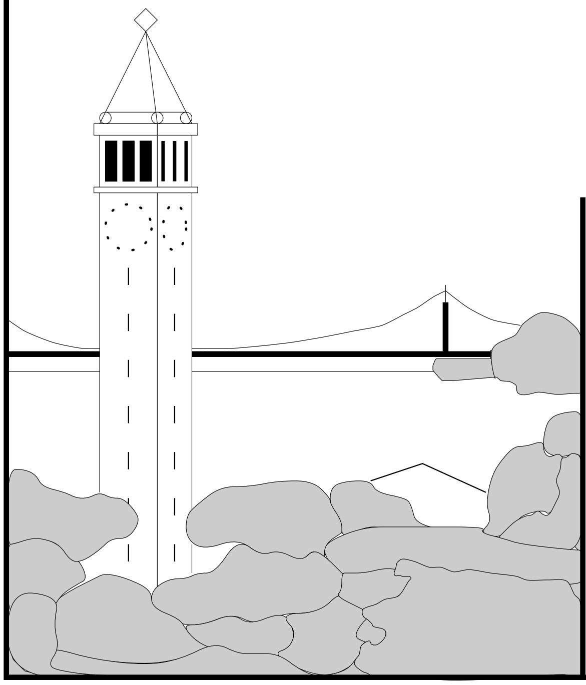
An Interactive Framework for Data Cleaning

Vijayshankar Raman   *Joseph M. Hellerstein*



Report No. UCB/CSD-0-1110

September 2000

Computer Science Division (EECS)
University of California
Berkeley, California 94720

An Interactive Framework for Data Transformation and Cleaning

Abstract

Cleaning organizational data of discrepancies in structure and content is important for data warehousing and Enterprise Data Integration. Current commercial solutions for data cleaning involve many iterations of time-consuming "auditing" to find errors, and long-running transformations to fix them. Users need to endure long waits and often write complex transformation programs. We present an interactive framework for data cleaning that tightly integrates transformation and discrepancy detection. Users gradually build transformations by adding or undoing transforms, in a intuitive, graphical manner through a spreadsheet-like interface; the effect of a transform is shown at once on records visible on screen. In the background, the system automatically infers the structure of the data in terms of user-defined domains and applies suitable algorithms to check it for discrepancies, flagging them as they are found. This allows users to gradually construct a transformation as discrepancies are found, and clean the data without writing complex programs or enduring long delays.

We choose and adapt a small set of transforms from existing literature and describe methods for their graphical specification and interactive application. We apply the Minimum Description Length principle to automatically extract the structure of data values in terms of user-defined domains. Such structure extraction is also applied in the graphical specification of transforms, to infer transforms from examples. We also describe methods for optimizing the final sequence of transforms for memory allocations and copies. This transformation facility is integrated into a spreadsheet-based data analysis package, allowing flexible analysis of arbitrarily transformed versions of the data.

1 Introduction

Organizations accumulate much data from their businesses that they want to access and analyze a consolidated whole. However the data often has inconsistencies in schema, formats, and adherence to constraints, due to many factors like merging from multiple sources and entry errors [7, 28, 19]. The data must be purged of such discrepancies and transformed into a uniform format before it can be used. Such *data cleaning* is one of the key challenges in data warehousing [7, 19]. Data transformation is also needed for extracting data from legacy data formats, and for Business-to-Business Enterprise Data Integration where two different organizations want to access each other's data and need it to be in a common format [16]. In this paper, we present Potter's Wheel[1], an interactive framework for data cleaning and transformation. Before that we briefly discuss current data cleaning technology to provide context.

1.1 Current Approaches to Data Cleaning

Data cleaning has 3 components: auditing data to find discrepancies, choosing transformations to fix these, and applying them on the data set. There are currently many commercial solutions for data cleaning (*e.g.* see [17]). They come in two forms: auditing tools and transformation tools. The user first audits the data to detect discrepancies in it using an auditing tool like ACR/Data or Migration Architect [2, 18]. Then she either writes a custom script or uses an "ETL" (Extraction/Transformation/Loading) tool like Data Stage or CoSort [15, 11] to transform the data, fixing errors and converting it to the format needed for analysis.

The data often has many hard-to-find special cases, so this process must be repeated until the "data quality" is good enough. As we see later, the data often has *nested discrepancies* that can be found only after others have been resolved – again necessitating more iterations. The individual iterations of this process involve running outlier detection algorithms or transformation algorithms that are typically linear or even super-linear in the data sizes (*e.g.* [33, 28, 5, 35]), and so are frustratingly slow on large datasets.

[1] Our technique for cleaning data resembles that of a potter molding clay on a wheel. The potter incrementally shapes clay by applying pressure at a point, just as the user incrementally constructs transformations by applying transforms on example rows.

1.2 Potter's Wheel Approach

There is no single magic program to automate data cleaning. A variety of techniques for transformation or discrepancy detection may be applicable in different circumstances, and the system should support them flexibly. The system architecture must also be carefully designed with the human-computer interaction in mind; human input is essential in the audit/transform loop, to act on discrepancies and select transformations.

In this paper, we present *Potter's Wheel*, an interactive framework for data cleaning that integrates transformation and discrepancy detection in a tight, closed loop. Users gradually build transformations by composing and debugging transforms[2], one step at a time, on a spreadsheet-like interface (see Figure 1; the details will be explained in later sections). Transforms are specified graphically, their effect is shown immediately on records visible on screen, and they can be undone easily if their effects are undesirable. Discrepancy detection is done automatically in the background, *on the latest transformed view of the data*, and anomalies are flagged as they are found.

The pipelining of transformation and discrepancy detection makes data cleaning a tight, closed loop where users can gradually develop and refine transformations as discrepancies are found[3]. This is in contrast to the current commercial data cleaning model where transformation and discrepancy detection are done as separate steps, often using separate software[4]. Hence users have to wait for a transformation to finish before they can check if it has fixed all anomalies. More importantly, some *nested discrepancies* can be resolved only after others have been resolved. For example, a typo in year such as "19997" can be found only after all dates have been converted to a uniform format of month, date, and year. Thus decoupling transformation and discrepancy detection makes it hard to find multiple discrepancies in one pass, leading to many unnecessary iterations.

Data transformation and analysis typically go hand-in-hand. First, the data needs to be transformed before analysis to clean it of errors. Moreover many analysis algorithms operate on certain portions of the data, or require it to be in particular formats. For example, an analyst may want to compute correlations against the year field of the date attribute, or a particular charting utility may want the input values to be in a particular form. The traditional decoupling of transformation and analysis tools hinders their pipelining and creates unnecessary delays, even if the operations are independently interactive. Hence Potter's Wheel integrates seamlessly into a spreadsheet-based data analysis framework as we describe in Section 2.5.

1.3 Transforms: Graphical Specification, Interactive Application, and Optimization

Interactive transformation has two aspects – the transforms must be easy to specify graphically, and they must be applied interactively, with immediate feedback to the user so that they can correct any errors.

Commercial ETL tools typically support only some restricted conversions between a small set of formats via a GUI, and provide ad hoc programming interfaces for general transformations (these are essentially libraries of conversions between standard formats: *e.g.* Data Builder, Data Junction's CDI SDK and DJXL [13, 14]). This hinders interactive transformation because errors in a program are not caught until the entire dataset has been transformed and rechecked for discrepancies. Moreover, it is often difficult or even impossible to write "compensatory scripts" to undo an erroneous transformation, forcing users to maintain and track multiple versions of potentially large datasets.

We have adapted from the research literature on transformation languages (*e.g.*, [36, 8, 33]) a small set of transforms that support many common transformations without explicit programming. Most of these are natural to specify

[2] We use transform as a noun to denote a single operation, and transformation as a noun to denote a sequence of operations.

[3] Incremental detection and transformation may lead to cascading changes and obscure the data lineage [52], if one transform causes an error elsewhere. Note however that it is precisely in such situations that an interactive, undoable way of transformation is needed. With Potter's Wheel the user can quickly find out that a transform was inappropriate and make amends, as we describe in Section 4.4. Whereas with a traditional tool she will not find out the problem with a transformation until the very end.

[4] Even vendors like Ardent that provide both ETL and "quality analysis" software provide them as two pieces of a suite; the user is expected to do discrepancy detection and transformation in separate stages [15].

graphically. However some transforms used to split values into atomic components are quite complex to specify. These transforms are often needed for parsing structures of values in "wrapper-generation" or "screen-scraping" tools (like Araneus [22] or Cohera Net Query [10]). Such tools typically require users to specify regular expressions or grammars to parse values. As we see in Section 4.3, even these are often inadequate and users have to write custom scripts. Such complex specification interfaces hinder interactive transformation; they are time-consuming and error-prone, and impose delays that distract the user from the task of cleaning. In contrast Potter's Wheel allows users to enter the desired results on example data, and automatically infers a suitable transform. This is performed using the structure extraction techniques described below under discrepancy detection. We describe such graphical specification, and the interactive application of these transforms, in Section 4.

This interactive approach also improves accuracy of transformations because users can easily experiment with different transforms. They can graphically specify or undo transforms, and need not wait for the entire dataset to be transformed to learn what effect a transform has had.

Potter's Wheel compiles the entire sequence of transforms into a program after the user is satisfied, instead of applying them piecemeal over many iterations. Users often specify/undo transforms in an order intuitive to them, resulting in unnecessary or sub-optimal transformations. The main cost in executing these transforms in a program is the CPU time needed for memory allocation and memory copies. Hence the final sequence of transforms can be optimized, collapsing redundant transforms and optimally pipelining them to minimize memory allocations and copies. We present such optimizations and preliminary performance results in Section 5.

1.4 Automatic Discrepancy Detection

There are many techniques for detecting whether a value violates the constraints of its domain. Generic values with no specific domain can be checked using outlier detection algorithms like [43, 5]. However many values belong to specific domains and require to be checked using custom techniques. These could be standard ones like spell-checking, address/zip code verification, or specialized one like finding errors in chemical formulae. Certain domains may impose other constraints likes uniqueness or functional dependencies that need special algorithms (*e.g.* [28, 30]).

We want to handle all such domains in an extensible fashion in Potter's Wheel. It is simple to allow users to define custom domains, along with verfication algorithms for these domains. However the data is often a composite structure containing parts from different domains, like *"Rebecca by Daphne du Maurier, et. al. Hardcover (April 8, 1948) $22.00"*[5] or *"Wal-Mart, 7401 Samuell Blvd, Dallas, TX 75228-6166 Phone: (214)319-2616"*[6]. Therefore the system must automatically parse a value into a structure composed of user-defined domains, and then apply suitable discrepancy detection algorithms. This is similar to the XML DTD inference problem addressed in [21]. However, unlike in XML DTDs, the domains are not just regular expressions but could be arbitrary inclusion functions. Moreover the data will have errors, both in structure and in adherence to the domain constraints. In Section 3.2 we describe how Potter's Wheel applies the Minimum Description Length principle to parse values in terms of user-defined domains.

1.5 Outline

We describe the architecture of Potter's Wheel in Section 2. We discuss how discrepancies are detected using user-defined domains in Section 3. In Section 4, we describe how Potter's Wheel supports interactive transformation. In Section 5 we present initial work on optimizing sequences of transforms for memory accesses. We look at related work in Section 6 and conclude with directions for future work in Section 7.

[5] From the web page of Amazon.com search results for Daphne Du Maurier

[6] From the web page of SwitchBoard.com search results for Walmart in Dallas

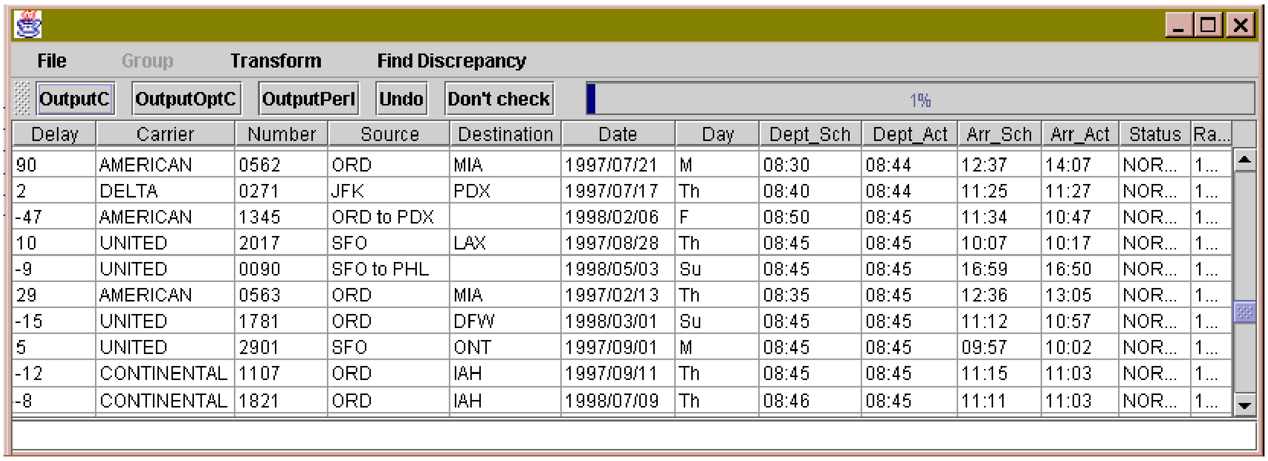


Figure 1: A snapshot of the Potter's Wheel User Interface on flight delay data from FEDSTATS [20]. More detailed screenshots are available at the software web page [41].

2 Potter's Wheel Architecture

The main components of Potter's Wheel architecture (Figure 2) are a *Data Source*, a *Transformation Engine* that applies transforms along 2 paths, an *Online Reorderer* to support interactive scrolling and sorting at the user interface [42, 40], and an *Automatic Discrepancy Detector*. We proceed to discuss these in turn.

2.1 Data Source

Potter's Wheel accepts input data as a single, merged input stream. This input can come from an ODBC source or an ASCII file source. The ODBC source can be used to access data from relational databases via SQL queries, or even from more complex sources via middleware (*e.g.* [50, 27, 6]). Clearly, schematic differences between sources will restrict the tightness of the integration via a query, as we will see in Section 4. Even Figure 1 shows poor mapping in the *Source* and *Destination* columns. Potter's Wheel will find areas of poor integration as discrepancies, and the user can transform the data, moving data values across columns to unify the data format.

When accessing records from ASCII files, each record is viewed as a single large column. The user can identify column delimiters graphically and split the record into constituent columns. Such parsing is especially necessary for unstructured data (such as from web pages). In this context parsing is traditionally a time-consuming and laborious process, also called "wrapping" or "screen-scraping". Potter's Wheel tackles this problem through a `Split` transform that can be specified by example (described in Section 4). Alternately, column types and delimiters can also be specified in a metadata file. If the user has parsed and converted a dataset once, they can store it as a macro for easy application on other similar datasets (Section 5).

2.2 Interface used for Displaying Data

Data that is read in from the input stream is displayed using a Scalable Spreadsheet [40] interface that allows users to interactively re-sort on any column, and scroll in a representative sample of the data, even over large datasets. This interface appears immediately, and the user does not have to wait until the data has been completely fetched from the source. This is an important requirement for interactive behavior over large datasets or never-ending data streams (such as sensor feeds).

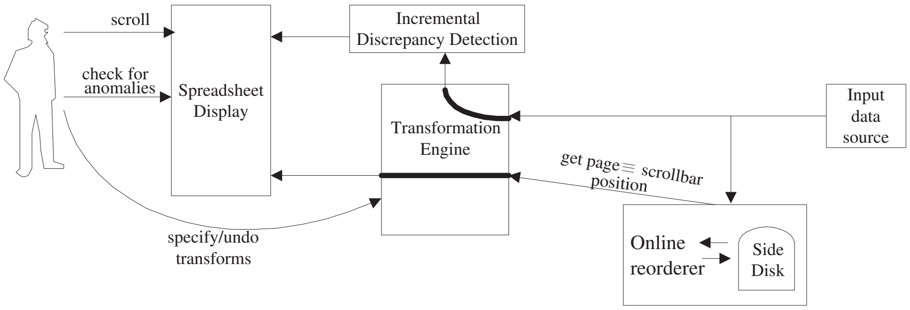


Figure 2: Potter's Wheel Framework

The interface supports this behavior using an Online Reorderer [42] that continually fetches tuples from the source and divides them into buckets based on a (dynamically computed) histogram on the sort column, spooling them to a side-disk if needed. When the user scrolls to a new region, the reorderer picks a sample of tuples from the bucket corresponding to the scrollbar position and displays them on the screen. Thus users can explore large amounts of data along any dimension. Such exploration is an important component of the analysis package that Potter's Wheel is integrated with (Section 2.5). We also believe that exploration helps users spot simple discrepancies by observing the structure of data values as a dimension changes.

2.3 Transformation Engine

Users specify transforms in a direct-manipulation fashion on this spreadsheet interface, by choosing appropriate columns or rows and selecting transforms, or by showing the desired effect on example values. We describe this further in Section 4.3. These transforms need to be applied in two places. First, they need to be applied to records visible on screen. With the spreadsheet user interface this is done when the user scrolls or jumps to a new scrollbar position. Since the number of rows that can be displayed on screen at a time is small, users perceive transformations as being instantaneous (this clearly depends on the nature of the transforms; we return to this issue in Section 4.2). Second, transforms need to be applied to records used for discrepancy detection because, as argued earlier, we want to check for discrepancies on transformed versions of data. Collections of transforms can also be compiled into an optimized program, as we describe in Section 5.

2.4 Automatic Discrepancy Detector

While the user is specifying transforms and exploring the data, the discrepancy detector applies appropriate algorithms to find errors in the data. Hence tuples fetched from the source are transformed and sent to the discrepancy detector, in addition to being sent to the Online Reorderer. The discrepancy detector decides on suitable algorithms for each field of the tuple based on its inferred structure. The strucutre of a field is inferred as soon as it is formed (*i.e.*, either when the input stream is started or when a new column is formed by a transform), as we describe in Section 3.2.

2.5 Integration into analysis framework

The spreadsheet interface used for transformation serves to show examples of the latest transformed data values; the users can explore these examples by sorting and scrolling along arbitrary dimensions. Users can also compute aggregates and plot histograms on this transformed data in an online, continually refining fashion, and can recursively partition it using any clustering algorithm, as described in [40].

```
public abstract class Domain {
    /** Required Inclusion Function — Checks if value satisfies domain constraints. */
    public abstract boolean match(char *value);

    /** Optional function – finds the number of values in this domain with given length. This could vary
        based on parameterization – see Section 3.3.*/
    public int cardinality(int length);

    /** Optional function – updates any state for this domain using the given value */
    public void updateStats(char* value);

    /** Optional function – gets the probability that all discrepancies have been found. Typically needs to know
        the total number of tuples in the data set (e.g. see [30]). */
    public float confidence(int dataSize);

    /** Optional function – checks if one pattern is redundant after another */
    public boolean isRedundantAfter(Domain d);
}
```

Figure 3: API for user-defined domains

3 Framework for Discrepancy Detection

As outlined in the introduction, we would like to handle in an extensible fashion any discrepancy detection algorithm, whether generic or domain specific. We want users to be able to define arbitrary *domains* along with discrepancy detection algorithms for those domains.

We describe the API for domains and discrepancy detection algorithms in Section 3.1. As argued before, column values are typically composite structures, and the system needs to infer the appropriate structure for its values in terms of these domains. We explain how this is done in Section 3.2. Some of the domains in this structure may need to be *parameterized* for the speific values (*e.g.* an integer domain can be parameterized with the mean and standard deviation so that it can track anomalous values). This is discussed further in Section 3.3. Once this detailed structure is inferred, the system parses values and sends individual components to suitable discrepancy detection algorithms, as described in Section 2.4.

3.1 Domains in Potter's Wheel

Domains in Potter's Wheel are defined through the interface shown in Figure 3. The only required function to implement is an inclusion function `match` to identify values in the domain. The optional `cardinality` function is helpful in structure extraction. `updateStats` is mainly used to parameterize the domains (Section 3.3). It can also be used by a discrepancy detection algorithm to accumulate state about the data. This accumulated state can be used to catch *multi-row* anomalies where a set of values are individually correct, but together violate some constraint. For example, a duplicate elimination algorithm could use `updateStats` to build an approximate hash table or Bloom filter of the values seen so far. The `confidence` method is helpful for probabilistic and incremental discrepancy detection algorithms, such as sampling based algorithms (e.g. [30]). The `isRedundantAfter` method is used in enumerating structures, as described in Section 3.2.

Potter's Wheel provides the following default domains: arbitrary ASCII strings (henceforth called ξ^*)[7] , char-

[7] In the rest of this paper we use ξ to refer to the alphabet of all printable ASCII characters.

acter strings (*Words*; likewise *AllCapsWords* and *CapWords* refer to words with all capitals and capitalized words), *Integers*, sequences of *Punctuation*, C-style *Identifiers*, IEEE floating points (henceforth called *Decimals*), English words checked according to ispell (*IspellWords*), common *Names* (checked by referring to the online 1990 census results [48]), *money*, and a generic regular-expression domain that checks values using the PCRE library [26].

3.2 Structure Extraction

A given value will typically be parseable in terms of the default and user-defined domains in multiple ways. For example, *"March 17, 2000"* can be parsed as ξ^*, as [A-Za-z]* [0-9]*, [0-9]*, or as [Ma-h]* [17]*, [20]*, to name a few. Structure extraction involves choosing the best structure for values in a column. Formally, given a set of column values $v_1, v_2, \ldots, v_n$ and a set of domains $d_1, d_2, \ldots, d_m$, we want to extract a structure $S = d_{s_1} d_{s_2} \ldots d_{s_p}$, where $1 \le s_1 \ldots s_p \le m$.

Recently Garofalakis *et. al.* [21] addressed a variant of this problem for XML DTD inference using the using the minimum description length (MDL) principle. They choose the best regular expression that matches a set of values. Whereas we need to infer structures involving arbitrary domains that are specified only as abstract inclusion functions. Moreover in our case the data will have errors, not only in adherence to the domain constraints, but also in the very structure. Hence we can only try to get an approximate structure. We first describe how to evaluate the appropriateness of a structure for a set of values and then describe ways of enumerating all structures so as to choose the best one.

Evaluating the Suitability of a Structure

There are three characteristics that we want in a structure for the column values.

Recall: The structure must match as many of the values as possible.

Precision: The structure must match as few other random values as possible.

Conciseness: The structure must have minimum length.

The first two criteria are standard IR metrics for evaluating the effectiveness of a pattern [49]. We need to consider recall because the values might be erroneous even in structure; all unmatched values are considered as discrepancies. Considering precision helps us avoid overly broad structures like ξ^* that do not uniquely match this column.

The last criterion of conciseness is used to avoid over-fitting the structure to the example values. For instance, we want to parse *March 17, 2000* as [A-Za-z]* [0-9]*, [0-9]* rather than as M a r c h 1 7 , 2 0 0 0. Note that for conciseness to be helpful the alphabet from which we create the structure must contain the user-defined domains. For instance if we did not have words and integers as domains in the alphabet, M a r c h 1 7 , 2 0 0 0 would be the better structure since it has the same recall (100%), better precision (since it avoids matching any other date), and smaller pattern length than [A-Za-z]* [0-9]*, [0-9]*. Intuitively, the latter is a more concise pattern, but this is only because we think of the word domain [A-Za-z]* as a single element *"Word"* in the alphabet.

These three criteria are typically conflicting, with broad patterns like ξ^* having high recall and conciseness but low precision, and specific patterns having high precision but low conciseness. An effective way to balance the tradeoff between over-fitting and under-fitting is through the MDL principle [44, 12]. MDL tries to minimize the total length required to encode the data using the structure. It has been used effectively in many applications like learning decision trees [39].

The description length of a structure for a set of patterns is defined as the length of theory for the structure plus the length required to encode the values given the structure. We need this to encapsulate Recall, Precision, and Conciseness. Conciseness is directly captured by the length of theory for the structure. For values that match the structure, the length required for encoding the data values captures the Precision. We tackle erroneous data values by positing that values not matching the structure are encoded explicitly by writing them out, i.e. using the structure ξ^*. The latter encoding is typically more space-intensive since it assumes no structure. Thereby we capture Recall.

```
/** Enumerate all structures of the domains d_{s_1} ... d_{s_p} that can be used to match a value v_i. */
void enumerate(v_i , d_1,...d_p) {
    Let v_i be a string of characters w_1 ... w_m
    for all domains d that match a prefix w_1...w_k of v_i do
        do enumerate(w_{k+1} ... w_m , d_{s_1},... d_{s_p}), avoiding all structures that begin with
          a domain d' that satisfies d'.isRedundantAfter(d)
        prepend d to all structures enumerated in the previous step
}
```

Figure 4: Enumerating different structures for a set of values

For example, consider a structure of *<Word> <Integer> <Integer>* and a value of *May 17 2025*. The length needed for encoding the structure is $3 \log(\text{number of domains})$. Then we encode the value by first specifying the length of each sub-components and then, for each component, specifying the actual value from all values of the same length. Thus the description length is $3 \log(\text{number of domains}) + 3 \log(\text{maximum length of values in each sub-component}) + 3 \log 52 + 2 \log 10 + 4 \log 10$.

In the above example, we are able to calculate the lengths of the value encodings for integers and words because we know that the domains are strings over an alphabet of 10 numerals and 52 letters, respectively. However computing the lengths of the encodings for arbitrary domains is harder.

Consider a structure $S = d_{s_1} d_{s_2} \ldots d_{s_p}$. Let $|T|$ denote the cardinality of any set T. The description length of a string v_i of length $len(v_i)$ using it is:

$$MDL(v_i, S) = \text{length of theory for } S \ + \ \text{length to encode } \ v_i \text{ given } S$$

Since the number of domains is m, we can represent each domain with $\log m$ bits. Let f be the probability that v_i matches the structure S. We have,

$$\begin{aligned} MDL(v_i, S) &= p \log m + f(\text{space to express } v_i \text{ using } S) \ + \ (1-f)(\text{space to express } v_i \text{ from scratch}) \\ &= p \log m + f(\text{space to express } v_i \text{ using } S) + (1-f)(\log |\xi^{len(v_i)}|) \end{aligned}$$

Let $AvgValLen = \sum_{1 \le i \le n} len(v_i)$ be the average length of the values in the column. The average space needed to encode the values in the column is,

$$= p \log m + f(\text{average space to express values } v_1 \ldots v_n \text{ using } S) \ + \ (1-f)(\log |\xi^{AvgValLen}|)$$

Just as in the example, we calculate the space required to express the values v_i using S by first encoding the lengths of its components in each domain and then encoding the actual values of the components. For any string w that falls in a domain d_u, let $len(w)$ be its length, and let $sp(w|d_u)$ be the space required to uniquely encode w among all the $len(w)$-length strings in d_u. Suppose that value v_i matches $S = d_{s_1} d_{s_2} \ldots d_{s_p}$ through the concatenation of sub-components $v_i = w_{i,1} w_{i,2} \ldots w_{i,p}$, with $w_{i,j} \in d_{s_j}$ $\forall 1 \le j \le p$. Let $MaxLen$ be the maximum length of the values in the column. Then the average space required to encode the values in the column is,

$$p \log m + (f/n) \times \sum_{i=1}^{n} \left(p \log MaxLen + \sum_{j=1}^{p} sp(w_{i,j}|d_{s_j}) \right) \ + \ (1-f)(\log |\xi^{AvgValLen}|) =$$

$$p \log m + AvgValLen \log |\xi| \ + \ fp \log MaxLen + \left(\frac{f}{n}\right) \sum_{i=1}^{n} \sum_{j=1}^{p} \log \frac{|\text{values of length } len(w_{i,j}) \text{ that satisfy } d_{s_j}|}{|\text{values of length } len(w_{i.j})|}$$

The best way to compute the cardinality in the above expression is using the `int cardinality(int length)` function for the domain d_{s_j}, if it has been defined. For other domains we compute the fraction directly by repeatedly choosing random strings of appropriate length and checking if they satisfy d_{s_j}. Note that these fractions are independent of the actual data values, and so can be pre-computed and cached for typical lengths. If the length is too high we

Example Column Value (Erroneous values in parentheses)	Num. Structures Enumerated	Final Structure Chosen (Num = Number, Punc = Punctuation)
-60	5	Integer
UNITED, DELTA, AMERICAN etc.	5	IspellWord
SFO, LAX etc. (some values like JFK to OAK)	12	AllCapsWord
1998/01/12	9	Num(len 4) Punc(/) Num(len 2) Punc(/) Num(len 2)
M, Tu, Thu etc.	5	Capitalized Word
06:22	5	Num(len 2) Punc(:) Num(len 2)
12.8.15.147 (some values like ferret03.webtop.com)	9	*Double Punc('.') Double*
"GET\b (some values like \b)	5	Punc(") IspellWord Punc(\)
/postmodern/lecs/xia/sld013.htm or /telegraph/	4	ξ^*
HTTP	3	AllCapsWord(HTTP)
/1.0	6	Punc (/) Double (1.0)

Table 1: Structures extracted for different kinds of columns. We only show sample values from each column. The domains we used are the default domains listed in Section 3.1. Structure parameterizations are given in parenthesis.

may need to check many values before we can estimate this fraction. Hence we compute the fraction of matches for a few small lengths, and extrapolate it assuming that the number of matches is a simple exponential function of the length. This works well for "regular" domains like identifiers or integers where the number of values of a given length is exponential in the length, but does not work well for domains like English words.

Choosing the best structure

We have seen how to evaluate the suitability of a structure for a given set of values. We want enumerate all structures that can match the values in a column and choose the most suitable one. This enumeration needs to be done carefully since since the structures are arbitrary strings from the alphabet of domains.

We apply the algorithm given in Figure 4 on a set of sample values from the column, and take the union of all the structures enumerated thus. We use 100 values as a default; this is adequate in all the cases that we have seen. During this enumeration, we prune the extent of recursion by not handling structures with certain meaningless combinations of domains such as <*word*> <*word*> or <*integer*> <*decimal*>. These are unnecessarily complicated versions of simpler structures like <*word*> and <*decimal*>, and will result in structures with identical precision and recall but lesser conciseness. We identify such unnecessary sequences using the `boolean isRedundantAfter(Domain d)` method of `Domain` that determines whether this domain is redundant immediately after the given domain.

After such pruning the number of structures we enumerate for a column reduces considerably, and is typically less than 10. Table 1 shows the number of structures enumerated for some example columns.

3.3 Structures with Parameterized Domains

So far the structures that we have extracted are simply strings of domains. But the column values are often much more restricted, consisting only of certain parameterizations of the domains. For example, all the sub-components from a domain might have a constant value, or might be of constant length, as shown in the structures of Figure 1.

Potter's Wheel currently detects two parameterizations automatically: domains with constant values and domains with values of constant length. Such parameterized structures are especially useful for automatically parsing the values in a column, which is used to infer splits by example in Section 4.3.

In addition, users can define domains that infer custom parameterizations, using the `updateStats` method. These apply custom algorithms to further refine the structure of the sub-components that fall within their domain. For example, a domain can accept all strings by default, but parameterize itself by inferring a regular expression that matches the sub-component values. An algorithm for this is given in [21].

The description length for values using a structure changes when the structure is parameterized. For the default parameterizations of constant values and constant lengths it is easy to adjust the formulas from the previous section. For custom parameterizations like the regular expression inference discussed above, the user must appropriately adjust the cardinality using `updateStats`.

3.4 Example Structures Extracted and Discrepancies Found

Consider the table shown in Figure 1 containing flight delay statistics. Table 1 shows the structures extracted for some of its column values, and also for some columns from a web access log. More example structures are given in Figure 9. In many cases we see that the dominant structure is chosen even in the face of inconsistencies; thereby the system can flag these structural inconsistencies as such and apply suitable algorithms for the others.

Using these the system flags several discrepancies that we had earlier added to the data. For example, the system flags dates such as 19998/05/31 in the date column as anomalies because the Integer domain for the year column parameterizes with a mean of 2043.5 and a standard deviation of 909.2. It finds the poor mapping in the Source and Destination columns as structural anomalies.

We also see that system occasionally chooses inappropiate structures. For example, values like 12.8.15.147 are chosen as *Decimal.Decimal*. This arises because *Decimal* is a more concise structure that *Integer.Integer*. This could be avoided either by defining a *Short* domain for values less than 255, or even by allowing a parameterization of the form *Integer(len* $\leq$ *3)*. An interesting example of over-fitting is the choice of *IspellWord* for flight carriers. Although most flight carrier names occur in the ispell dictionary, some like TWA do not. Still *IspellWord* is chosen because it is cheaper to encode TWA explicitly with a ξ^* structure than to encode all carriers with a *AllCapsWord* structure. As a result the system flags TWA as an anomaly. The user could either choose to ignore this anomaly, or specify a minimum Recall threshold to avoid over-fitting. In any case, this example highlights the importance of involving the user in the data cleaning process.

4 Interactive Transformation

We have adapted from existing literature on transformation languages (*e.g.* [33, 8]) a small set of transforms. Our focus is on the important goal of interactive transformation – we want users to construct transformations gradually, adjusting them based on continual feedback. This goal breaks up into the following requirements.

Ease of specification: We want users to specify transforms through graphical operations; writing code for the transformation is time-consuming and error-prone, and distracts users from the main job of identifying discrepancies and choosing transforms to fix them. The traditional literature on transformation has relied on declarative specification languages that are powerful but rather hard to specify (Section 4.3. Often users need to supply regular expressions or even grammars for performing transforms. Instead transforms should be specifiable through intuitive GUI based operations, or at worst by outlining the desired effect on example values.

Ease of interactive application: Once the user has specified a transform, they must be given immediate feedback on the results of its application so that they can correct or augment it. We describe how transforms are applied

Transform	Definition		
`Format`	$\phi(R, i, f)$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, f(a_i)) \mid (a_1, \ldots, a_n) \in R\}$
`Add`	$\alpha(R, x)$	$=$	$\{(a_1, \ldots, a_n, x) \mid (a_1, \ldots, a_n) \in R\}$
`Drop`	$\pi(R, i)$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n) \mid (a_1, \ldots, a_n) \in R\}$
`Copy`	$\kappa((a_1, \ldots, a_n), i)$	$=$	$\{(a_1, \ldots, a_n, a_i) \mid (a_1, \ldots, a_n) \in R\}$
`Merge`	$\mu((a_1, \ldots, a_n), i, j, \text{glue})$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_{j-1}, a_{j+1}, \ldots, a_n, a_i \oplus \text{glue} \oplus a_j) \mid (a_1, \ldots, a_n) \in R\}$
`Split`	$\omega((a_1, \ldots, a_n), i, \text{splitter})$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, \text{left}(a_i, \text{splitter}), \text{right}(a_i, \text{splitter})) \mid (a_1, \ldots, a_n) \in R\}$
`Divide`	$\delta((a_1, \ldots, a_n), i, \text{pred})$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, a_i, \texttt{null}) \mid (a_1, \ldots, a_n) \in R \wedge \text{pred}(a_i)\} \cup$ $\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, \text{null}, a_i) \mid (a_1, \ldots, a_n) \in R \wedge \neg\text{pred}(a_i)\}$
`Fold`	$\lambda(R, i_1, i_2, \ldots i_k)$	$=$	$\{(a_1, \ldots, a_{i_1-1}, a_{i_1+1}, \ldots, a_{i_2-1}, a_{i_2+1}, \ldots, a_{i_k-1}, a_{i_k+1}, \ldots, a_n, a_{i_l}) \mid$ $(a_1, \ldots, a_n) \in R \wedge 1 \leq l \leq k\}$
`Select`	$\sigma(R, \text{pred})$	$=$	$\{(a_1, \ldots, a_n) \mid (a_1, \ldots, a_n) \in R \wedge \text{pred}((a_1, \ldots, a_n))\}$

Notation: R is a relation with n columns. i, j are column indices and a_i represents the value of a column in a row. x and glue are values. f is a function mapping values to values. $x \oplus y$ concatenates x and y. splitter is a position in a string or a regular expression, $\texttt{left}(x, \text{splitter})$ is the left part of x after splitting by splitter. pred is a function returning a boolean.

Table 2: Definitions of the various transforms. `Unfold` is defined in Appendix A.

incrementally in Section 4.2.

Undos and Data Lineage: After the user sees feedback on a transform they may want to replace it with a better one. Hence the system must support undos of transforms. Moreover, incremental transformation creates a data lineage problem [52] where one transform creates other cascading discrepancies. The user must be able to identify discrepancies that were originally present in the data from ones resulting from other transformations. We describe how Potter's Wheel supports undos and data lineage in Section 4.4.

4.1 Transforms supported in Potters Wheel

The transforms used in Potter's Wheel are adapted from existing literature on transformation languages (*e.g.* [33, 8]). We describe them briefly here before proceeding to discuss their interactive application and graphical specification. Readers interested in more illustrative examples can refer to [41]. Formal definitions and proofs of expressive power are given in Table 2 and Appendix B respectively.

One-to-one Mappings of Rows

The most basic transforms operate on individual rows. The simplest is `Format` that applies a function to the value of a column in every row. We provide default functions for common operations like regular-expression based substitutions and arithmetic operations, but also allow user defined functions. Column and table names can be *demoted* into column values using special characters in regular expressions; these are useful in conjunction with the `Fold` transform described below.

The remaining one-to-one transforms perform column operations that can be used to unify data collected from different sources into a common format, as illustrated in Figures 5 and 6.

`Drop`, `Copy`, and `Add` transforms allow users to drop or copy a column, or add a new column. While `Add`ing a new column, its value can be set to a constant, a random number, or a serial that provides unique values.

The `Merge` transform concatenates values in two columns, optionally interposing a constant (the delimiter) in the middle, to form a single new column. `Split` splits a column into two or more parts, and is used typically to parse a value into its constituent components. The splitting positions are often difficult to specify if the data is not well structured or has errors. We allow splitting by specifying character positions, regular expressions, and even by performing splits on example values. We elaborate on this in Section 4.3.

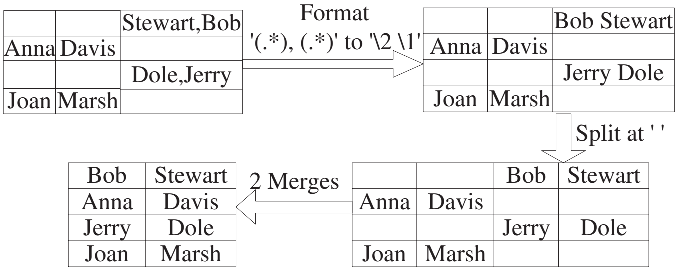


Figure 5: Using `Format`, `Merge` and `Split` to clean name format differences

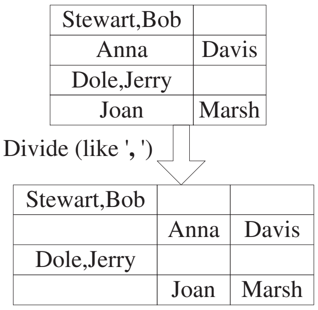


Figure 6: Using `Divide` to separate different name formats

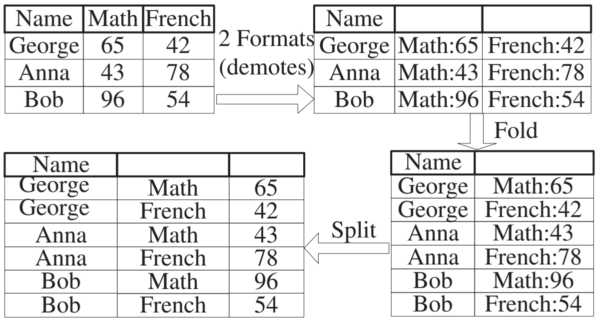


Figure 7: `Fold` to fix higher-order differences

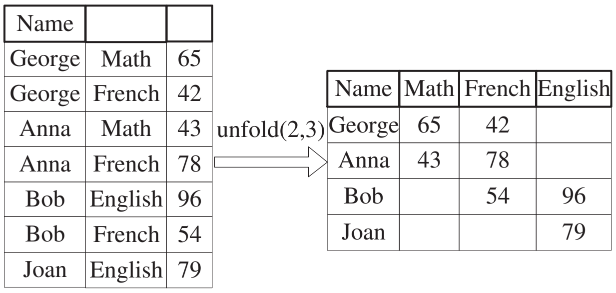


Figure 8: `Unfold`-ing into three columns

Occasionally, logically different values (maybe from multiple sources) are bunched into the same column. The `Divide` transform performs a conditional division of a column sending values into one of two new columns based on a predicate, as shown in Figure 6. This is useful if we want a certain transform to applied only on particular values.

Many-to-Many Mappings of Rows

Many-to-Many transforms help to tackle higher-order *schematic heterogeneities* [37] where information is stored partly in data values, and partly in the schema, as shown in Figure 7.

The `Fold` transform converts one row into multiple rows, folding a set of columns together into one column and replicating the rest, as defined in Table 2. Conversely `Unfold` takes *two* columns, collects rows that have the same values for all the other columns, and unfolds the two chosen columns. Figure 7 shows an example with student grades where the subject names are demoted into the row via `Format`, grades are `Fold`ed together, and then `Split` to separate the subject from the grade. `Unfold` is used to "unflatten" tables and move information from data values to column names, as shown in Figure 8. Values in one column are used as column names to align the values in the other column.

These two transforms are adapted from the Fold and UnFold restructuring operators of SchemaSQL [32]. However, unlike in SchemaSQL, we do not automatically demote column names in `Fold` because often there is no meaningful column name. More detailed discussion of the differences with SchemaSQL, along with formal definitions of `Fold` and `UnFold`, are given in Appendix A.

Power of Transforms

As we prove in Appendix B, these transforms can be used to perform all one-to-many mapping of rows. Some transforms like `Divide`, `Add` and `Drop` are in theory obviated by `Merge`, `Split` and UDF-based `Formats`. However they help to specify many operations more naturally, as we will see in Section 4.3. Moreover specifying transforms directly via these operations permits many performance optimizations in their application that would be impossible if they were done opaquely as a UDF-based `Format` (Section 5).

`Fold` and `Unfold` are essentially the same as the restructuring operators of SchemaSQL, and can also be used to *flatten* tables, converting it to a form where the column and table names are all literals and do not contain data values. For a formal definition of (un)flattening and an analysis of the power of `Fold` and `Unfold`, see [32, 23].

As future work we plan to study the power of these transforms for for converting data between self-describing and flat formats. For example, a row with multiple columns can be converted into a nested structure by demoting the column names as tags (as in Figure 7), unfolding set-valued attributes distributed in multiple rows, and merging attributes that are in separate columns.

4.2 Interactive Application of Transforms

A simple way of giving immediate feedback on the transformations is to apply the transforms incrementally as tuples stream in. This allows us to immediately show the results of the transforms on tuples visible on the screen of the spreadsheet interface. Also this lets the system pipeline discrepancy detection on the results of the transforms, thereby checking if additional transforms need to be applied to fix any uncaught discrepancies.

Among the transforms discussed above, all the one-to-one transforms as well as the `Fold` transform are functions on a single row. Hence they are easy to apply interactively. As discussed in Section 2, these transforms can be applied in a pipelined fashion on the input stream before sending it for discrepancy detection, and also on the records currently shown on the screen.

However the `Unfold` transform operates on a set of rows with matching values, which could potentially involve scanning the entire data. Hence in our current implementation we do not allow `Unfold` to be specified graphically at the user interface. For displaying records on the screen of the spreadsheet we can avoid this problem by not displaying a complete row but instead displaying more and more columns as distinct values are found, and filling data values in these columns as the corresponding input rows are read. However, progressively adding more and more columns in the spreadsheet interface could confuse the user. This problem of asynchronous column addition can be avoided by implementing an abstraction interface (column roll up) where, upon an `Unfold`, all the newly created columns are shown as one rolled up column. When the user clicks to unroll the column it expands into a set of columns corresponding to the distinct values found so far.

However, pipelining discrepancy detection on the unfolded results is much harder. We don't even know what or how many columns are going to be there in the output – this depends on the number of distinct values there are in the *Unfolding* column. As described in Section 3, the system analyzes the structure of each column and chooses appropriate discrepancy detection algorithms for it. Hence discrepancy detection on the unfolded results will have to wait until a sizable number of values have been formed in each result column. We plan to address interactive discrepancy detection over unfolded results in future work.

4.3 Graphical Specification of Transforms

As mentioned before, `Divide`, `Add`, and `Drop` are in theory redundant, but they help to specify transforms more naturally. For instance, Drop-ing a column is much simpler than Merge-ing it with another column and then Format-ing it to remove the unnecessary part (Appendix B expands on this idea).

Many of the transforms described earlier, such as `Add`, `Drop`, `Copy`, `Fold`, and `Merge` are simple to specify graphically. Users can highlight the desired columns and choose the appropriate transform. Some transforms like `Format` and `Divide` need textual input in the form of arithmetic expressions or predicates, but this input is natural and unavoidable.

<table>
<tr><th>Example Values Split By User
(| is user specified split position)</th><th>Inferred Structure</th><th>Comments</th></tr>
<tr><td>Thatcher, Margaret |, £52,072
Major, John |, £73,238
Tony Blair |, £1,00,533</td><td>$(<\xi^*>< \text{',' money}>)$</td><td>Parsing is possible despite no good delimiter. A regular expression domain can infer a structure of [0-9,]* for the last component.</td></tr>
<tr><td>MAA |to| SIN
JFK |to| SFO
LAX |–| ORD
SEA |/| OAK</td><td>$(<\text{len 3 identifier}><\xi^*><\text{len 3 identifier}>)$</td><td>Parsing is possible despite multiple delimiters.</td></tr>
<tr><td>321 Blake #7 |, Berkeley |, CA 94720
719 MLK Road |, Fremont |, CA 95743</td><td>$(<\text{number } \xi^*><\text{',' word}><\text{','}\ (\text{2 letter word})\ (\text{5 letter integer})>)$</td><td>Parsing is easy because of consistent delimiter.</td></tr>
</table>

Figure 9: Parse structures inferred from various split-by-examples

However the `Split` transform is often hard to specify precisely. Splits are often needed to parse values in a column into consituent parts, as illustrated in Figure 9. This is an important problem for commercial integration products. Tools like Microsoft Excel automatically parse standard structures like comma-separated values. There are many research and commercial "wrapper-generation" tools (e.g. Araneus [22], Cohera Net Query(CNQ) [10], and Nodose [4]) that try to tackle the general problem for "screen-scraping" unstructured data found on web pages. However these tools often require sophisticated specification of the split, ranging from regular expression split delimiters to context free grammars. But even these cannot always be used to split unambiguously. For instance in the first entry of Figure 9, commas occur both in delimiters and in the data values. As as result, users often have to write custom scripts to parse the data.

4.3.1 Split by Example

In Potter's Wheel we aim that users should not have to worry about specification of complex regular expressions or programs to parse values. We want to allow users to specify most splits by performing them on example values.

The user selects a few example values $v_1, v_2, \ldots, v_n$ and in a graphical, direct-manipulation [46] way shows how these are to be split, into components $(w_{1,1}w_{1,2}\ldots w_{1,m}), (w_{2,1}\ldots w_{2,m}), \ldots, (w_{n,1}\ldots w_{n,m})$. As done during discrepancy detection, the system infers a structure for each of the m new columns using MDL, and uses these structures to split the rest of the values. These structures are general, ranging from simple ones like constant delimiters or constant length delimiters, to structures involving parameterized user-defined domains like "<IspellWord>–++<x[ab]*>–<Word(len 3)>". Therefore they are better than simple regular expressions at identifying split positions. Figure 9 contains some sample structures that Potter's Wheel extracts from example splits on different datasets. We see that even for the ambiguous-delimiter case described earlier, Potter's Wheel extracts good structures that can be used to split unambiguosly.

Some values may still not be unambiguously parseable using the inferred structures. Other values may be anomalous and not match the inferred structure at all. We flag all such values as discrepancies to the user.

Splitting based on inferred structures

Since the structures inferred are more general than just regular expressions, splitting a value based on these is not easy. Figure 10 shows a naive algorithm for parsing a value that considers the inferred structures from left to right, and tries

```
/** Split a value v_i using the structures S_1, S_2, ... S_k */
void split(v_i, S_1, ... S_k) {
    Let v_i be a string of characters w_1 ... w_m
    for all prefixes w_1...w_j of v_i that satisfy S_1 do
        split(w_{j+1} ... w_m, S_2 S_3 ... S_k)
    If the previous step did not return exactly one way of splitting, report v_i as a possible discrepancy
}
```

Figure 10: Naive method of splitting a value using the inferred structures.

```
/** Split a value v_i using the structures S_1, S_2, ... S_k */
void split(v_i, S_1, ... S_k) {
    Let v_i be a string of characters w_1 ... w_m
    Let v_1, v_2, ... v_n be the example values used to infer the structures for the split
    Let the user-specified split for each value v_i be x_{i,1} x_{i,2} ... x_{i,k}
    As in Section 3.2, compute for all structures S_j the space required to express the values
   x_{1,j}, x_{2,j}, ... x_{n,j} using S_j. Call this sp_j.
   Choose the structure S_j with the least value of X_j.
    for all substrings w_a...w_b of v_i that satisfy S_j do
        split(w_1 ... w_{a-1}, S_1 ldots S_{j-1})
        split(w_{b+1} ... w_m, S_{j+1} ldots S_p)
    If the previous step did not return exactly one way of splitting, report v_i as a possible discrepancy
}
```

Figure 11: Efficient method of splitting a value using the inferred structures. Considers the structures in decreasing order of specificity.

to match them against all prefixes of unparsed value. This exhaustive algorithm is very expensive for “imprecise” structures that match many prefixes. Quick parsing is particularly needed when the split is to be applied on a large dataset after the user has chosen the needed sequence of transforms.

Therefore we use an alternative algorithm (Figure 11) that matches the inferred structures in decreasing order of specificity. It first tries to find a match for the most specific structure, and then recursively tries to match the remaining part of the data value against the other structures. The idea is that in the initial stages there will be few alternative prefixes for matching the structures, and so the value will be quickly broken down into smaller pieces that can be parsed later. The specificity of a structure is computed as the description lengths of the (appropriate substrings) of the example values using the structure. The value is quickly divided into many smaller substrings because the specific structures such as delimiters are caught quickly. Hence this algorithm is more efficient.

4.4 Undoing Transforms and Tracking Data Lineage

One way of supporting undos of transfoms would be to perform “compensating transforms”. However this would result in unnecessary transformation and then untransformation of every record. Moreover many transforms (such as regular-expression-based substitutions and some arithmetic expressions) cannot be undone unambiguously. Undoing these requires the system to maintain multiple versions of potentially large datasets.

Therefore Potter’s Wheel adopts a strategy of never changing the actual data records during the transformation. It merely collects transforms as the user adds them, and applies them only on the records displayed on the screen, in essence showing a view using the transforms specified so far. Undos are performed by simply removing the concerned

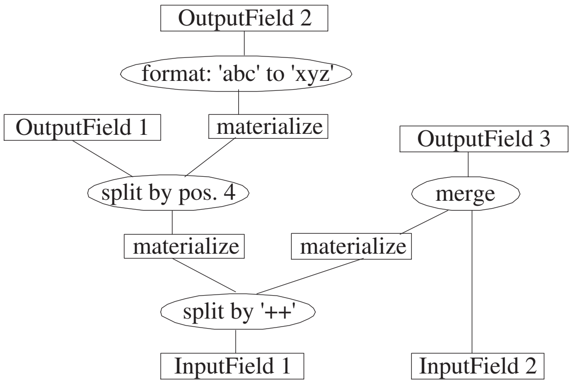


Figure 12: An example of a transformation that can be optimized

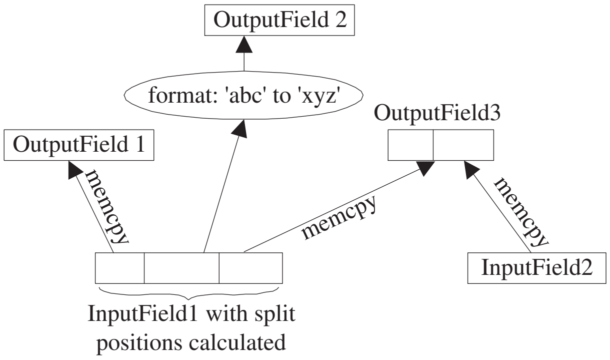


Figure 13: Optimized execution of the transformation

transform from the sequence. Collecting transforms and applying them on the dataset at the end also allows the system to optimize their application as described in Section 5.

Although we may have to transform the same tuple multiple times if the user scrolls back and forth to the same point, we only need to transform a screen-full of tuples at the scrolling and thinking speed of the user. The program generated at the end for the whole transformation will typically be long running, since it needs to manipulate the entire dataset. Our goal is to make interactive only the process of developing the right transformation for cleaning; the final program can be run unsupervised as a batch process. It could also be used as a wrapper [6] on the data source for subsequent accesses.

A problem with transforming data on the fly is ambiguity in data and anomaly lineage [52]; the user does not know whether a discrepancy was caused because of an error in the data or because of a transform. Not changing the actual data records also allows easy tracing of data lineage. As mentioned before discrepancy detection is always performed on the latest transformed version of the data. If the user wishes to know the lineage of a particular discrepancy, the system only needs to apply the transforms one after another, checking for discrepancies after each transform, to see whether the value was originally a discrepancy or whether the discrepancy was introduced by a transform.

5 Compiling a Sequence of Transforms into a Program

After the user is satisfied with the constructed sequence of transforms they can compile it into a transformation. This transformation can either be a C or Perl program, or macro. A macro is useful for applications that need a long and laborious sequence of the basic transforms. This sequence is saved as a macro that can subsequently be applied directly on any input.

The goal of Potter's Wheel is to let users specify transforms as they are needed — often only when discrepancies are found — in an order that is natural to the user. Hence the resultant transformation often has redundant or sub-optimal transforms. As we will discuss below, executing such a sequence of transforms exactly in the order specified by the user can be quite inefficient. Hence we want to convert the sequence of transforms specified by the user into one that is more efficient for execution, and compile them into an optimized program to perform the transformation on the database. We describe work on optimizing the final sequence of transform for memory accesses.

We consider only optimizations of transformations having one-to-one transforms only. Since such a transformation needs to be applied once for each row in the dataset we want to execute it efficiently. Thus, our granularity of optimization is a row-to-row mapping.

Transform	Input	Output
Format (to expr.,from expr.)	*singleton LLBS* A1	*singleton LLBS* having format(A1)
Merge	*LLBSs* A1..An, one from each input	*LLBS* flatten(A1,A2,..An)
Split (position 1, position 2, ...)	*LLBS* A1	*LLBS* output1, *LLBS* output2, ...
Split (regular expr. 1, regular expr. 2 ...)	*singleton LLBS* A1	*singleton LLBSs* output1, output2 ...
k-way Copy	*LLBS* A1	*LLBS* A1, *LLBS* A1, ... k times
Output (output buffer)	*LLBS* A1	concatenate A1 in output buffer
Input	*input buffer* A1	A1 as a *singleton LLBS*
Add (constant/serial/random)	*string* A1	A1 as a *singleton LLBS*
Divide (predicate)	*singleton LLBS* A1	A1 if predicate satisfied, else null
Materialize	*LLBS* A1	concatenate A1 in new o/p buffer and return this as a *singleton LLBS*

Figure 14: Operations performed by transforms. A singleton LLBS contains exactly one LBS.

Our main concern in this optimization is CPU cost. Since I/O and tuple transformation are pipelined, and we do only large sequential I/Os, the I/O cost is masked by the CPU cost. CPU cost is composed mainly of memory copies, buffer (de)allocations, and regular expression matches. `Format` and `Split` may involve regular expression matches. However almost any transform, if implemented naively without looking at the transforms that come before and after it, will have to (a) allocate a suitable sized buffer for the output, (b) transform the data from an input buffer to the output buffer, and (c) deallocate the output buffer after it has been used. This buffer is often of variable size (because transforms such as `Merge`, `Split`, and `Format` change data sizes and the data itself could be of variable size) and so must be allocated dynamically. With this approach, transforms like `Merge` and `Split` involve memory-to-memory copies as well. We use the term *materialization* to refer to the strategy of creating a buffer for the output of a transform.

Consider the example transformation shown in Figure 12 with materialization done between all successive transforms. The ovals represent transform operators and the rectangles represent materialization. The graph is executed bottom up, each operator executing when all of its inputs are ready. Three materializations are needed if the graph is executed naively, but these can be completely avoided as shown in Figure 13. The two `Split`'s can be done with one pass over the buffer `InputField1` to find out the split positions, and pointers to these substrings can been given as input to `Format` and copied to `OutputField1`. Similarly the third substring after the split can directly placed at the beginning of `OutputField3`.

The above optimization can be viewed as simply a programming "hack", but it is not clear how to compile an arbitrary transformation into an optimized program. In the rest of this section we generalize this idea. We show how to minimize materializations based on the constraints of various transforms. We then study how we can compile an optimized graph into a program that does no unnecessary materializations.

5.1 Determining a Minimal Set of Materialization Points

Some transforms impose constraints on the materialization of their inputs and/or outputs. Arbitrary `UDFs` and `Format` need their inputs to be materialized[8] and produce outputs that are materialized (because of the requirements of our regular-expression library and our arithmetic expression parser). `Split` by regular expression needs to have a

[8] We say that a value is materialized if it is present in a single contiguous memory buffer.

materialized input but need not materialize its output; it can stop at determining the split positions and directly pass on the buffer to its outputs.

Materialization can also be done for performance. Materializing before a `Copy` can be used to avoid reapplying the transforms below the copy to generate each of its outputs. This problem is akin to that of deciding whether to cache the results of part of a query in multi-query optimization [45, 38]. Ideally we want to materialize only if the cost of materialization is less than the (optimal) cost of performing the transforms before the copy to produce all the outputs. However determining the optimal cost of performing these transforms is very difficult since it depends on the (optimal) location of other materialization points both before and after the copy [45, 38]. Currently we adopt a heuristic of always inserting a materialization point before a `Copy`, except when there are no transforms before it.

We add to the transformation graph only those materializations that are needed to meet these constraints.

5.2 Restructuring the Transformation Graph

After inserting the minimum materializations needed, we simplify the transformation graph using some simple restructuring operations. We coalesce successive `Merges` into a single `Merge` and coalesce successive `Splits` into a single multi-output `Split`. The resultant `Split` is parameterized by the ordered combination of the splitting conditions of the constituent `Splits`.

However, if a regular-expression-based `Split` comes immediately after a position-based `Split`, we *do not* coalesce them together. Doing so would force the materialization of a bigger string before the coalesced `Split` as compared to not coalescing them and materializing after the position-based `Split` only.

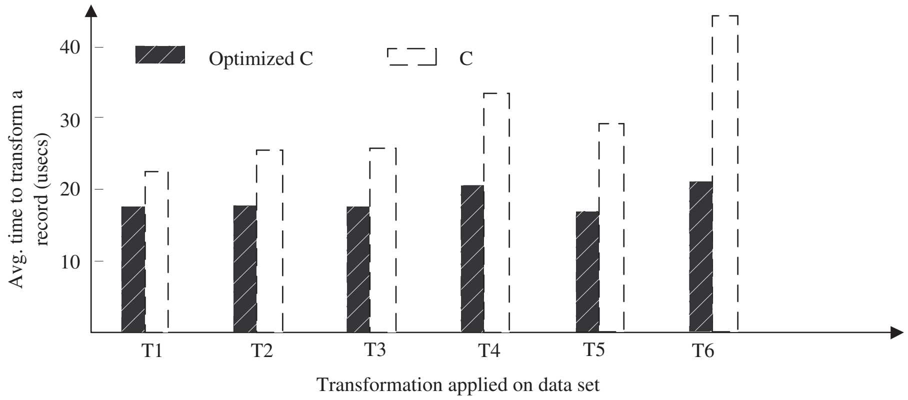
We also remove from the transformation graph all nodes whose only eventual ancestors are `Drops`.

5.3 Generating Optimized Code for Transforms

We aim to generate code for a transformation graph that never allocates or copies a buffer except at materialization points. We perform a bottom up traversal of the transformation graph. Each node is a task that needs to be performed; performing the task corresponds to looking at its input, applying the transform (or materializing if it is a materialize node) and then "passing on" results to the nodes above it. A node can *fire* only when all its inputs are available, so we maintain a queue of nodes that are ready to fire and repeatedly pick a node from it to fire. A node enters the queue when all its inputs are available.

We need a way for nodes to "pass on" their transformed results to the nodes above them without copies. Passing a pointer to a single buffer fails because operations like `Merge` combine multiple buffers. Hence the mechanism we use for "passing data" is an ordered *list of length based strings* (LLBS). Each *length based string* (LBS) consists of a pointer to a buffer along with the length of the buffer, and is typically a window in a larger null-terminated string.

Figure 14 gives the operations performed by (the code generated for) each transform along with its input and output. By operating only on LLBSs, our transforms never have to materialize except at the points described in Section 5.1. Due to lack of space we explain only the interesting ones. `Merge` accepts an LLBS from each input and outputs a "flattened" LLBS, *e.g.* if the inputs are `(lbs1, lbs2, lbs3)` and `(lbs4, lbs5)`, `Merge` outputs `(lbs1, lbs2, lbs3, lbs4, lbs5)`. A Split based on position takes in a LLBS and further refines it based on the new split positions *e.g.* `Split` by position 5 of the LLBS `(`` abcdef'', ``ghijklmn'')` produces two output LLBSs `(``abcde'')` and ("f", "ghijklmn"). `Add` statically creates a buffer with a suitable constant once (*not* once per record) and outputs this as an LBS, changing the buffer value each time if it is supposed to generate random numbers or serial numbers.



Transformation	Constituent Sequence of Transforms
T1	`Split` Date at position 5
T2	`Split` Source by 'to', `Merge` the right part with Destination
T3	`Merge` Source, Destination, `Split` result at position 4
T4	`Split` Source by 'to', `Merge` right part with Destination, `Split` Date by '/',
	`Format` resulting years with '19998' to '1998'
T5	`Split` Source by 'to', `Merge` right part with Destination, `Format` result with 'to ' to ''
	`Split` Date at position 5, `Copy` Delay
T6	`Split` Source by 'to', `Merge` right part with Destination, `Drop` Dept_Sch and Dept_Act
	`Split` Date at position 5, `Copy` Delay, `Add` constant 'Foo Bar'

Figure 15: Time taken for different transformations. The gain due to optimization increases with the number of transforms

5.4 Advantage of Compilation into Programs with Minimum Materialization

Figure 15 compares the average time taken for transforming a row using programs generated by Potter's Wheel for 8 transformations. We study the times taken for C programs generated with and without the optimizations of this section. These programs correspond to different transformations of a flight statistics data set[9]. All the generated C programs were compiled with the highest optimization settings in the Visual C++ 6.0 compiler.

We run 6 different transformations, ranging from single transforms to long sequences. We see that the optimizations described above give a speedup that varies from about 45% for simple transforms like T2 and T3, to about 110% for sequences of many transforms like T6. Note that both the optimized and non-optimized versions need to parse the input and copy the fields not involved in the transformation from the input to the output. Hence more elaborate transformations will get greater speedups.

We have also modified Potter's Wheel to generate Perl programs for the above transformations. However we found that the C programs are more than an order of magnitude faster than the Perl programs (we do not show them on the graph to avoid skewing the scale). This is a key advantage of automatically generating a program from a graphically specified transformation; if the user manually programs a transformation she is likely to choose a scripting language

[9] We used data downloaded from FEDSTATS [20] for all flights originating from Chicago O'Hare, San Fransisco, and New York JFK airports in 1997 and 1998. The columns in this dataset are shown in Figure 1, and it has 952771 records, with a total size of 73.186 MB. The schema for this dataset (shown in the user interface of Figure 1) is Delay:Integer, Carrier:Varchar(30), Number:Char(5), Source:Varchar(10), Destination: Varchar(5), Data: Char(13), Day:Char(3), Dept_Sch:Char(5), Dept_Act:Char(5), Arr_Sch:Char(5), Arr_Act:Char(5), Status:Char(10), Random:Integer. We ran experiments on a 400 MHz Intel Pentium 2 processor with 128 MB memory running Windows NT 4.0. We accessed this data from an ASCII file data source.

like Perl due to its ease of rapid programming.

6 Related Work

A nice description of the commercial data cleaning process is given in [7]. There are many commercial ETL (Extraction/Transformation/Loading) tools (also known as "migration" or "mapping" tools) that support transformations of different degrees, ranging from tools that do default conversions between a small set of common formats (*e.g.* White-Crane, Cognos [51, 9]) to fairly general tools that support a wide variety of transformations (such as Data Junction, Data Builder [14, 13]). Many of these tools provide a visual interface for specifying some common transformations, but typically require users to program other transformations using conversion libraries (*e.g.* Data Junction's CDI SDK and DJXL [14]). Moreover, these tools typically perform transformations as long running batch processes, so users do not early feedback on the effectiveness of a transformation.

Complementing the ETL tools are data quality analysis tools (a.k.a. "auditing" tools) that find discrepancies in data. This operation is again long-running and causes many delays in the data cleaning process. Typically these tools are provided by different vendors (from those who provide ETL tools) *e.g.* Migration Architect, Data Stage, ACR/Data [18, 15, 2]. Some vendors like Ardent and Evoke provide both ETL and analysis tools, but as different components of a software suite, leading to the problems described in Section 1.

Many scripting languages such as Perl and Python allow Turing complete transformations. However, writing scripts is difficult and error-prone due to a poor integration with exploration or discrepancy detection mechanisms. Often the data has many special cases and only after executing the script does one find that it does not handle all the cases.

There has been much work on languages and implementation techniques for performing higher-order operations on relational data [8, 33, 32, 37]. Our horizontal transforms are very similar to the restructuring operators of SchemaSQL [32]. Likewise much work on declarative transformation languages [1, 8]. Focus on linguistic power rather than ease of specification or interactive application.

In recent years there has been much effort on integration of data from heterogeneous data sources via middleware (*e.g.* Garlic, TSimmis, Hermes, Disco, Webmethods [6, 25, 3, 47, 50]). These efforts do not typically address the issue of errors or inconsistencies in data, or of data transformations. Recently, Haas *et. al.* have developed a tool to help users match the schema in these heterogeneous databases and construct a unified view [24]. We intend to extend Potter's Wheel in a similar manner with an interactive way of specifying rules for mapping schemas from multiple sources.

There has been some algorithmic work on detecting deviations in data [5], on finding approximate duplicates in data merged from multiple sources [28], and on finding hidden dependencies and their violations [29, 35]. Many of these algorithms are inherently "batch" algorithms, optimized to complete as early as possible and not giving any intermediate results. There are a few sampling based approaches that are incremental [30]. However these are not integrated with any mechanism to fix the discrepancies.

Extracting structure from semi-structured data is becoming increasingly important for "wrapping" or "screen-scraping" useful data from web pages. Many tools exist in both the research and commercial world. Examples are Nodose [4] and Araneus [22] among research projects, and Cohera Net Query [10] among commercial products. As discussed in Section 4.3, these tools typically require users to specify regular expressions or grammars; however these are often not sufficient to unaambiguously parse the data, so users have to write custom scripts. [21] addresses the problem of inferring regular expressions from a set of values, and use it to infer DTDs for XML documents. We tackle the case of general user-defined domains, and structural errors in data.

7 Conclusions and Future Work

Data cleaning and transformation are important tasks in many contexts such as data warehousing and data integration. The current approaches to data cleaning are time-consuming and frustrating due to long-running noninteractive operations, poor coupling between analysis and transformation, and complex transformation interfaces that often require user programming.

We have described Potter's Wheel, a simple yet powerful framework for data transformation and cleaning. Combining discrepancy detection and transformation in an interactive fashion, we allow users to gradually build a transformation to clean the data by adding transforms as discrepancies are detected. We allow users to specify transforms graphically, and show the effects of adding or undoing transforms instantaneously, thereby allowing easy experimentation with different transforms.

Potter's Wheel implements a set of transforms that are easy to specify graphically, can be implemented interactively, and are still quite powerful, handling all one-one and one-to-many mappings of rows as well as some higher-order transformations. Since the transformation is broken down into a sequence of simple transforms it can perform detailed optimizations when compiling them into a program. It provides a general and extensible mechanism for discrepancy detection in terms of user-defined domains. It uses the MDL principle to automatically infer the structure of column values in terms of these domains, so as to apply suitable detection algorithms. Structure extraction is also used to specify transforms by example.

An important direction for future work is integration with a system that handles interactive specification of the schema mapping between different sources, along the lines of Clio [24]. This will enable Potter's Wheel to automatically merge inputs from multiple sources before unifying theie formats.

Currently we assume that each attribute of a tuple is an atomic type. A interesting extension is to handle nested and semi-structured data formats that are now likely with XML becoming popular. We have seen that the transforms we provide can handle some transformations involving nested data, but the exact power of the transforms in this regard needs to be studied further. We also want to explore ways of detecting structural and semantic discrepancies in semi-structured data. An additional avenue for future work is more detailed optimizations of transforms, such as coalescing successive regular expression `Format`s together.

Acknowledgment: The scalable spreadsheet interface that we used was developed along with Andy Chou. Ron Avnur, Mike Carey, H.V. Jagadish, Laura Haas, Peter Haas, Marti Hearst, Renee Miller, Mary Tork Roth, and Peter Schwarz made many useful suggestions for the design of the transforms and the user interface. Renee Miller pointed us to work on handling schematic heterogeneities and suggested ways of handling them. Subbu Subramanian gave us pointers to related work in transformations. We used (source-code) modified versions of the PCRE-2.01, ispell-3.1.20, and calc-1.00 [26, 31, 34] libraries to support Perl-compatible regular expressions, perform spelling checks, and perform arithmetic operations respectively. Computing and network resources were provided through NSF RI grant CDA-9401156. This work was supported by a grant from Informix Corporation, a California MICRO grant, NSF grant IIS-9802051, a Microsoft Fellowship, and a Sloan Foundation Fellowship.

References

[1] S. Abiteboul, S. Cluet T. Milo, P. Mogilevsky, J. Simeon, and S. Zohar. Tools for data translation and integration. *IEEE Data Engg. Bulletin*, 22(1), 1999.

[2] ACR/Data. http://www.unitechsys.com/products/ACRData.html.

[3] S. Adali and R. Emery. A uniform framework for integrating knowledge in heterogeneous knowledge systems. In *Proc. Intl. Conference on Data Engineering*, 1995.

[4] B. Adelberg. NoDoSE — A tool for semi-automatically extracting structured and semistructured data from text documents. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, 1998.

[5] A. Arning, R. Agrawal, and P. Raghavan. A linear method for deviation detection in large databases. In *Proc. Intl. Conf. on Knowledge Discovery and Data Mining*, 1996.

[6] M. Carey, L. Haas, P. Schwarz, M. Arya, W. Cody, R. Fagin, M. Flickner, A. W. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. Williams, and E. Wimmers. Towards Heterogeneous Multimedia Information Systems: The Garlic Approach. In *RIDE-DOM*, 1995.

[7] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. In *SIGMOD Record*, 1997.

[8] W. Chen, M. Kifer, and D. S. Warren. HiLog: A foundation for higher-order logic programming. In *Journal of Logic Programming*, volume 15, pages 187–230, 1993.

[9] COGNOS Accelerator. http://www.cognos.com/accelerator/index.html.

[10] Cohera Corporation. CoheraNetQuery. http://www.cohera.com/press/presskit/CNQ_DataSheet032700.pdf.

[11] CoSORT. http://www.iri.com/external/dbtrends.htm.

[12] T. Cover and J. Thomas. *Elements of Information Theory*. John Wiley, 1991.

[13] DataBuilder. http://www.iti-oh.com/pdi/builder1.htm.

[14] Data Junction. http://www.datajunction.com/products/datajunction.html.

[15] Data Stage. http://www.ardentsoftware.com/datawarehouse/datastage/.

[16] Data Integration Information Center: series of articles. Intelligent Enterprise magazine. Also available at http://www.intelligententerprise.com/diframe.shtml.

[17] Data extraction, transformation, and loading tools (ETL). http://www.dwinfocenter.org/clean.html.

[18] Migration Architect. http://www.evokesoft.com/products/ProdDSMA.html.

[19] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11), 1996.

[20] FEDSTATS. http://www.fedstats.gov.

[21] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: A system for extracting document type descriptors from XML documents. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, 2000.

[22] S. Grumbach and G. Mecca. In search of the lost schema. In *Intl. Conf. on Database Theory*, 1999.

[23] M. Gyssens, L. Lakshmanan, and S. Subramanian. Tables as a paradigm for querying and restructuring. In *ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, 1996.

[24] L. Haas, R. J. Miller, B. Niswonger, M. T. Roth, P. M. Schwarz, and E. L. Wimmers. Transforming heterogeneous data with database middleware: Beyond integration. *IEEE Data Engg. Bulletin*, 22(1), 1999.

[25] J. Hammer, H. Garcia-Molina, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. Information Translation, Mediation, and Mosaic-Based Browsing in the TSIMMIS System. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, 1995.

[26] P. Hazel. PCRE 2.03. ftp://ftp.cus.cam.ac.uk/pub/software/programs/pcre/.

[27] J. M. Hellerstein, M. Stonebraker, and R. Caccia. Independent, open enterprise data integration. *IEEE Data Engg. Bulletin*, 22(1), 1999.

[28] M. Hernandez and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Mining and Knowledge Discovery*, 2(1), 1997.

[29] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *Proc. Intl. Conference on Data Engineering*, 1998.

[30] J. Kivinen and H. Manilla. Approximate dependency inference from relations. *Theoretical Computer Science*, 149(1):129–149, 1995.

[31] G. Kuennig. International Ispell version 3.1.20. ftp.cs.ucla.edu.

[32] L. Lakshmanan, F. Sadri, and S. Subramanian. On efficiently implementing SchemaSQL on a SQL database system. In *Proc. Intl. Conference on Very Large Data Bases*, 1999.

[33] Laks V.S. Lakshmanan, Fereidoon Sadri, and Iyer N. Subramanian. SchemaSQL: A language for intereoperability in relational multi-database systems. In *Proc. Intl. Conference on Very Large Data Bases*, 1996.

[34] R. K. Lloyd. calc-1.00. http://hpux.cae.wisc.edu.

[35] H. Mannila and K. Raiha. Algorithms for inferring functional dependencies from relations. *Data and Knowledge Engineering*, 12(1):83–99, 1994.

[36] K. Michael and G. Lausen. FLogic: A higher-order language for reasoning about objects, inheritance, and Scheme. *SIGMOD Record*, 18(6):134–146, 1990.

[37] R. J. Miller. Using schematically heterogeneous structures. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, 1998.

[38] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *Proc. Intl. Conference on Data Engineering*, 1988.

[39] J. R. Quinlan and R. L. Rivest. Inferring decision trees using the minimum description length principle. *Information and Computation*, pages 227–248, 1989.

[40] V. Raman, A. Chou, and J. M. Hellerstein. Scalable spreadsheets for interactive data analysis. In *ACM SIGMOD Wkshp on Research Issues in Data Mining and Knowledge Discovery*, 1999.

[41] V. Raman and J. M. Hellerstein. Potter's Wheel A-B-C: An interactive framework for data analysis, cleaning, and transformation. http://control.cs.berkeley.edu/abc, September 2000.

[42] V. Raman, B. Raman, and J. Hellerstein. Online dynamic reordering for interactive data processing. In *Proc. Intl. Conference on Very Large Data Bases*, 1999.

[43] S. Ramaswamy, R. Rastogi, and K. Shim. Efficient algorithms for mining outliers from large data sets. In *Proc. ACM SIGMOD Intl. Conference on Management of Data*, 2000.

[44] J. Rissanen. Modeling by shortest data description. *Automatica*, 14:465–471, 1978.

[45] T. Sellis. Multiple-query optimization. *TODS*, 1988.

[46] B. Shneiderman. The future of interactive systems and the emergence of direct manipulation. *Behavior and Information Technology*, 1(3):237–256, 1982.

[47] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to heterogeneous data sources with DISCO. *IEEE Trans. on Knowledge and Data Engineering*, 10(5), 1998.

[48] US Census Bureau. Frequently occurring first names and surnames from the 1990 census. http://www.census.gov/genealogy/www/freqnames.html.

[49] C. van Rijsbergen. *Information Retrieval*. Butterworths, 1975.

[50] WebMethods. Resolve complex B2B integration challenges once and for all. http://www.webmethods.com/content/1,1107,SolutionsIndex,FF.html.

[51] White Crane's Auto Import. http://www.white-crane.com/ai1.htm.

[52] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc. Intl. Conference on Data Engineering*, 1997.

A More details on Fold and Unfold Transforms

Many-to-Many transforms help to tackle higher-order *schematic heterogeneities* [37] where information is stored partly in data values, and partly in the schema. Figure 19 shows a case where a student's grades are listed as one row per course in one schema, and as multiple columns of the same row in another.

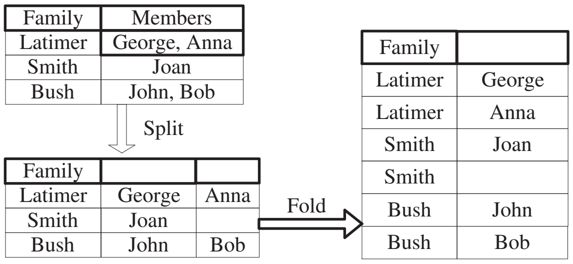


Figure 16: Folding a set of values into a single column

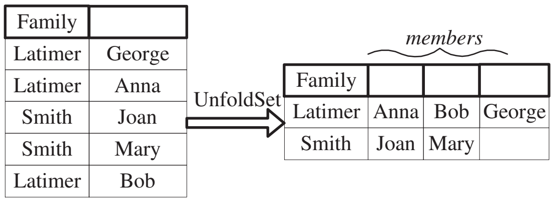


Figure 17: Unfolding a set of values, without an explicit column name to align

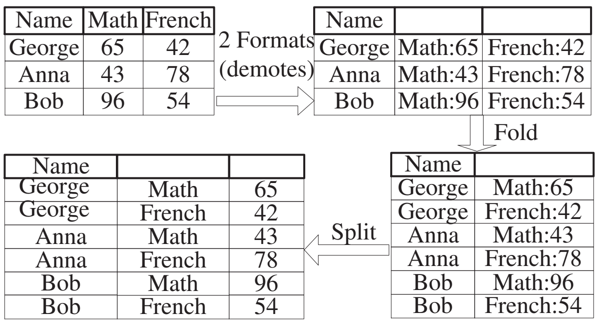


Figure 18: Fold to fix higher-order differences

Name	Math	French
George	65	42
Anna	43	78
Bob	96	54

Name		
George	Math	65
George	French	42
Anna	Math	43
Anna	French	78
Bob	Math	96
Bob	French	54

Figure 19: Higher order differences in data

Fold **Columns**

Fold converts one row into multiple rows, folding a set of columns together and replicating the rest, as defined in Table 2. Formally, a row $< a_1|a_2|\cdots|a_i|a_{i+1}|a_{i+2}|\cdots|a_{i+k-1}|\cdots|a_n| >$, on a k way fold of columns i through $i+k-1$ will form k rows $< a_1|a_2|\cdots|a_{i-1}|a_i|a_{i+1}|a_{i+k}|a_{i+k+1}|\cdots|a_n| >$, $< a_1|a_2|\cdots|a_{i-1}|a_i|a_{i+2}|a_{i+k}|a_{i+k+1}|\cdots|a_n| >$, and so on. Figure 18 shows an example with student grades where the subject names are demoted into the row via Format, grades are Folded together, and then Split to separate the subject from the grade.

Fold is adapted from the Fold restructuring operator of SchemaSQL [32], except that we *do not automatically demote column names* in Fold. Although demote and Fold are often done in succession to resolve schematic heterogeneities, we do not bundle these because there are many situations where there may not be a meaningful column name. *E.g.* columns formed by transformations, and columns containing expanded representation of sets, have no column name. Figure 16 shows an example where Fold without demote is needed. If the user wants, Fold and demote can be made into a macro as we describe in Section 5.

Note that the ability to fold arbitrarily many columns in one operation is crucial, and *cannot* be performed using a series of two-column folds because it leads to incorrect duplicate semantics (Figure 20).

Unfold **Columns**

Unfold is used to "unflatten" tables and move information from data values to column names, as shown in Figure 21. This is exactly the same as the Unfold restructuring operation of SchemaSQL [32]. However, since Fold is different, Unfold is *not* the exact inverse of Fold. Fold takes in a set of columns and folds them into *one* column, replicating the others. Unfold takes *two* columns, collects rows that have the same values for all the other columns, and unfolds the two chosen columns. This asymmetry is unavoidable; Unfold needs two columns because values in one column are used as column names to align the values in the other column.

Formally, Unfold(T, i, j) on the i'th and j'th columns of a table T with n columns named $c_1 \ldots c_n$ (a column with no name is assumed to have a NULL name) produces a new table with $n+m-2$ columns named $c_1 \ldots c_{i-1}, c_{i+1} \ldots c_{j-1}, c_{j+1} \ldots c_n, u_1$

George	Math:65	French:42	History:98
Anna	Math:43	French:78	History:32

fold

George	Math:65	History:98
George	French:42	History:98
Anna	Math:43	History:32
Anna	French:78	History:32

fold

George	Math:65
George	History:98
George	French:42
George	History:98
Anna	History:32
Anna	Math:43
Anna	History:32
Anna	French:78

Figure 20: A series of 2-column folds will not fold 3 columns together; note the duplicate History records

Name		
George	Math	65
George	French	42
Anna	Math	43
Anna	French	78
Bob	English	96
Bob	French	54
Joan	English	79

unfold(2,3)

Name	Math	French	English
George	65	42	
Anna	43	78	
Bob		54	96
Joan			79

Figure 21: Unfold-ing into three columns

where $u_1 \dots u_m$ are the distinct values of the i'th column in T. Every maximal set of rows in T that have identical values in all columns except the i'th and j'th, and distinct values in the i'th column, produces exactly one row. Specifically, a maximal set $\mathcal{S}$ of k rows $(a_1, \dots, a_{i-1}, u_l, a_{i+1}, \dots, a_{j-1}, v_l, a_{j+1}, a_{j+2}, \cdots, a_n)$ where l takes k distinct values in $[1..m]$, produces a row $(a_1, \cdots, a_{i-1}, a_{i+1}, \dots, a_{j-1}, a_{j+1}, a_{j+2}, \dots, a_n, v_1, v_2, \cdots, v_m)$. If a particular v_p in $v_1 \dots v_m$ does not appear in column j of the set $\mathcal{S}$, it is set to NULL. Values of column i, the *Unfolding* column, are used to create columns in the unfolded table, and values of column j, the *Unfolded* column, fill in the new columns.

Implementing Unfold in a generated program is simple; it is much like the group-by functionality of SQL. We sort by the columns other than the *Unfolding* and *Unfolded* columns and scan through this sorted set, collapsing sets of rows into one row.

Fold allows us to flatten our tables into a common schema where all the information is in the columns, thereby resolving schematic heterogeneities. Unfold allows us to reconvert the unified schema into a form where some information is in column names. Fold and Unfold are essentially the same as the restructuring operators of SchemaSQL, and the only restructuring operators of SchemaSQL we miss are Unite and Split that are used for manipulating multiple tables. For a more detailed analysis of the power of Fold and Split for (un)flattening tables, and also for application to OLAP, see [32, 23].

A.1 Unfolding Sets of Values

Since Unfold automatically promotes column names we cannot use it to restructure set valued attributes since there is no explicit column name. Figure 17 gives an example of the desired restructuring. In this case any alignment suffices. Our main `Unfold` operator needs a column from which it can promote column names for the unfolded values and use these names for alignment. Hence we use the following variant of `Unfold` for sets.

`UnfoldSet`(T, i) on the i'th column of a table T with n columns named $c_1 \dots c_n$ (a column with no name is assumed to have a NULL name) produces a new table with $n + m - 1$ columns named $c_1 \dots c_{i-1}, c_{i+1} \dots c_n, NULL, NULL, \dots (m$ NULLs), where m is the size of the largest set of rows in T with identical values in all columns except the i'th column.

For any maximal set of k tuples $\mathcal{S} = \{(a_1, a_2, \dots, a_{i-1}, v_l, a_{i+1}, \dots, a_n) \mid l = 1, 2, \dots k\}$, with identical values in all columns except the i'th, `UnfoldSet` generates one tuple $(a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n, v_1, v_2, \dots v_k, NULL, NULL, \dots (m-k)$ NULLs). Note that the ordering of $v_1, \dots v_k$ is not specified by the definition, because UnfoldSet does not enforce an alignment. In Figure 17, the family members' names could be permuted in any way in the resulting table. In an implementation a default ordering, such as lexicographic, must be used.

B Power of Transforms

In this section we analyses the power of vertical transforms and horizontal transforms. We use n-ways versions of the `Split` and `Merge` transforms for simplicity — these can be implemented by $n-1$ applications of regular `Splits` and `Merges` respectively.

B.1 Power of One-to-One Transforms

Theorem 1: *One-to-One Transforms can be used to perform all one-to-one mappings of tuples in a table.*
Proof: Suppose that we want to map a row $(a_1, a_2, \ldots, a_n)$ to $(b_1, b_2, \ldots, b_m)$. Let b_i be defined as $b_i = g_i(a_{i_1}, a_{i_2}, \ldots, a_{i_l})$. Assume that | is a character not present in the alphabet from which the values are chosen. An obvious way of performing this transformation is as follows:
split(`format`(merge($a_1, a_2, \ldots, a_n$, |), `udf`), |)
where split and `merge` are m-ary and n-ary versions of the `Split` and `Merge` transforms defined in Section 4 respectively, and `udf` is a UDF that converts $a_1|a_2|\cdots|a_n$ into $b_1|b_2|\cdots|b_m$. While this approach allows us to use only a few transforms, it forces us to write unnecessary UDFs.

However, the use of `Drop` and `Copy` transforms allows one to do the transformation using only UDFs $g_1, \ldots, g_m$ – these UDFs are essential because they are explicitly used in the definition of $b_1, \ldots b_m$. This is important because in many cases it will be possible to express these functions $g_1, \ldots g_m$ in terms of regular-expression and arithmetic-expression based `Formats`, thus avoiding any user programming. Since a given a_j may be used in multiple conversion functions g_i and a `Format` automatically drops the old value (Table 2), we need to make an explicit copy of it. Hence, to form b_i, we first make a `Copy` of $a_{i_1}, \ldots a_{i_l}$, `Merge` these to form $a_{i_1}|a_{i_2}|\cdots|a_{i_l}$, and apply g_i on this merged value to form b_i. After applying this process to form $b_1, \ldots, b_m$, we must `Drop` $a_1, \ldots, a_n$.

B.2 Power of Many-to-Many Transforms

We prove that by combining Many-to-Many and One-to-One Transforms, we can perform one-to-many transformations of rows in a table. In addition, by using `Format` for demoting, and `Fold` and `Unfold`, we can move information between schema and data and thereby flatten and unflatten tables. The `Fold` and `Unfold` transforms are essentially the same as the restructuring operators of SchemaSQL, and the only restructuring operators of SchemaSQL that we miss are Unite and Split that are used for manipulating multiple tables. For a more detailed analysis of the power of these restructuring operators for flattening tables, see [32, 23].

Theorem 2: *Many-to-Many Transforms when combined with One-to-One transforms can perform all one-to-many mappings of tuples in a table.*
Proof:

Suppose that we want to map a row $(a_1, a_2, \ldots, a_n)$ to a set of rows
$\{(b_{1,1}, \ldots, b_{1,m}), (b_{2,1}, \ldots, b_{2,m}), \ldots, (b_{k,1}, \ldots, b_{k,m})\}$. Note that the number of output rows k *itself can vary* as a function of $(a_1, \ldots, a_n)$. Let K be the maximum value of k for all rows in the domain of the desired mapping. Assume that | is a character not present in the alphabet from which the values are chosen.

We first perform a one-to-one mapping on $(a_1, \ldots, a_n)$ to form
$(b_{1,1}|b_{1,2}|\cdots|b_{1,m},\quad b_{2,1}|b_{2,2}|\cdots|b_{2,m},\quad \ldots,\quad b_{k,1}|b_{k,2}|\cdots|b_{k,m},\quad NULL,\quad NULL,\quad \ldots\quad)$, with $K-k$ NULLs at the end. We then perform a K way `Fold`, and a `Select` to remove all the resulting NULLs. Finally, we perform a m-way `Split` by | to get the desired mapping.

Note: The above proof assumes that K, the upper bound on the number of rows a single row can map into, can be bounded. For example, if an individual field of a row were to be a document, such a row cannot be mapped into a form where there is one word per row, since the number of words per document is unlikely to known *a priori* to the user.