Section 4, using exceptions and data lineage.

5.1 Use of exceptions and data lineage

We present two examples to illustrate the use of the mechanism of exceptions and data lineage facility that lead to the

refinement of the cleaning criteria and interactive merging, respectively.

The extraction of authors, title and event name (as specified in Section 3) throws the following exception when trying to separate its elements for the second citation in the motivating example:

KeyDirtyData:
12|D. Quass, A. Gupta, I. Mumick, J. Widom, Making views self-maintanable for data, PDIS'95

Extraction^{exc}:
12|CiteSeerException - TitleIsEmpty

This happens because "Making views self-maintanable for data" is considered to be yet another author according to the first extraction criteria that expects . ; " to separate the author list from the title. After modifying (as explained in Section 2) the logic of the function extractAuthorTitleEvent(), called within the extraction mapping, the correct tuples are returned as output of the Extraction operation:

DirtyAuthors:
1|D. Quass|12
2|A. Gupta|12 ...

DirtyEvents:
12|PDIS'95

DirtyTitles:
12|Making views self-maintanable for data

DirtyTitlesDirtyAuthors:
12|1
12|2 ...

Let us consider the merging operator MergeAuthors presented in Section 3. The function *getLongestAuthorName()* throws an exception if there are distinct author names in the same cluster and more than one with maximum length. In that case, a tuple is written in the exceptional output data flow named MergeAuthors^{exc}. Suppose the tuple MergeAuthors^{exc}: 2 | EqualSizeException is inserted in the exception relation generated by the MergeAuthors operator. The user can then trace this tuple back to the input tuples of ClusterAuthors and DirtyAuthors that have generated it, by soliciting the lineage of the corresponding merging exceptional tuples and clustering output tuples.

ClusterAuthors: 2 | o_2, o_5 DirtyAuthors: 2 | A Gupta
10 | H Gupta

The tuples that constitute the lineage of this exceptional tuple permit the user to discover that her interaction is needed because the system failed to choose an author name. The user may thus insert directly the correct author name into the Authors relation, if "A Gupta" and "H Gupta" are indeed the same person. Otherwise, the user may update the corresponding DirtyAuthors tuples so that they are no longer considered as candidate matches by the matching operator (e.g., expand to "Ashish Gupta" and "Himanshu Gupta").

A similar interaction is required for merging titles (recall the two titles of the motivating example: "Making views...") since the same criteria (longest one) is used to merge automatically and these titles have the same length.

5.2 Application of the methodology

We perform some experiments in order to validate the mechanisms and methodology proposed to refine data cleaning programs and correct data not handled automatically.

Given the set of dirty bibliographic references used to construct the Citeseer site (containing 2 million citations), we chose two sets of 1,000 tuples: a training set to refine criteria and construct auxiliary dictionaries, and a running set to assess the approach.

We apply the methodology described in Section 4 to the training set and tune the data cleaning program to the domain of bibliographic references. Then, we run the cleaning program for the running set and obtained the results presented in Table 5.2. Two kinds of results were obtained for the following three sequences of <action(s) + state[condition]> of the diagram in Figure 3: *(1)* run + data inspected[refinement useful]; *(2)* refine + run + data inspected[refinement not useful]; *(3)* modify + run + final state. First, the number of exceptions thrown for each type of operation is shown. Second, the accuracy of the data transformed in each phase is shown in terms of the recall and precision metrics.

Table 1: Number of exceptions and accuracy of data obtained

Phase	Extract.	Normalz.	Merg.	R/P
(1)	81	16	48	0.32/0.40
(2)	58	3	48	0.36/0.41
(3)	0	0	0	0.74/0.65

We consider the whole set of exceptions classified according to three types of data transformations. *Extraction* criteria enclose the extraction of author names, title and event name (Step 2 in the strategy presented in the introduction); and the extraction of volume, year, number, city name, month, etc from each citation (Step 3 in the strategy). A *normalization* transformation is applied to the extracted event names (and before duplicate elimination is applied to events) and aims at finding the standard event name from a dictionary that maps significative keywords to standard event names (e.g. the keywords "parallel, distributed, information, conference" identify the standard event name: "International Conference on Parallel and Distributed Information Systems (PDIS)"). If more than one standard event name is matched, an exception is thrown. The user can

correct these either by refining the dictionary or writing directly the correct standard event when no more refinement is possible. The *merging* operations refer to the duplicate elimination of author names, titles and event names (Step 4 in the strategy). Merging exceptions are thrown when no unique, author name, title and event name, respectively can be automatically determined for each cluster.

From the analysis of the table, we remark that some exceptions, as normalization (column Normalz.) are mostly solved by refining criteria/dictionaries and others, as merging (Merg. column) require interactive cleaning of data. The extraction (Extract. column) operations benefit partly from the refinement of extraction functions (the number of extraction exceptions was reduced in 30%) and partly from the user procedure.

The last column of the table (R/P) represents two metrics, usually used in the Information Retrieval domain, that assess the quality of the data transformed. We used the running set of 1,000 citations manually cleaned as reference and call it the set of correct citations. *Recall* gives the fraction of correct citations returned by the automatic cleaning program. *Precision* gives the fraction of citations returned by the automatic cleaning program that are indeed correct.

The results reported confirm our expectations in what concerns the quality of data obtained. In fact, after each phase of debugging and user interaction, the accuracy obtained is better. It slightly improves after refinement of code and dictionary entries (e.g. recall increases from 0.32 to 0.36) and a superior gain is obtained after the user data corrections (recall increases to 0.74).

6 Conclusions

In this paper, we presented a mechanism of exceptions and a data lineage facility that assist the user in tuning a data cleaning program.

From our experience with the whole set of Citeseer dirty data, we concluded that for large amounts of data whose level of dirtiness is considerably high, the data cleaning system must be tuned using different samples of data (in this paper we report two sets of 1,000 dirty citations). This corresponds to the methodology used in commercial data cleaning projects. Data domains (e.g. bibliographic references in Computer Science) handled for the first time need always incremental code refinement and enrichment of auxiliary dictionaries. The re-use of cumulated refinements is only profitable after at least one project for a given data domain (unless of course all source of standard knowledge, e.g. dictionary of all universities that issue reports in the Citeseer cleaning application, is supplied as an entry of the cleaning process). The advantage that our data lineage facility and mechanism of exceptions brings is to ease and assist the user in the tuning activity.

Two issues suggest future work. First, machine learning techniques could be incorporated to correct exceptions. When duplicate records are frequent, it would be useful to train the system for automatically modify exceptional tuples according to a user-established correction pattern. Second, a limiting factor for the full incremental execution of a data cleaning program as described in section 4 is the clustering operator. It would be interesting to analyze the classes of clustering methods that permits incremental execution.

References

[AGS93] Inderpal Singh Mumick Ashish Gupta and V.S. Subrahmanian. Maintaining Views Incrementally. In *Proc. of ACM SIGMOD Conf. on Data Management*, 1993.

[CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, March 1997.

[CW91] Stefano Ceri and Jennifer Widom. Deriving Production Rules for Incremental View Maintenance. In *Proc. of the Int. Conf. on Very Large Databases*, 1991.

[CW00] Yingwei Cui and Jennifer Widom. Practical Lineage Tracing in Data Warehouses. In *ICDE*, 2000.

[GFS+01a] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian Saita. Declarative Data Cleaning: Language, Model, and Algorithms. Technical report, INRIA, 2001.

[GFS+01b] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Declarative Data Cleaning: Language, Model, and Algorithms. In *VLDB*, Rome, Italy, September 2001.

[Inf] Informatica. Informatica home page. http://www.informatica.com/.

[Ins] NEC Research Institute. Research Index (CiteSeer). http://citeseer.nj.nec.com/.

[Int] Evolutionary Technologies International. Home Page of ETI. http://www.evtech.com.

[RD00] Erhard Rahm and Hong Hai Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Engineering Bulletin*, 23(3), September 2000.

[Sag] Sagent. Sagent home page. http://www.sagenttech.com/.

[Val] Vality. Home page of the Integrity tool. http://www.vality.com/html/prod-int.html.