Data cleaning and transformation using the AJAX framework

Helena Galhardas

INESC-ID and Instituto Superior Técnico, Avenida Prof. Cavaco Silva, Tagus Park, 2780-990 Porto Salvo, Portugal
hig@inesc-id.pt

Abstract. Data quality problems arise in different application contexts and require appropriate handling so that information becomes reliable. Examples of data anomalies are: missing values, the existence of duplicates, misspellings, data inconsistencies and wrong data formats. Current technologies handle data quality problems through: *(i)* software programs written in a programming language (e.g., C or Java) or an RDBMS programming language, *(ii)* the integrity constraints mechanisms offered by relational database management systems; or *(iii)* using a commercial data quality tool. None of these approaches is appropriate when handling non-conventional data applications dealing with large amounts of information. In fact, the existing technology is not able to support the design of a data flow graph that effectively and efficiently produce clean data.
AJAX is a data cleaning and transformation tool that overcomes these aspects. In this paper, we present an overview of the entire set of functionalities supported by the AJAX system. First, we explain the logical and physical levels of the AJAX framework, and the advantages brought in terms of specification and optimization of data cleaning programs. Second, the set of logical data cleaning and transformation operators is described and exemplified, using the declarative language proposed. Third, we illustrate the purpose of the debugging facility and how it is supported by the exception mechanism offered by logical operators. Finally, the architecture of the AJAX system is presented and experimental validation of the prototype is briefly referred.

1 Introduction

Data cleaning aims at removing errors and inconsistencies from data sets in order to produce high quality data. Data quality concerns arise in three different contexts: *(i)* when one wants to correct data anomalies within a single data source (e.g., duplicate elimination in a file); *(ii)* when poorly structured or unstructured data is migrated into structured data (e.g., when fusing data obtained from the Web); or *(iii)* when one wants to integrate data coming from multiple sources into a single new data source (e.g., in the context of data warehouse construction). In these contexts, the following data quality problems, also called as dirty data, are typically encountered:

– Data coming from different origins may have been created at different times, by different people using different conventions to map real world entities into data. For instance, the same customer may be referred to in different tables by slightly different but correct names, say "John Smith", "Smith John" or "J. Smith". This problem is called the object or instance identification problem, duplicate elimination or record linkage problem in the case of a single source.
– The fact that fused data is produced and used by different entities also enables the existence of missing values. To be aware of a client's age, for instance, is important for a marketing department but not relevant at all for the accounting one.
– Data may be written in different formats. Since no standard notation is generally imposed, data fields may embed data of different natures (the so called free-form fields). An example is a street field that incorrectly contains the zip code and the country name. Moreover, abbreviations as well as synonyms may be used to refer to an object that is represented by their full names in another record.
– Data can contain errors, usually due to mistyping, such as "Joh Smith", even when the same naming conventions are used in different databases.
– Data can have inconsistencies: for instance, two records corresponding to the same person may carry two different birth dates.

Current technologies try to solve these data quality problems in three different ways [1]: *(i)* ad-hoc programs written in a programming language like C or Java, or in an RDBMS (Relational Database Management System) proprietary language; *(ii)* RDBMS mechanisms for guaranteeing integrity constraints; or *(iii)* data transformation scripts using a data quality tool. The use of a general purpose or an RDBMS proprietary language makes data quality programs difficult to maintain and optimize. The mechanisms supported by an RDBMS to enforce integrity constraints do not address the major part of data instance problems. Finally, there is an extensive market of tools to support the transformation of data to be loaded in a data warehouse, the ETL (Extraction, Transformation and Loading) tools, that enclose some data cleaning functionalities. Other data quality tools have been developed from scratch to address specific data quality problems as address standardization and name matching[1].

When an application domain is well understood (e.g., cleaning U.S. names and addresses in a file of customers), there exists enough accumulated know-how to guide the design and implementation of a data cleaning program [11]. Thus, designers can easily figure out which data transformation steps to follow, the operators to use and how to use them (e.g., adjusting parameters). However, for non-conventional applications, such as the migration of largely unstructured data into structured data, or the integration of heterogeneous scientific data sets in cross disciplinary areas (e.g., environmental science), existing data quality tools

[1] The reader can find a recent classification of the existing commercial and research data quality tools in [1].

are insufficient for writing data cleaning programs. The main challenge with these tools is the design of a data flow graph that effectively generates clean data, and can perform efficiently on large sets of input data. This two-fold task can be difficult to achieve, because: (i) there is no clear separation between the logical specification of data transformations and their physical implementation, and (ii) there is no support for debugging the reasoning behind cleaning results nor interactive facilities to tune a data cleaning program.

We have proposed the AJAX tool[2][12] to overcome these two aspects. The main contributions of AJAX with respect to existing data cleaning technology are the following:

- A data cleaning *framework* that attempts to separate the logical and physical levels of a data cleaning process. The logical level supports the design of a data flow graph that specifies the data transformations needed to clean the data, while the physical level supports the implementation of the data transformations and their optimization. An analogy can be drawn with database application programming where database queries can be specified at a logical level and their implementation can be optimized afterwards without changing the queries.
  We propose *five logical data transformation operators* encapsulating distinct semantics that are orthogonal and complete. These operators derive from an analysis of the types of mappings with respect to input and output tuples that are expressed by intuitive and conceptual data transformations. This approach is original when compared to commercial data cleaning tools in the sense that it prevents from having a large number of operators that are sometimes redundant. Our operators were proposed to extend SQL in order to specify those mappings.
- A *declarative language* for specifying these data cleaning logical operators. A *mechanism of exceptions* is associated to each logical operator and provides the foundation for explicit user interaction.
- A *debugger (or explainer mechanism)* that helps the user in debugging and tuning a data cleaning application program. Such a debugger facility, commonly used in programming environments, is new in the domain of data cleaning applications. An audit trail mechanism allows the user to navigate through the results of data transformations in order to discover why some records are not automatically treated. To solve those cases, the user may refine some cleaning criteria or manually correct data items.

AJAX does not provide any method to discover data problems that need to be cleaned. Before specifying a data cleaning and transformation program using AJAX, the user must be aware of the data anomalies that need to be solved. An interesting direction for future work would be to enrich the set of operators already provided by AJAX with new operators that are able to analyze data and automatically (by applying statistical techniques or data mining algorithms)

[2] The first prototype of AJAX was designed and implemented at Inria Rocquencourt.

detect the data quality problems that need to be solved. However, this issue is not addressed in the current version of the system.

This paper presents an overview of the entire set of functionalities supported by the AJAX system. First, we explain the logical and physical levels of the AJAX framework, and the advantages brought in terms of specification and optimization of data cleaning programs. Second, the set of logical data cleaning and transformation operators is described and exemplified, using the declarative language proposed. We also illustrate the SQL equivalent of two of the AJAX operators. Third, we illustrate the purpose of the debugging facility and how it is supported by the exception mechanism offered by logical operators. Finally, the architecture of the AJAX system is presented and experimental validation of the prototype is briefly referred.

Most of these aspects have been published separately elsewhere [9], [10], but none of the previous publications concerning AJAX provided a broad description that covers all details.

The rest of this paper is organized as follows. In Section 2, we present our motivating example. Then, Section 3 details the principles of the AJAX framework. Section 4 explains the debugger mechanism. The architecture of the AJAX system and experimental validation are presented in Section 5. Related work is summarized in Section 6 and we conclude in Section 7.

2 Motivating example

We illustrate the functionalities of AJAX using a case study. The application consists of cleaning and migrating a set of textual bibliographic references, extracted from postscript or pdf files that were obtained by a Web crawler[3], into a set of structured and duplicate-free relations.

Suppose we wish to migrate the original Citeseer dirty set of strings that correspond 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. The purpose of the underlying input-output schema mapping is to derive structured and clean textual records so that meaningful queries can be performed (e.g., how many papers a given author has published in 2005).

Figure 1 presents an example for two dirty citations that represent the same bibliographic reference. The corresponding cleaned instances are produced by the data cleaning process and stored in the four resulting relations. In the figure, the Publications table contains a single tuple that stores the correct and duplicate-free information represented by the two dirty citations. The title in this tuple, "Making Views Self-Maintainable for Data Warehousing", is the correct one among the two dirty titles, and the event key value ("PDIS") references

[3] This information was used to construct the Citeseer Web site [18].

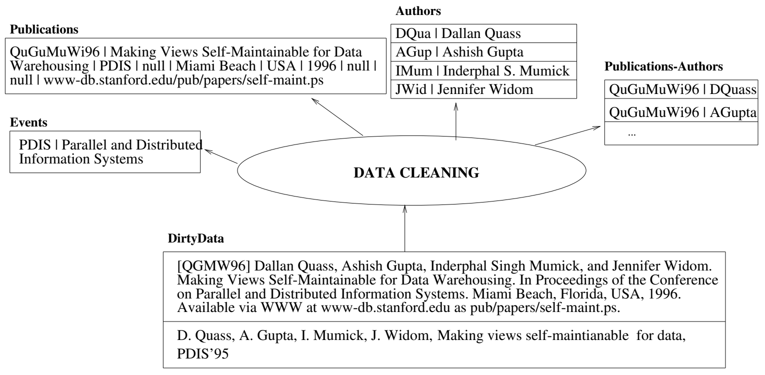
Publications
QuGuMuWi96 | Making Views Self-Maintainable for Data Warehousing | PDIS | null | Miami Beach | USA | 1996 | null | null | www-db.stanford.edu/pub/papers/self-maint.ps
Authors
DQua | Dallan Quass
AGup | Ashish Gupta
IMum | Inderphal S. Mumick
JWid | Jennifer Widom
Publications-Authors
QuGuMuWi96 | DQuass
QuGuMuWi96 | AGupta
...
Events
PDIS | Parallel and Distributed Information Systems
DATA CLEANING
DirtyData
[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.
D. Quass, A. Gupta, I. Mumick, J. Widom, Making views self-maintianable for data, PDIS'95

Fig. 1. Cleaning textual bibliographic references - an example.

the standardized event name ("Parallel and Distributed Information Systems") stored into the Events table. The fields concerning the location ("Miami Beach" and "USA") and the url where the paper is available, have been correctly extracted by the cleaning process and associated to the cleaned publication instance. Finally, "1996" was identified as the correct year of publication. The Authors table stores one row for each real author. The data cleaning process recognizes the two distinct forms of writing the same author name and chooses the longest one. The Publications-Authors table keeps the references for cleaned authors and cleaned publications.

3 AJAX framework

The development of a data cleaning program able to solve problems as the ones described in Section 2 actually involves two activities. One is the design of the graph of data transformations that should be applied to the input dirty data and whose main focus is the definition of "quality" heuristics that can achieve the best accuracy of the results. A second activity is the design of "performance" heuristics that can improve the execution speed of data transformations without sacrificing accuracy. AJAX separates these two activities by providing a logical level where a graph of data transformations is specified using a declarative language, and a physical level where specific optimized algorithms with distinct tradeoffs can be selected to implement the transformations.

3.1 Logical Level

A partial and high-level view of a possible data cleaning strategy for handling the set of bibliographic references introduced in Section 2 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.

At the logical level, the main constituent of a data cleaning program is the specification of a data flow graph where nodes are data cleaning operators, and the input and output data flows of operators are logically modeled as database relations. The design of our logical operators was based on the semantics of SQL primitives extended to support a larger range of data cleaning transformations.

Each operator can make use of externally defined functions or algorithms that implement domain-specific treatments such as the normalization of strings, the extraction of substrings from a string, etc. External functions are written in a 3GL programming language and then registered within the library of functions and algorithms of AJAX.

The semantics of each operator includes the automatic generation of a variety of exceptions that mark input tuples which cannot be automatically handled by an operator. This feature is particularly required when dealing with large amounts of dirty data which is usually the case of data cleaning applications. Exceptions may be generated by the external functions called within each operator. At any stage of execution of a data cleaning program, a debugger mechanism enables users to inspect exceptions, analyze their provenance in the data flow graph and interactively correct the data items that contributed to its generation. Corrected data can then be re-integrated into the data flow graph.

3.2 Logical operators

We now present our logical operators based on a classification of data transformations where we consider the type of mapping that they express with respect to their input and output tuples. The proposed operators are parametric in the sense that they may enclose the invocation of generic external functions. A natural choice is to use SQL queries to express these mappings. This led us to introduce a logical operator, called *view*, that corresponds to an arbitrary SQL query. There are several obvious advantages of doing this: SQL is a widespread used language, and existing RDBMSs include many optimization techniques for SQL queries. However, the relational algebra is not expressive enough to capture the new requirements introduced by data transformation and cleaning applications as stated in [3]. Our next operator, called *map*, captures all iterator-based mappings that take a single relation as input and produces several relations as output (and therefore, several tuples for each input tuple). The map operator is proposed to enable the application of any kind of user-defined function to each input tuple. A map has the general form of an iterator-based one-to-many mapping. In the Citeseer example, formatting, standardization and extraction are implemented through a map operator.

The third operator, called *match*, captures a specific sub-class of iterator-based many-to-one mappings that consists of associating a similarity value to any two input records using an arbitrary similarity metric. The match takes two relations as input and produces one output relation. This operation is obviously expressible using a view operator but having it as a distinct first-class operator considerably facilitates its optimization. The fourth operator, called *cluster*, captures a subclass of non iterator-based many-to-many mappings that consists of transforming an input relation into a nested relation where each nested set is a cluster of records from the input relation, and the clustering of records is arbitrary. One example of the cluster operator is the application of a transitive closure method to assemble similar event records. We decided to define this operator for two reasons. The first reason is the fact that it accepts a particular signature, i.e., pairs of tuples equipped with a distance. The second reason for considering it as a first-class operator is due to the possibility of optimizing the match and cluster operators. The next operator, called *merge*, captures another subclass of non iterator-based many-to-many mappings that corresponds to grouping input elements according to a given criterion, and then applying an arbitrary aggregate data mapping to the elements of each group. This operator is an extension of the SQL group-by aggregate query where user-defined aggregate functions can be used.

To illustrate the use of these operators, we show in Figure 2 the simplified graph of data transformations, that corresponds to the cleaning strategy introduced earlier in this section, in terms of our logical operators. The numbering beside each data cleaning operation corresponds to an intuitive transformation in the strategy. For each output relation of Step 2, we have to identify and eliminate duplicate records. In the figure, duplicate eliminations corresponding to Step 5 are mapped into sequences of one match, one cluster, and one merge operator. Every other transformation is mapped into a single logical operator.

3.3 Declarative language

AJAX provides an expressive and declarative language for specifying data cleaning programs, which consists of SQL statements enriched with a set of specific primitives to express map, match, cluster, merge and view transformations. Each one of these primitives corresponds to a transformation whose physical implementation takes advantage of existing RDBMS technology. The declarative nature of the language provides opportunities for automatic optimization and facilitates the maintenance of a data cleaning program.

Syntactically, each operator specification has a FROM clause that indicates the input data flow, a CREATE clause, which names the output data flow (for further reference), a SELECT clause specifying the format of the output data and a WHERE clause to filter out non interesting tuples from the input. An optional LET clause describes the transformation logics that has to be applied

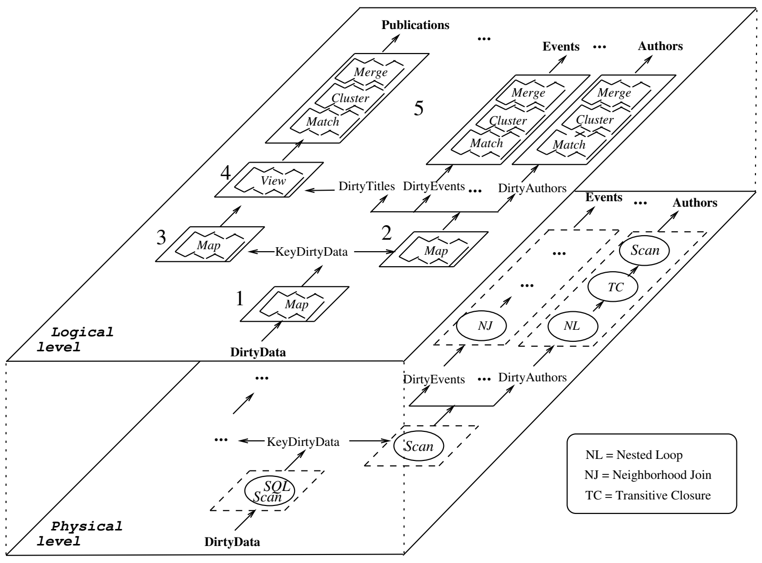

Fig. 2. Graph of logical and physical data transformations for the bibliographic references.

to each input item (tuple or group of tuples) in order to produce output items. This clause contains limited imperative primitives: the possibility to define local variables, to call external functions or to control their execution via if/then/else constructs. Finally, the cluster operator includes a BY clause which specifies the grouping algorithm to be applied, among the ones existing in the AJAX library of algorithms.

To illustrate the semantics and syntax of the AJAX operators, we exemplify the map operator that corresponds to the data transformation 1 and the match operator represented by 5 in Figure 2, in Examples 1 and 2 respectively.

Example 1. The following map operator transforms the relation DirtyData{paper} into a "target" relation KeyDirtyData{paperkey, paper} by adding a serial number to it. The LET clause contains a statement that constructs a predicate 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 fact DirtyData(a) in the instance of DirtyData[4], if generateKey(a) does not return an exception value exc, then a fact Key(a, generateKey(a)) is added to the instance of Key. Otherwise, a fact DirtyData^{exc}(a) is added to the instance of DirtyData^{exc} (which is the map output relation that stores exception tuples). We shall say that this statement "defines" a relation Key{paper, generateKey}[5]. The schema of the target relation

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

[5] 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.

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. Finally, the constraint stipulates that a paper attribute value must never be null.

```
CREATE MAP AddKeytoDirtyData
FROM DirtyData
LET Key = generateKey(DirtyData.paper)
{ SELECT Key.generateKey AS paperKey, DirtyData.paper AS paper INTO KeyDirtyData
CONSTRAINT NOT NULL paper}
```

□

A map operator that produces a single output relation and whose let-clause encloses only atomic assignment statements as the example above may be implemented by one *insert into ... select from* clause and one *create table* clause (as illustrated in Example 2). However, in a general case, a map operator may produce one or more tuples (belonging to a single or several output relations) for each input tuple. In such situation, it may not be possible to write SQL statements that represent the same semantics.

Example 2. The SQL equivalent of the map defined in Example 1 is as follows:

```
CREATE TABLE KeyDirtyData(paperKey varchar2(100),
    paper varchar2(1024) NOT NULL);
INSERT INTO KeyDirtyData
    SELECT generateKey() paperKey, dd.paper paper
    FROM DirtyData dd
```

□

Example 3 illustrates a match operation. The let-clause has the same meaning as in a map operation with the additional constraint that it *must* define a relation, named distance, within an atomic assignment statement. 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} whose instance has one tuple for every possible pair of tuples taken from the instance of DirtyAuthors. The where-clause filters out the tuples of distance for which editDistanceAuthors returned a value greater than a constant value given by maxDist. Finally, the **into** clause specifies the name of the output relation (here, MatchAuthors) whose schema is the same as distance.

Example 3. This (self-)match operator takes as input the relation DirtyAuthors{authorKey, name} twice. Its intention is to find possible duplicates within DirtyAuthors.

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

□

A simple match operator is mapped onto a *create table clause* and an *insert into clause* that encloses a nested query. The inner query computes the distance values and the outer query imposes a condition on the distance obtained, according to a given maximum allowed distance. Example 3 is mapped into the following SQL statements.

Example 4.

```
CREATE TABLE MatchAuthors(authorKey1 varchar2(100),
    authorKey2 varchar2(100), distance number);
INSERT INTO MatchAuthors
  SELECT authorKey1, authorKey2, distance
  FROM (SELECT a1.authorKey authorKey1, a2.authorKey authorKey2,
               editDistanceAuthors(a1.name, a2.name) distance
               FROM DirtyAuthors a1, DirtyAuthors a2)
  WHERE distance < maxDist;
```

□

3.4 Physical level

At the physical level, certain decisions can be made to speed up the execution of data cleaning programs. First, the implementation of the externally defined functions can be optimized. Second, an efficient algorithm can be selected, among a set of alternative algorithms, to implement each logical operator. A very sensitive operator to the choice of execution algorithm is matching. An original contribution of our data cleaning system is the possibility to associate with each optimized matching algorithm, the mathematical properties that the similarity function used in the match operator must have in order to enable the optimization, and the parameters that are necessary to run the optimized algorithm. Then, our system enables the user to specify, within the logical specification of a given matching operator, the properties of the distance function, together with the required parameters for optimization. The system can consume this information to choose the best algorithm to implement a match. The important point here is that users control the proper usage of optimization algorithms. They first determine (in the logical specification) the matching criteria that would provide accurate results, and then provide the necessary information to enable optimized executions. Figure 2 shows the algorithms selected to implement each logical operation.

3.5 Optimization of the match operator

The match operator computes an approximate join between two relations. The semantics of this operation involves the computation of a Cartesian product between two input relations using an arbitrary distance function. Such semantics guarantees that all possible matches are captured under the assumption that correct record matching criteria are used. However, while doing so, a performance penalty is incurred since the Cartesian product based semantics with external function calls is usually evaluated (e.g. within an RDBMS) through a nested loop join algorithm with external function calls. The match operator is thus one of the most expensive operators in our framework once a considerable amount of data is involved.

For this reason, we dedicate particular attention to the match optimization opportunities. A match operator with an acceptance distance of ϵ computes a distance value for every pair of tuples taken from two input relations, and returns those pairs of tuples (henceforth, called *candidate matches*) that are at a maximum distance of ϵ from each other. In fact, since the distance function is an approximation of the actual closeness of two records, a subsequent step must determine which of the candidate matches are the *correct matches* (i.e., the pairs of records that really correspond to the same individual).

For very large data sets, the dominant factor in the cost of a match is the Cartesian product between the two input relations. One possible optimization is to pre-select the elements of the Cartesian product for which the distance function must be computed, using a *distance filter* that allows some *false matches* (i.e., pairs of records that are falsely declared to be within an ϵ distance), but no *false dismissals* (i.e., pairs of records falsely declared to be out of an ϵ distance). This pre-selection of elements is expected to be cheap to compute.

Distance-filtering optimization This type of optimization has been successfully used for image retrieval [7] and matching of textual fields [14]. Formally, the result of a match between two input relations S_1 and S_2 in which the distance, *dist*, between two elements of S_1 and S_2 is required to be less than some ϵ, is a set:

$$\{(x, y, dist(x, y)) \mid x \in S_1 \wedge y \in S_2 \wedge dist(x, y) \leq \epsilon\} \tag{1}$$

The distance filtering optimization requires finding a mapping f (e.g., get the length of a string) over sets S_1 and S_2 , with a distance function $dist'$ much cheaper than *dist*, such that:

$$\forall x, \forall y,\ dist'(f(x), f(y)) \leq dist(x, y) \tag{2}$$

Having determined f and $dist'$, the optimization consists of computing the set of pairs (x, y) such that $dist'(f(x), f(y)) \leq \epsilon$, which is a superset of the desired result:

$$Dist\_Filter = \{(x, y) \mid x \in S_1 \wedge y \in S_2 \wedge dist'((f(x), f(y)) \leq \epsilon\}$$

Given this, the set defined by (1) is equivalent to:

$$\{(x, y, dist(x, y)) \mid (x, y) \in Dist\_Filter \wedge dist(x, y) \leq \epsilon\} \tag{3}$$

A generic algorithm that implements this optimization is shown in Figure 3. This algorithm, called *Neighborhood Join* or NJ for short, is effective when

```
Input: $S_1$, $S_2$, $dist$, $\epsilon$, $dist'$, $f$
{
  $P_1$ = set of partitions of $S_1$ according to $f$
  $P_2$ = set of partitions of $S_2$ according to $f$
  $\forall s_1 \in p_1, p_1 \in P_1 : f(s_1)$ = cte
  $\forall s_2 \in p_2, p_2 \in P_2 : f(s_2)$ = cte
  for each partition $p_1 \in P_1$ do {
    for each partition $p_2 \in P_2$ such that $dist'(f(p_1), f(p_2)) \leq \epsilon$ do {
      for each element $s_1 \in p_1$ do {
        for each element $s_2 \in p_2$ do {
          if $dist(s_1, s_2) \leq \epsilon$ then
              Output = Output $\cup$ $(s_1, s_2)$ }}}}
}
```

Fig. 3. Neighborhood Join algorithm.

both the number of partitions generated by the mapping f, and the number of elements in the partitions selected by the condition on $dist'$ wrt ϵ, are much smaller than the size of the original input data set. The filter used in Figure 3, map $= f$, serves to partition the input data sets and order the partitions accordingly. After applying this partitioning, only the pairs of tuples that belong to partitions satisfying $dist'(f(p_1), f(p_2)) \leq \epsilon$ are compared through the distance function *dist*. This condition is imposed through the first two for cycles of the algorithm.

This optimization is illustrated below on a match operation of the Citeseer data cleaning program that takes as input the relation DirtyTitles{pubKey, title, eventKey} twice. The line between the %'s is an annotation that indicates the type of optimization and the distance filtering property of the distance function.[6] Annotations can then be used by AJAX to guide the optimizer on choosing the appropriate physical execution algorithm for the match operator. We assume that maxDist is an integer. The editDistanceTitles function is based on the Damerau-Levenshtein metric [17] that returns the number of insertions, deletions and substitutions needed to transform one string into the other.

Example 5.

```
CREATE MATCH MatchDirtyTitles
FROM DirtyTitles p1, DirtyTitles p2
LET distance = editDistanceTitles(p1.title, p2.title)
WHERE distance < maxDist
%distance-filtering: map=length; dist=abs %
INTO MatchTitles
```

The Damerau-Levenshtein edit-distance function has the property of always returning a distance value bounded by the difference of lengths l of the strings compared. Thus, if l exceeds the maximum allowed distance maxDist, there is

[6] In the Citeseer application, the distance filtering optimization was also applicable for matching author and event names.

no need to compute the edit distance because the two strings are undoubtedly dissimilar. This property suggests using as mapping f, the function computing the length of a string, and as $dist'$ a function abs such that $\mathsf{abs}(x, y) = |x - y|$.

4 Debugging data cleaning programs

The goal of a data cleaning process is to produce clean data of high quality, i.e., consistent and error-free. When handling large amounts of data with a considerable level of dirtiness, automatic cleaning criteria are not able to cover the entire data set. There are two main reasons for this: cleaning criteria may need to be refined, and some cleaning decisions cannot be automatically disambiguated and thus user interaction is needed. In current technology, tuples that are rejected by the data transformations are inserted into a log file to be later analyzed by users. When the number of rejected tuples is large, which is usually the case when treating large data sets, it is fundamental to provide a user-friendly environment for discovering why some dirty records are not handled by the cleaning process. Our framework offers a facility to assist the user on this task. First, we provide a mechanism of exceptions that marks tuples that cannot be handled automatically as mentioned in Section 3. Second, a debugger mechanism is provided to allow the user to interactively inspect exceptions.

To better illustrate the problem, consider the standardization of citations and the extraction of author names, title and event names that correspond to transformation *2* in Figure 2. We may consider that the separation between the author list and the title is done by one of the two punctuation marks: {“;.“}. However, some citations, as the second dirty one in Figure 1 (i.e., “ D. Quass, A. Gupta, I. Mumick, and J. Widom, Making views self-maintanable for data, PDIS’95”), use a comma between these two kinds of informations, so it is not easy to automatically detect where does the author list finish and the title starts. Therefore, the user may need to refine the corresponding extraction criteria so that this situation becomes automatically treated. Another example concerns the duplicate elimination applied to dirty publication records (transformation *5* in Figure 2). The two titles presented in the motivating example (starting by “Making Views...”) are considered duplicates and need to be consolidated into a single title (the correctly written instance in this case). If the consolidation phase uses an automatic criterion that chooses the longest title among duplicates, then it cannot decide which is the correct one among these two titles, since they have the same length. Here again, manual intervention is required.

In order to mark input tuples that cannot be transformed automatically, a logical operator generates one *exceptional output relation* per input to store such tuples. The other output relations of an operator, called *regular output relations*, contain transformed tuples. An exceptional tuple corresponds to an input tuple that does not satisfy the cleaning criteria associated to the transformation. Given this, during the execution of a data cleaning program, a *debugger or explainer facility* offers the following functionality to the user: (i) inspection of exceptional tuples using data derivation mechanisms; (ii) navigation through the data flow graph to discover how exceptional tuples were generated, and (iii) support for refining cleaning criteria and modifying tuples to remedy exceptions. This functionality allows the user to tune a data cleaning application and, consequently to improve the accuracy of the cleaned data.

5 Architecture

The architecture of the AJAX system is represented in Figure 4. There are two types of components in the system: *repositories* that manage data or fragments of code; and *operational units* that constitute the execution core. AJAX encloses the following three repositories:

– The *data repository* stores data in a relational database management system or in a set of text files and offers a JDBC-like interface. In both cases, data include all input data of a data cleaning application (including dictionaries), the cleaned output data and the intermediate relations generated by logical operators, including exceptional output relations.
– The *library of functions* encloses the code of the external functions that are called within each logical operator. Examples of such functions are specific string matching functions (e.g., edit-distance [14]). This library contains a set of default functions and can be extended to include new user-defined functions.
– The *library of algorithms* encloses the clustering algorithms (e.g., by transitive closure) that can be invoked within a cluster operator and the physical algorithms that implement the logical operators. Analogously to the library of functions, users can add new algorithms to the set of existing ones.

The core of the AJAX system is implemented by the following operational units:

– The *analyzer,* which parses a data cleaning program and generates an equivalent internal representation;
– The *optimizer*[7], that assigns efficient physical execution algorithms to the logical operators specified and returns an optimal execution plan for a given data cleaning specification;
– The *code generator*, that generates executable code to implement each logical operator in the execution plan;
– The *execution engine*, which executes operators according to the order determined by the specification and the optimizer;
– The *debugger or explainer*, that triggers an audit trail mechanism allowing the user to discover why exceptional tuples are generated and supporting interactive data modification to correct exceptions.

[7] In the current version of the AJAX prototype, the optimization decisions are manually taken.

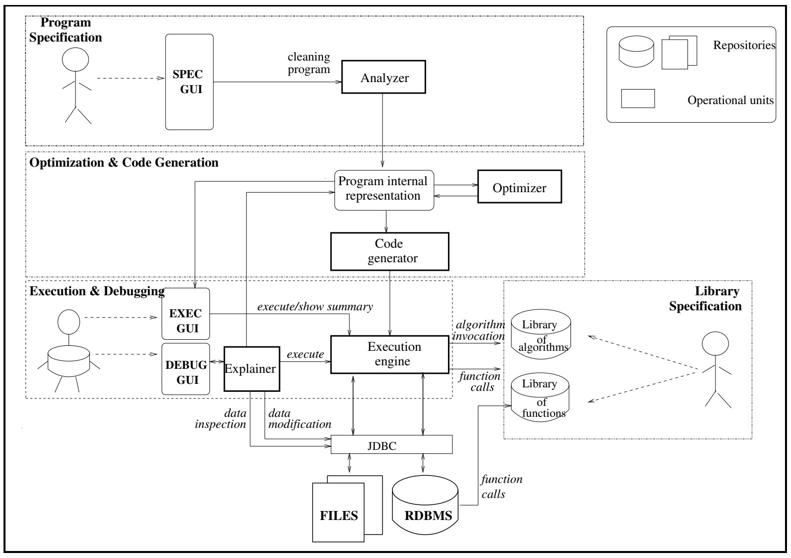

Fig. 4. Architecture of the system.

5.1 Using AJAX

In [11], we present the performance results obtained when using the AJAX system to clean subsets of Citeseer bibliographic references. These results show two kinds of evidence. First, we report the execution times obtained for cleaning three subsets of the Citeseer data set with distinct sizes. We also show the percentage of the execution time devoted to the match operations for each subset. Second, we illustrate the advantage of providing distinct physical execution algorithms for the match operator. We use different execution algorithms for the same logical semantics and we report the execution times and data quality obtained.

More recently, we applied AJAX for specifying and executing a data migration process concerning dam safety information. The main goal here was to map data that obeyed to a given schema into a distinct target data schema. In this real-world application, the exception mechanism was extensively used for detecting input data which was not automatically transformed by the specified data transformation criteria [8].

6 Related work

The first problem with commercial tools is the existence of data transformations whose semantics is defined in terms of their implementation algorithms. To avoid this issue, a data cleaning model must be envisaged to separate logical operations from their physical implementations. There are two important results in the research literature which are concerned with the model and execution of data cleaning transformations. The main goal of the Potter's Wheel prototype [15] developed at the University of California at Berkeley, is to interleave the application of simple logical data transformations to data samples with the detection of data problems. IntelliClean [13] is another data cleaning prototype that offers a way of expressing data transformation rules through an expert system shell. None of these systems is concerned with the independence between logical and physical data cleaning operations. Recently, [16] has proposed a rigorous approach to the problem of optimizing an ETL process defined as a workflow of data transformation activities. The authors model the ETL optimization problem as a global state-space search problem. In our approach, we use local optimization, since an ETL transformation program must be represented by a set of extended relational algebra expressions to be optimized one at a time. Several RDBMSs, like Microsoft SQL Server, already include additional software packages specific for ETL tasks. However, to the best of our knowledge, the capabilities of relational engines, for example, in terms of optimization opportunities are not fully exploited for ETL tasks.

The second open problem in commercial data cleaning tools is the lack of support for user interaction during the execution of a data cleaning application. In fact, the user interaction may be required to debug the results of data transformations, refine the cleaning criteria enclosed, and manually correct data not automatically transformed. There are two important research areas that permit to fulfill this gap. First, the field of data lineage as studied in [4, 5, 2, 19] offers useful notions for browsing the results of data cleaning transformations. Second, the incremental propagation of changes in the context of view maintenance as studied in [6] supplies the basic notions for efficiently integrating data items manually corrected in the flow of data cleaning transformations.

7 Conclusions

In this paper, we provided a global overview of the AJAX system. The description intends to survey all the design and technical aspects of the system and show in which way they constitute a novelty with respect to the existing technology.

The prototype is currently being used in real-world data migration, transformation and cleaning applications so that exhaustive experimental validation can be produced. Moreover, we plan to improve AJAX functionalities according to the requirements of the application scenarios being tested. More concretely, the specification language is being extended, the mechanism of exceptions for the view operator needs to be reformulated, the debugger mechanism needs to be re-designed in order to handle a large amount of exceptions. Finally, some effort has to be put in the design and implementation of a cost-based optimizer and a graphical interface must be constructed to make it easier to specify the cleaning criteria and visualize the results.

References

1. J. Barateiro and H. Galhardas. A survey of data quality tools. *Datenbank Spektrum (invited paper)*, (14):15–21, August 2005.
2. Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and Where: A Characterization of Data Provenance. In *ICDT*, 2001.
3. P. Carreira, H. Galhardas, J. Pereira, and A. Lopes. Data mapper: An operator for expressing one-to-many data transformations. In *DAWAK*, 2005.
4. Yingwei Cui and Jennifer Widom. Practical Lineage Tracing in Data Warehouses. In *ICDE*, 2000.
5. Yingwei Cui and Jennifer Widom. Lineage Tracing for General Data Warehouse Transformations. In *Proc. of VLDB*, 2001.
6. Françoise Fabret. *Optimisation du Calcul Incrémentiel dans les Langages de Règles pour Bases de Données*. PhD thesis, Université de Versailles Saint-Quentin, 1994.
7. Christos Faloutsos, Ron Barber, Myron Flickner, Jim Hafner, Wayne Niblack, Dragutin Petkovic, and William Equit. Efficient and effective querying by image content. *JIIS*, 3(3/4), 1994.
8. H. Galhardas and J. Barateiro. InfoLegada2gB: an application for migrating dam safety information. unpublished.
9. H. Galhardas, D. Florescu, D. Shasha, and E. Simon. Ajax: An extensible data cleaning tool. *ACM SIGMOD Int'l Conf. on Management of Data*, 2(29), 2000.
10. H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C. A. Saita. Declarative data cleaning: Language, model, and algorithms. In *Proc. of the Int'l Conf. on Very Large Data Bases (VLDB'01)*, 2001.
11. Helena Galhardas. *Nettoyage de Données: Modèle, Langage Déclaratif, et Algorithmes*. PhD thesis, Université de Versailles Saint-Quentin, 2001.
12. Helena Galhardas, Daniela Florescu, Dennis Shasha, and Eric Simon. AJAX: An Extensible Data Cleaning Tool. In *SIGMOD (demonstration paper)*, 2000.
13. Mong Li Lee, Tok Wang Ling, and Wai Lup Low. A Knowledge-Based Framework for Intelligent Data Cleaning. *Information Systems Journal - Special Issue on Data Extraction and Cleaning*, 2001.
14. Gonzalo Navarro. A Guided Tour to Approximate String Matching. *ACM Computing Surveys*, 33(1):31–88, March 2001.
15. Vijayshankar Raman and Joseph M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *Proc. of VLDB*, Rome, 2001.
16. A. Simitsis, P. Vassiliadis, and T. K. Sellis. Optimizing ETL processes in data warehouses. In *ICDE*, 2005.
17. T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Theory*, 147:195–197, 1981.
18. Microsoft Research (Sponsored by) NSF, NASA. CiteSeer.IST. http://citeseer.ist.psu.edu/.
19. Allison Woodruff and Michael Stonebraker. Supporting Fine-Grained Data Lineage in a Database Visualization Environment. In *ICDE*, 1997.