Declarative Data Cleaning: Language, Model, and Algorithms

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

Daniela Florescu
Propel, USA
Daniela.Florescu@propel.com

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

Eric Simon
INRIA Rocquencourt, France
Eric.Simon@inria.fr

Cristian-Augustin Saita
INRIA Rocquencourt, France
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. This holds regardless of the application – relational database joining, web-related, or scientific. In all cases, existing ETL (Extraction Transformation Loading) and data cleaning tools for writing data cleaning programs are insufficient. The main challenge is the design and implementation of a data flow graph that effectively and efficiently generates clean data. Needed improvements to the current state of the art include (i) a clear separation between the logical specification of data transformations and their physical implementation (ii) an explanation of the reasoning behind cleaning results, (iii) and interactive facilities to tune a data cleaning program. This paper presents a language, an execution model and algorithms that enable users to express data cleaning specifications declaratively and perform the cleaning efficiently. We use as an example a set of bibliographic references used to construct the Citeseer Web site. The underlying data integration problem is to derive structured and clean textual records so that meaningful queries can be performed. Experimental results report on the assessment of the proposed framework for data cleaning.

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).

To illustrate the difficulty of data cleaning for the Web, we first introduce a concrete running example. The Citeseer Web site (see [13]) 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, and J. Widom:
Making views self-maintanable for data, 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 "main-

---

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

Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001

tainable"). Finally, records may hold different information, e.g., city is missing in the second record.

1.1 Existing technology

The problem of data cleaning is well known [1] for decision support systems and data warehouses. Extraction, Transformation Loading (ETL) tools and data reengineering tools provide powerful software platforms to implement a large data transformation chain, which extracts 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 [18].

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. The difficulty comes from (i) a lack of clear separation between the logical specification of data transformations and their physical implementation, and (ii) the lack of data lineage and user interaction facilities to tune a data cleaning program.

Consider the problem of removing duplicates from the set of author names extracted from the sample Citeseer data set. This problem a priori requires comparing all pairs of author names using an approximate matching function and grouping together all author names that are most probably denoting the same person (e.g., "Dallan Quass" and "D. Quass"). After that, each group can be inspected and a representative (e.g., "Dallan Quass") can be chosen.

Some tools (Integrity [21] and Informix DataCleanser DataBlade module [4]) provide an approximate matching operator. However, this operator is not logical; it consists of a specific optimized algorithm, parameterized with some user-provided criteria, that avoids an exhaustive comparison of all pairs of records. For instance, only those pairs of authors whose last name starts with the same letter will be compared if the user provides a "blocking criteria" based on the initial of the last name. Sometimes, the algorithm implemented by the matching operator is well documented (e.g., multi-pass neighborhood method [11]). In other cases it is either obscure or confidential. This is a problem because the choice of the pairs of records to compare strongly influences the reliability of the result of the matching operation. For instance, it is important to know if successful matches have been lost by the algorithm, when in fact this depends on the mathematical properties of the approximate matching function, and the data manipulated. Even if a matching algorithm is documented, this is not satisfactory because a single non exhaustive matching algorithm cannot fit all situations.

To understand the second difficulty, it is important to realize that the more "dirty" the data, 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. Furthermore, 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.

1.2 Contributions

This paper describes a data cleaning *framework* that attempts to separate the logical and physical levels. The logical level supports the design of the 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. For instance, at the logical level, the matching operator specifies the approximate matching functions used to compare two records, while at the physical level, a specific implementation can be chosen that avoids, say, the evaluation of all record combinations without loosing pertinent combinations. 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. More specifically, this paper presents the following technical contributions that have been implemented in the AJAX system [6]:

- A declarative language for data cleaning, based on five logical data transformation operators. These operators extend the data transformations expressible with SQL99 and can be composed to express all the data transformations from data cleaning we have found in the research literature.
- The semantics of operators integrates the generation of exceptions that provides the foundation for explicit user interaction and the stepwise refinement of data cleaning using a data lineage mechanism.
- A notation to specify the properties of approximate matching functions. These properties enable the system to select an optimized implementation for the matching operation.

The paper is organized as follows. Section 2 gives an overview of our data cleaning framework. Section 3 presents the syntax and semantics of our declarative language for specifying a data cleaning program at the logical level. The fourth section explains the implementation of matching operations while Section

5 reports the experiments done using a Citeseer data set of 500,000 records. Section 6 summarizes other related work and Section 7 concludes.

2 Framework Overview

The development of a data cleaning program actually involves two activities. One is the design of the graph of data transformations that should be applied to the input dirty data. The focus there is to define "quality" heuristics that can achieve the best accuracy (i.e., level of cleaning quality) of the results. A second activity is the design of "performance" heuristics that can improve the execution speed of data transformations without sacrificing accuracy. Our data cleaning framework 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 can be selected to implement the transformations.

2.1 Logical Level

To illustrate our approach, 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 a possible data cleaning strategy 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 operations 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. The design of our logical operators was based on the semantics of SQL primitives that we 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, the computation of the distance between two values, etc. External functions are written in a 3GL programming language and then registered within the library of functions and algorithms of the data cleaning tool.

The semantics of each operator includes the automatic generation of a variety of exceptions that mark tuples which cannot be automatically handled by an operator. This feature is particularly required when dealing with large amounts of dirty data as is usually the case of data cleaning applications. Exceptions may be generated by the external functions called within each operator. If external functions are written in Java, the programmer specifies the generation of exceptions by using the Java mechanism of exceptions. For each exception thrown, the data item that generated it is then stored together with a textual description of the exception. At any stage of execution of a data cleaning program, a data lineage 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. This functionality proved to be essential in our experiments with Citeseer. Details are given in [8].

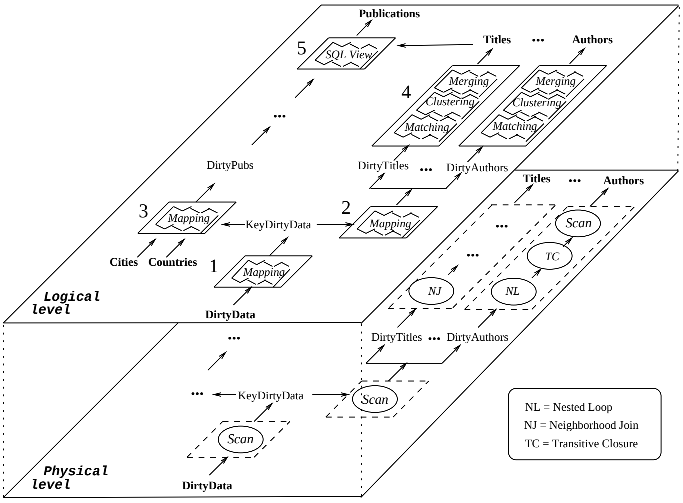

Figure 1: Framework for the bibliographic references

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.

2.2 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 to implement a logical operation among a set of alternative algorithms. As suggested earlier, a very sensitive operator to the choice of execution algorithm is matching. An original contribution of our data cleaning system is to associate with each optimized matching algorithm, the mathematical properties that the distance function used in the matching 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 an algorithm to implement a matching. 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 1 shows the algorithms selected to implement each logical operation.

3 Specification Language

This section gives a presentation by example of the five logical operators (mapping, view, matching, clustering, and merging) offered by our declarative language for expressing data cleaning transformations. A formal description of our operators and the BNF grammar for their syntax can be found in [7]. In [7], we also show how to express data cleaning transformations existing in commercial systems or introduced in the research literature as a composition of our operators.

3.1 Mapping Operator

A mapping operator takes a single relation as input and produces one or more relations. It expresses arbitrary one-to-many database mappings. That is, each tuple from the input relation can generate zero or more tuples into the output relations, independently or not from the other tuples of the input relation.

Example 3.1: The following mapping operator transforms each tuple of relation DirtyData{paper} into a tuple of relation KeyDirtyData{paperkey, paper} by adding a serial number to it. This transformation corresponds to Step 1 of Figure 1.

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

Example 3.1 illustrates the syntax of a mapping operator. 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. Then, the **let** keyword introduces a let-clause which is a sequence of *assignment statements*.

Example 3.2: The mapping command below transforms KeyDirtyData defined above into four target relations. 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(AuthorTitleEvent.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}
```

In each assignment statement, a relation is assigned a functional-expression that involves the invocation of an external function (previously registered into the library of external functions of the data cleaning system). If the functional-expression returns an atomic value, the assignment is an *atomic assignment statement*. The let-clause in Example 3.1 contains an atomic assignment statement that assigns a relation Key using an external (atomic) function generateKey that takes as argument a variable DirtyData.paper ranging over attribute paper of DirtyData. If the functional-expression returns a table then the assignment is a *table assignment statement*. The let-clause in Example 3.2 contains two table assignment statements. In the first one, relation AuthorTitleEvent is assigned an external function directly, whereas in the second one relation AuthId is assigned an SQL **select from where** expression that makes use of a previously defined relation.

We explain the semantics of an assignment statement using the statement of Example 3.1. For every tuple, noted DirtyData(a)[1], in DirtyData, 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 $\mathsf{DirtyData}^{exc}$(a) is added to relation $\mathsf{DirtyData}^{exc}$. In which case, no tuple will be generated in relation Key for tuple DirtyData(a). We shall say that this statement *defines* a relation Key{paper, generateKey}[2]. Note that the input relation of a map-

---

[1] Where a denotes 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.

ping can also be used as an argument of an external function in an assignment statement. This means that the result of the external function for a given tuple of the input relation may depend on the other tuples of the relation. This provides a more general usage of external functions than SQL currently does and makes the mapping operator very expressive. The semantics of a table assignment statement is similar except that each tuple from the input relation may generate a set of tuples into the relation assigned in the statement.

The **where** keyword introduces a filter expressed as a conjunctive normal form in a syntax similar to an SQL where-clause. This filter can reference any attribute from the input relation or the relations defined by the let-clause. Finally, the schema of the output relations is specified by one or more **select into** expressions, called *output clauses*, that specify the schema of each output relation, and its associated constraints. For instance, the "{ SELECT Key.generateKey AS ...}" clause in Example 3.1 indicates that the schema of KeyDirtyData is built using the attributes of Key and DirtyData. Output 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. For instance, in Example 3.2, if for a given input tuple, one of the output constraints is violated, the execution of the mapping does not stop but instead the input tuple is added to relation $\mathsf{KeyDirtyData}^{exc}$.

We now explain the semantics of the mapping operator[3]. First, the assignment statements that compose a let-clause are evaluated in their order of appearance in the let-clause. This completely defines the instances of the relations assigned in the let-clause. Then, the filter specified by the where-clause is evaluated to generate a relation, say U, resulting from the cartesian product of all the relations defined by the let-clause followed by the elimination of all the tuples of U that do not satisfy the filter. Next, relation U is used to construct the instance of each output relation (by projecting on its attributes). Finally, for each output relation, the constraints are checked for each tuple.

3.2 View Operator

The view operator merely corresponds to an SQL query, augmented with some integrity checking over its result. As such, it can express limited many-to-one mappings (those expressible in SQL), whereby each tuple in the output relation results from some combination of tuples taken from the input relations. However, the difference in semantics with a regular SQL query is the management of exceptions that can be generated by the constraints in the output-clause.

Example 3.3: The following example specifies the view operation that aggregates together the data that result from the extraction of volume, number, year, etc of each citation with the corresponding titles and events, free of duplicates (step 5 in Figure 1).

```
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 }
```

3.3 Matching Operator

A matching operator computes an approximate join between two relations. More specifically, it computes a distance value for each pair of tuples in the Cartesian product of the two input relations using an arbitrary distance function. This operator is fundamental in all data cleaning. It could be expressed as an SQL query that includes a call to an external function, and thus as a view operation. However, as we shall see later, a matching operation can be implemented by different kinds of specialized algorithms, but recognizing when to use such algorithms in the general syntax of an SQL query would be quite difficult. On the contrary, making it a first class operator facilitates its optimization.

Example 3.4 illustrates a matching operation. The let-clause has the same meaning as in a mapping 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[4] 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 output relation (here, MatchAuthors) whose schema is the same as distance.

Example 3.4: This (self-)matching operator takes as input the relation DirtyAuthors{authorKey, name} twice. Its intention is to find possible duplicates within DirtyAuthors.

```
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 only subtlety in the SQL-like syntax of the matching operator is the possible use of the symbol "+" following a relation name in the **from** clause (it could

---

[3] Note that this does not define how a mapping operator is actually implemented.

[4] When E is defined using an atomic function foo, we abusively allow to use expression E as a shorthand of foo

be "a1 +" in the above example). It indicates that a relation, called DirtyAuthors$^{no-match}$, containing all the pairs of records of DirtyAuthors that did not match (i.e., that do not appear in the output relation) will also be returned by the operator. Exceptions can be thrown by the evaluation of the external functions in the let-clause.

3.4 Clustering Operator

A clustering operation takes a single input relation that defines a set of elements and returns a single output relation that groups the elements into a set of clusters. Conceptually, the output relation is a nested relation wherein each inner relation corresponds to a cluster. There are two ways of specifying the set of elements defined by the input relation. First, each tuple of the input relation can define an element. In this case, each element of a cluster will be identified by its tuple identifier in the input relation. A typical example of a clustering operation in this case is an SQL group-by query that groups together all the tuples of a relation that are identical on some attribute values. However, we allow arbitrary clustering operations that are more general than the SQL group-by.

As a second possibility, each tuple of the input relation can define a distance between two elements each of which being identified by a specific attribute of the input relation, called key attribute. In this case, the clustering operation will be applied to the union of all the elements found in the pairs defined by each tuple of the input relation. Each element of a cluster is identified by its corresponding key attribute value in the input relation. Example 3.5 illustrates the clustering operation in this case.

Example 3.5: Consider the relation MatchAuthors generated by the matching operation of Example 3.4. We specify a clustering operation over MatchAuthors where each cluster consists of a set of DirtyAuthors tuples that are sufficiently close to each other (they probably correspond to the same author). The clustering method views each tuple of MatchAuthors as a binary relationship between DirtyAuthors tuples, and groups in the same cluster all tuples that are transitively connected.

```
CREATE CLUSTERING clusterAuthorsByTranstiveClosure
FROM MatchAuthors
ON authorKey1, authorKey2
BY METHOD transitive closure
WITH PARAMETERS authorKey1, authorKey2
INTO clusterAuthors
```

The **on** clause specifies how the input relation defines the set of elements to be clustered. If this clause is omitted then each tuple of the input relation defines an element. Otherwise, the attributes corresponding to the identifiers of the first and second components of the pairs of elements defined by the tuples of the input relation are specified in the clause. The attributes of the input relation used by the clustering algorithm are specified through a **with parameters** clause. Other clustering methods require more parameters to be passed in the with parameters clause; for instance, if we use a "nearest-neighbor" clustering method, an additional parameter is the maximum distance from the centroid of the cluster. Many clustering methods exist (see e.g., [14]), each providing certain properties to the clusters they produce. For instance, the transitive closure method of Example 3.5 produces disjoint clusters. Unlike the other operators, the clustering operator does not generate exceptions.

The result of the clustering operation is a relation that always has one attribute, named cluster_id, plus additional attributes determined as follows. If each tuple of the input relation defines a pair of elements with a distance value, then the output relation has two additional attributes each of which corresponding to the identifier of an element in the first or second component of those pairs. Otherwise, each element of a cluster identifies a tuple in the input relation and there is one additional attribute in the output relation corresponding to a tuple identifier. In our example, the output of the clustering operation over MatchAuthors is a relation with three attributes, one for each of the two DirtyAuthors relations, and one cluster_id attribute. Thus, clusterAuthors has for schema {cluster_id, authorKey1, authorKey2}.

Example 3.6: 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, we would have the following tuples in the output relation, say clusterAuthors (not all tuples are listed below):

```
clusterAuthors: 1 | 1 | null
                1 | 6 | null
                1 | 7 | null
                1 | null | 1
                1 | null | 6 ...
                2 | 2 | null
                2 | 10 | null ...
```

3.5 Merging Operator

The merging operation takes a single relation as input and returns one relation as output. It partitions the input relation according to some grouping attributes and collapses each partition into a single tuple using an arbitrary aggregation function. This operator is not expressible in SQL99 because it requires the possibility of having user defined aggregation functions.

Example 3.7: 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 an output 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}.

Example 3.8 describes this merging operation. The **using** clause is similar to a **from** clause with the following differences. Until now, a **let** clause was always defined wrt the relation(s) indicated in the **from**. In the case of merging, the **let** clause is defined wrt the grouping attribute(s) of the relation indicated in the **using** clause. 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 (that have been previously grouped by cluster_id). The let-clause is used to construct the attribute values that will compose each tuple over the target relation.

Example 3.8:

```
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 }
```

We use a specific notation to ease 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 and S_2. Let A be an attribute of S_1. Then, if p is a variable ranging over the attribute domain cluster_id of P, the expression S_1(p).A refers to the set of tuples: {x.A | x is a tuple over S_1 and the identifier of x belongs to cluster p}. In the example above, ca is a variable ranging over the cluster_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 name with maximum length belonging to the same cluster. In general, the evaluation of the merging operator generates exceptional tuples whenever exceptions are thrown in the let-clause or some output constraint is violated.

4 Implementation of Matching

Our data cleaning system takes a specification of a data cleaning program expressed in the declarative language and generates a Java program, in which each operator's specification is translated into a Java class. Several important optimization decisions are made during the code generation. In this section, we focus on the implementation of matching.

4.1 Optimization problem

A matching 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 matching is the Cartesian product between the two input relations. There are two main kinds of optimizations that enable to reduce this cost. The first one 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. A second type of optimization is to use an approximate method that compares a limited number of records with a good expected probability that most candidate matches will be returned.

4.2 Distance-filtering optimization

This type of optimization has been successfully used for image retrieval [5]. Formally, the result of a matching 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:

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

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

$$\forall x, \forall y, \; dist'(f(x), f(y)) \leq dist(x,y) \quad (2)$$

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:

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

```
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 $f$
  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 $\cup$ $(s_1, s_2)$ }}}}
}
```

Figure 2: Neighborhood Join algorithm

A generic algorithm that implements this optimization is shown in Figure 2. This algorithm, called *Neighborhood Join* or NJ for short, is effective when 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.

This optimization is illustrated below on a matching operation of the Citeseer data cleaning program that takes as input the relation DirtyTitles{pubKey, title, eventKey} twice, and is specified as shown below (the line between the %'s is explained later)[5]. We assume that maxDist is an integer. The editDistanceTitles function is based on the Damerau-Levenshtein metric [20] that returns the number of insertions, deletions and substitutions needed to transform one string into the other.

Example 4.1:

```
CREATE MATCHING 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 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 $\mathsf{abs}(x, y) = |x - y|$.

A key feature of our data cleaning framework is to enable the specification of the properties of the distance function that can be used to optimize the execution of the matching as annotations in the create matching clause. The above example shows the annotation (between the %'s) for the distance filtering property[6]. Here, the type of property is specified as well as the mapping and distance functions, whose code must be provided to the system. These annotations are used by the data cleaning system to guide the code generation for an operation. The next section shows the effectiveness of this kind of optimization on the Citeseer example.

4.3 Approximate methods

A virtue of the distance filtering optimization is that, as a consequence of Equation (2), it does not allow false dismissals. The optimization presented here optimizes the computation of candidate matches at the risk of losing candidate matches. A good representative of this type of method is the multi-pass neighborhood method (MPN) proposed in [11].

The MPN method consists of repeating the two following steps after performing the outer-union of the input relations: 1) choose a key (consisting of one or several attributes, or substrings within the attributes) for each record and sort records accordingly; 2) compare those records that are close to each other within a fixed, usually small, sized window. The criteria for comparing records is defined by a distance function encoded in a proprietary programming language (for instance, "C") [12]. Each execution of the two previous steps (each time with a different key) produces a set of pairs of matching records (i.e., candidate matches). Formally, suppose that there are n records to be matched and the size of the window is p, $p \leq n$. Then $(n - p) + 1$ windows will be defined over the n sorted records. Each window, noted W_i, contains the list of records from record i to record $i + p$. Thus, for a single pass, given $O_i = \{(x, y, dist(x, y)) \mid x \in W_i, y \in W_i, dist(x, y) \leq \epsilon\}$, the result of MPN is: $\cup_{i=1}^{(n-p)+1} O_i$.

When there are several passes, a final transitive closure is applied to all the pairs of records that have been returned, yielding a union of all pairs generated by all independent passes, plus all those pairs that can be inferred by transitivity of equality (on record ids).

An annotation for MPN within the matching of Example 4.1 could be the following: %MPN: key = title; window size = 100%. Note that only the parameters of the physical algorithm are specified in the absence of particular mathematical properties of the algorithm.

5 Experiments

The purpose of our experiments was to assess the possibility of specifying implementations of matching using annotations in the logical specification of matching. We ran our experiments on a single-CPU Pentium III workstation with a i686 CPU at 501MHz, cache size 512KB, and 1G bytes of RAM having Linux as its operating system. We used ORACLE8i as our database system and JDK1.3 to execute the Java code.

We compared the performance of three distinct physical algorithms for the matching operator: nested loop (NL), that corresponds to the naive semantics of the matching; multi-pass neighborhood method (MPN) and neighborhood join (NJ). The last two were presented in Section 4 and correspond to optimizations of the first algorithm. All algorithms were implemented in Java[7]. Since the MPN method computes an approximation of the candidate matches defined by a matching operation, we used a measure of quality, called *recall*, defined as the number of matches returned by the algorithm divided by the number of candidate matches.

We applied these algorithms to the matching of authors, events, and publications for two subsets of dirty

---

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

[6] This obviously supposes that when the NJ algorithm was registered within the data cleaning system, the parameters map and dist required by the algorithm were declared to the system.

[7] A fourth physical algorithm where the optimization capabilities of an RDBMS could be used to execute the matching operation was also considered. It was not taken into account in the measurements since most RDBMS do not optimize approximate joins.

bibliographic references with respective cardinalities of 100,000 records (15MB), and 500,000 records (86MB). The matching of authors (resp. events) computes the pairs of duplicate author names (resp. event names), while the matching of publications considers two publications as duplicates if their titles are close enough and the corresponding events are equal (this is an extension of the MatchDirtyTitles matching presented in example 4.1). All matching operations used a matching criteria based on the Damerau-Levenshtein distance.

We compared the results of MPN and NJ. The MPN method has two parameters: the sort keys and a window size. The algorithm is executed as many times as the number of sort keys. The sort keys tested were the following: LTR and RTL - sort entries by author name and alphabetic order from left to right and from right to left, respectively; S - sorts entries by author name without blank spaces; and L - sort entries by author's last name. We combined these keys in order to run the algorithm for 1, 2 and 4 passes. The window sizes tested were: 100; 1,000; and 10,000. The NJ used the distance filtering presented in section 4 that is defined by: map=length; dist=abs[8]. By definition, the NJ algorithm has always a recall of 1.

Table 1 shows the results obtained[9]. The columns named MPN refer to the multi-pass neighborhood algorithm and the last column, named NJ refers to the neighborhood algorithm. Let us pick the fourth line of the table . The first MPN(a) column contains the minimum execution time (T = 4.9) to obtain a recall (R = 0.806) superior to 0.8. The second MPN(a) column registers the window size (WS = 100) and the number of passes (NP = 2) needed to obtain such time and recall values. The two MPN(b) columns report the same measures (T = 745.0, R = 0.987) and parameters (WS = 10,000; NP = 4) for obtaining a maximum recall value. The last column represents the execution time (T = 790.0) of the NJ algorithm (recall = 1).

Comparing the time/recall columns, we conclude that in general, MPN can be faster but less accurate than NJ. If execution time is crucial and accuracy can be neglected to a certain level, MPN is worthwhile; otherwise NJ is the best choice. The differences observed depend on the domain of data the matching is applied to. For event names, whose values are strongly unstructured (even if a normalization against a dictionary has been applied), the NJ algorithm is able to achieve a recall of 1 much faster than the MPN method; this difference is less remarkable for author names. Thus, the results of this experiment confirm the usefulness of providing more than one physical implementation for the same matching operator.

6 Other Related Work

Related work falls into three main categories: high level languages to express data transformations, data cleaning frameworks, and algorithms to support matching, clustering and merging operations.

Several languages have been recently proposed to express data transformations: SQL99 [10], WHIRL's SQL [2], and SchemaSQL [15]. Our language supports operations such as clustering and merging that are not expressible in SQL99. Furthermore, in SQL the occurrence of an exception immediately stops the execution of a query. In contrast, our semantics enables to compute the entire set of tuples that caused the occurrence of exceptions. Last, unlike SQL, our language enables the optimization of a matching operation by making it a first-citizen operator in the language. WHIRL's SQL extends SQL queries with a special join operator that uses a similarity comparison function based on the vector-space model commonly adopted in statistical information retrieval. This join operator is a special case of our matching operator. Furthermore, WHIRL does not support clustering and merging operations. SchemaSQL is a powerful extension of SQL that includes operations to restructure a relational schema. This language is useful to perform integration queries in relational multi-database systems. It is complementary to our language in the sense that we do not allow schema restructuring operations involving meta-data whereas SchemaSQL does not allow neither arbitrary clustering and merging operations nor optimized matching.

Several frameworks have been proposed for data integration and cleaning. We already compared our work to commercial ETL and data cleaning tools. Research prototypes include IntelliClean [16] and Potter's Wheel-ABC [19]. Lee et al [16] propose a rule-based approach to express matching, clustering and merging operations, which is implemented using the Java Expert System Shell. However, their framework provides a fixed matching algorithm (the MPN method) and the approach does not clearly scale up for very large data sets due to the use of an expert system shell. Like us, Potter's Wheel promotes an interactive approach whereby users are able to apply a set of simple transformations to modify samples of data and see the results interactively. However, unlike us, their focus is on interleaving data transformation and discrepancy detection to facilitate the refinement of data transformations. Their techniques for automatic discrepancy detection that run as a background process behind the data transformation could be applied to our context. The transforms supplied by Potter's Wheel fall into two categories: one-to-one and many-to-many map-

---

[8] The selectivity of the filter (percentage of comparisons computed) was: 11% for events (filter based on the length of the event name); 30% for authors (filter based on the length of the last name and on the length of the entire name without blank spaces); and 16% for publications (filter based on the length of the title).

[9] The execution times obtained for the NL execution (for 100,000 tuples) were 30, 180 and 4 minutes for Authors, Events and Publications, respectively. The execution times obtained using the NJ (as shown in table 1) for the same subsets of data correspond to gains of 33%, 72%, and 68% respectively.

Dirty tuples	**Matching**	**MPN(a)**:T/R	**MPN(a)**:WS/NP	**MPN(b)**:T/R	**MPN(b)**:WS/NP	**NJ**:T/R
100,000	Authors	0.4/0.915	100/1	58.95/0.998	10,000/2	25.8
100,000	Events	6.9/0.846	100/2	398.1/0.999	10,000/2	66.0
100,000	Publications	0.17/0.887	100/1	6.27/0.999	10,000/2	1.7
500,000	Authors	4.9/0.806	100/2	745.0/0.987	10,000/4	790.0
500,000	Events	454.5/0.862	1,000/2	3607.0/0.975	10,000/2	2555.0
500,000	Publications	0.73/0.861	100/1	77.0/0.996	10,000/2	63.0

Table 1: Implementation of the matching using MPN and NJ. T/R is the execution time/recall ratio. WS is the window size, and NP is the number of passes for the MPN method.

pings of rows. The first set of transforms correspond to dispatchers according to the classification presented in [3] so they can be expressed as mapping operators by our framework, as we detail in [7]. The second set of transforms encloses the *fold* and *unfold* transforms. If these operations manipulate only values of columns, they correspond to our mapping and merging operators. However, if as in SchemaSQL, they exchange data values with metadata values (column names), they are not expressible in our framework.

Finally, several algorithms have been proposed to implement matching, clustering, and merging operations. The MPN algorithm [11] described earlier has been further optimized in [17] using a tighter integration of the matching and clustering phases. [9] proposes an approximate string join algorithm using the distance filtering optimization we presented in Section 4, and implemented on top of an RDBMS.

7 Conclusions

We presented a data cleaning framework whose main originality is a separation in two clear layers: logical and physical. The main features of the framework described in this paper are: *(i)* a declarative language to specify the flow of logical transformations; *(ii)* a declarative specification of user interaction based on the automatic generation of exceptions during operator's execution, and *(iii)* a declarative way to select an optimized implementation for the matching operator. Our experiments showed that the separation between the logical specification of a matching and its implementation gives the proper control to the user of the tradeoff that arises between performance and recall.

Acknowledgments

The authors would like to thank Judy Cushing and Ioana Manolescu for having reviewed a previous version of this paper.

References

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

[2] W. Cohen. Integration of Heterogeneous Databases without Common Domains Using Queries based on Textual Similarity. In *Proc. of ACM SIGMOD*, 1998.

[3] Y. Cui and J. Widom. Lineage Tracing for General Data Warehouse Transformations. In *Proc. of VLDB*, 2001.

[4] EDD. Home page of DataCleanser DataBlade Module. http://www.npsa.com/edd/.

[5] C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equit. Efficient and effective querying by image content. *JIIS*, 3(3/4), 1994.

[6] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. AJAX: An Extensible Data Cleaning Tool. In *SIGMOD (demonstration paper)*, 2000.

[7] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Declarative Data Cleaning: Language, Model, and Algorithms. Extended version of the VLDB'01 paper, 2001.

[8] H. Galhardas, D. Florescu, D. Shasha, E. Simon, and C.-A. Saita. Improving data cleaning qaulity using a data lineage facility. In *Workshop on Design and Management of Data Warehouses (DMDW)*, Interlaken, Switzerland, June 2001.

[9] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate String Joins in a Database (Almost) for Free. In *Proc. of VLDB*, Rome, September 2001.

[10] P. Gulutzan and T. Pelzer. *SQL-99 Complete, Really.* R&D Books, 1999.

[11] M. A. Hernandez and S. J. Stolfo. The Merge/Purge problem for large databases. In *Proc. of ACM SIGMOD*, 1995.

[12] M. A. Hernandez and S. J. Stolfo. Real-world data is dirty: Data Cleansing and the Merge/Purge problem. *Journal of Data Mining and Knowledge Discovery*, 2(1):9–37, 1998.

[13] N. R. Institute. Research Index (CiteSeer). http://citeseer.nj.nec.com/.

[14] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall Advanced Reference Series, 1988.

[15] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian. SchemaSQL - A Language for Interoperability in Relational Multi-database Systems. In *Proc. of VLDB*, Mumbai, 1999.

[16] M. L. Lee, T. W. Ling, and W. L. Low. A Knowledge-Based Framework for Intelligent Data Cleaning. *Information Systems Journal - Special Issue on Data Extraction and Cleaning*, 2001.

[17] A. Monge. Matching Algorithms within a Duplicate Detection System. *IEEE Data Engineering Bulletin*, 23(4), December 2000.

[18] E. Rahm and H. H. Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Engineering Bulletin*, 23(3), September 2000.

[19] V. Raman and J. M. Hellerstein. Potter's Wheel: An Interactive Data Cleaning System. In *Proc. of VLDB*, Rome, 2001.

[20] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Theory*, 147:195–197, 1981.

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