Potter's Wheel: An Interactive Data Cleaning System

Vijayshankar Raman and Joseph M. Hellerstein

University of California at Berkeley
{rshankar, jmh}@cs.berkeley.edu

Abstract

Cleaning data of errors in structure and content is important for data warehousing and integration. Current solutions for data cleaning involve many iterations of data "auditing" to find errors, and long-running transformations to fix them. Users need to endure long waits, and often write complex transformation scripts.

We present Potter's Wheel, an interactive data cleaning system that tightly integrates transformation and discrepancy detection. Users gradually build transformations to clean the data by adding or undoing transforms on a spreadsheet-like interface; the effect of a transform is shown at once on records visible on screen. These transforms are specified either through simple graphical operations, or by showing the desired effects on example data values. In the background, Potter's Wheel automatically infers structures for data values in terms of user-defined domains, and accordingly checks for constraint violations. Thus users can gradually build a transformation as discrepancies are found, and clean the data without writing complex programs or enduring long delays.

1 Introduction

Organizations accumulate much data that they want to access and analyze as a consolidated whole. However the data often has inconsistencies in schema, formats, and adherence to constraints, due to many factors including data entry errors and merging from multiple sources [6, 13]. The data must be purged of such discrepancies and transformed into a uniform format before it can be used. Such *data cleaning* is a key challenge in data warehousing [6]. Data transformation is also needed for extracting data from legacy data formats, and for Business-to-Business Enterprise Data Integration [26].

1.1 Current Approaches to Data Cleaning

Data cleaning has three components: auditing data to find discrepancies, choosing transformations to fix these, and applying the transformations on the dataset. There are currently many commercial solutions for data cleaning (*e.g.*, see [9] for an overview). They come in two forms: auditing tools and transformation tools. The user first audits the data to detect discrepancies using an auditing tool like Unitech Systems' ACR/Data or Evoke Software's Migration Architect. Then she either writes a custom script or uses an *ETL* (Extraction/Transformation/Loading) tool like Data Junction or Ascential Software's DataStage 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 of auditing and transformation must be repeated until the "data quality" is good enough. This approach has two problems.

- *Lack of interactivity:* Transformation is typically done as a batch process, operating on the whole dataset without any feedback. This leads to long, frustrating delays during which users have no idea if a transformation is effective.

  Such delays are compounded by a decoupling of transformation and discrepancy detection – these are often done as separate steps, with separate software. This forces users to wait for a transformation to finish before they can check if it has fixed all anomalies. More importantly, some *nested discrepancies* arise only after others have been fixed. *E.g.*, a typo in a year field such as "19997" can be found (by running a suitable algorithm on the year values) only after all dates have been converted to a uniform date type – until then, the year values cannot be isolated from the date strings. Thus the decoupling makes it hard to find multiple discrepancies in one pass, leading to many unnecessary iterations.

- *Need for much user effort:* Both transformation and discrepancy detection need significant user effort, making each step of the cleaning process painful and error-prone.

  Commercial ETL tools typically support only some restricted transforms[1] between a small set of formats via a GUI, and provide ad hoc programming interfaces for general transforms (these are essentially libraries of conversions between standard formats: *e.g.* Data Junction's DJXL). Even system-supported transforms often need to be specified in sophisticated ways that involve regular expressions or grammars (Section 4.3).

  The discrepancy detection technique must match the data domain – it may be a standard method like spell-checking, or a specialized one like spotting non-standard names for automobile parts. Unfortunately data values are often composite structures of values from different domains, like *"Rebecca by Daphne du Maurier, et. al Hardcover (April 8, 1948) $22.00"* (from Amazon.com search results for

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

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

Daphne Du Maurier). Hence users must either write custom programs for each such structure, or design transforms to parse data values into atomic components for anomaly detection, and then back into unified wholes for output.

1.2 Potter's Wheel Approach

Data cleaning is intrinsically a complex, knotty task, with many interrelated problems. Any solution must support transformation and discrepancy detection in an integrated fashion. On one hand, the transforms provided must be general and powerful enough to do most tasks without explicit programming, and the system must extensibly support the variety of discrepancy detection algorithms applicable in different domains. On the other hand, since the cleaning process involves user interaction, the system must support transformation and discrepancy detection through simple specification interfaces and with minimal delays.

Potter's Wheel is an interactive data cleaning system that integrates transformation and discrepancy detection in a single interface. The software is publicly available from Berkeley [22], and some of the main ideas are also used in Cohera's Content WorkBench [8].

Users gradually build transformations in Potter's Wheel by composing and debugging transforms, 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. This 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.

1.2.1 Interactive Transformation

From the literature on transformation languages (*e.g.*, [1, 7, 16]) we have adapted a small set of transforms that support common transformations without explicit programming. Most of these are simple and easy to specify graphically. However some transforms used to parse and split values into atomic components are quite complex. Their specification requires users to enter regular expressions or grammars, and in some cases write custom programs (Section 4.3). Instead Potter's Wheel lets users specify the desired results on example values, and automatically infers a suitable transform, using the structure extraction techniques described below. We describe such graphical specification, and the incremental application of these transforms, in Section 4.

Potter's Wheel compiles a sequence of transforms into a program after the user is satisfied, instead of applying them piecemeal over many iterations. Users specify or undo these transforms in orders they find natural, often only when discrepancies are found, and this exploratory behavior could result in redundant or sub-optimal transforms. In addition, the main cost in the transformation is that of memory allocation and copying. In the full version of the paper [22], we discuss how the final sequence of transforms can be converted to a more optimal form, including ways of pipelining transforms to minimize memory allocations and copies.

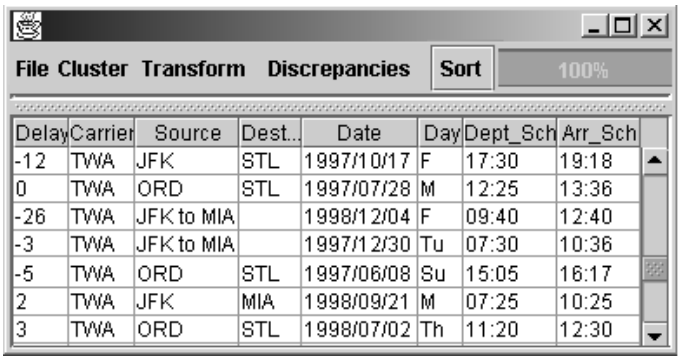

Figure 1: A snapshot of the Potter's Wheel user interface on flight delay data from FEDSTATS (www.fedstats.gov).

1.2.2 Extensible Discrepancy Detection

Potter's Wheel allows users to define custom *domains*, and corresponding algorithms to enforce domain constraints. However since the data values are often composite structures, the system needs to automatically parse a string value into a structure composed of user-defined domains, and then apply suitable discrepancy detection algorithms.

This is similar to the problem of inferring regular expression structures from examples, that has been addressed in the machine learning literature (*e.g.*, [20, 5]). We are however not interested in abstract structures like regular expressions, but rather in structures in terms of user-defined domains. For example, parsing flight records like "*Taylor, Jane, JFK to ORD on April 23, 2000 Coach*" as "[A-Za-z,]* [A-Z]^3 to [A-Z]^3 on [A-Za-z]* [0-9]*, [0-9]* [A-Za-z]*" does not help much with detecting anomalies that satisfy the basic pattern. Whereas parsing it as "[A-Za-z,]* $<Airport>$ to $<Airport>$ on $<Date>$ $<Class>$" would allow us to detect logical errors like false airport codes or dates.

We believe that application developers will specify useful application-specific domains and corresponding domain constraints (like date, airport code, construction part name), provided Potter's Wheel can automatically infer patterns in terms of these domains and apply suitable algorithms. The difficulty is that these domains are typically not specified as explicit patterns but rather as encapsulated set-membership functions that Potter's Wheel cannot understand. A second unique feature of pattern learning in the data cleaning context is that the values will have discrepancies in their structure itself; hence Potter's Wheel can only detect approximate structures. There is a tradeoff here between choosing structures that match most of the values in a column and choosing structures that do not overfit the data values. Section 3 describes how the Minimum Description Length principle [24] can be used to extract approximate structures for values in a way that balances this tradeoff.

2 Potter's Wheel Architecture

The main components of the 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 [23, 21], and an *Automatic Discrepancy Detector*.

2.1 Data Source

Potter's Wheel accepts input data as a single, pre-merged stream, that can come from an ODBC source or any ASCII file descriptor (or pipe). The ODBC source can be used to query data from DBMSs, or even from distributed sources via middleware. In practice, schematic differences between sources will restrict the tightness of the integration via a query (even Figure 1 shows poor mapping in the *Source* and *Destination* columns). Potter's Wheel will flag areas of poor integration as errors, and the user can transform the data, moving values across columns to unify the data format.

When reading from ASCII files, each record is viewed as a single wide column. The user can identify column delimiters graphically and split the record into constituent columns. Such parsing is more complex and time-consuming on poorly structured data (such as from web pages). Potter's Wheel helps this process through a Split transform that can be specified by example (Section 4). Column types and delimiters can also be specified in a metadata file. Once a dataset has been parsed, the transformation can be stored as a macro for easy application on similar datasets.

2.2 Interface used for Displaying Data

Data read from the input is displayed on a Scalable Spreadsheet interface [21] that allows users to interactively re-sort on any column, and scroll in a representative sample of the data, even over large datasets. When the user starts Potter's Wheel on a dataset, the spreadsheet interface appears immediately, without waiting until the input has been completely read. This is important when transforming large datasets or never-ending data streams.

The interface supports this behavior using an Online Reorderer [23] 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 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 screen. Thus users can explore large amounts of data along any dimension. Exploration helps users spot simple discrepancies by observing the structure of data as values in the sort-column change.

2.3 Transformation Engine

Transforms specified by the user need to be applied in two scenarios. First, they need to be applied when records are rendered on the 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.

2.4 Automatic Discrepancy Detector

While the user is specifying transforms and exploring the data, the discrepancy detector runs in the background, applying appropriate algorithms to find errors in the data. Hence

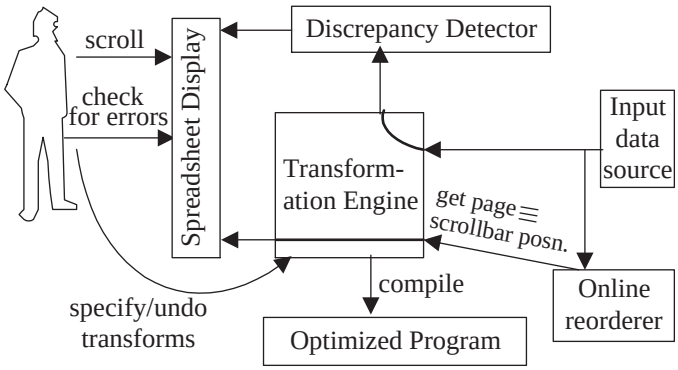

Figure 2: Potter's Wheel Architecture

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 first parses values in each field into *sub-components* according to the structure inferred for the column. The structure of a column is a sequence of user-defined domains, and 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. Then suitable algorthims are applied for each sub-component, depending on its domain. For example, if the structure of a column is $<number><word><time>$ and a value is 19 January 06:45, the discrepancy detector finds 19, January, and 06:45 as sub-components belonging to the $<number>$, $<word>$, and $<time>$ domains, and applies the detection algorithms specified for those domains.

2.5 Compiling a Sequence of Transforms

After the user is satisfied with the sequence of transforms, Potter's Wheel can compile it into a transformation, and export it as either a C or Perl program, or a Potter's Wheel macro – the latter can be invoked on other datasets to reapply the transformation without respecifying each transform. In future we want to support compilation into declarative languages like SQL or XSLT, so that a database system could perform further optimizations.

3 Extensible Discrepancy Detection

Potter's Wheel allows users to define arbitrary *domains*, along with corresponding discrepancy detection algorithms. We describe the API for specifying domains in Section 3.1. Based on these domains, the system automatically infers appropriate structures for values in each column (Section 3.2). Some of the domains in this structure are then *parameterized* for the specific column values, as discussed in Section 3.3. Once this detailed structure is inferred, the system parses values and sends individual components to suitable discrepancy detection algorithms (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 function required to be implemented is an inclusion function match to identify values in the domain. The optional cardinality function is helpful in structure extraction as we describe in Section 3.2. up-

```
public abstract class Domain {
    /** Required Inclusion Function — Checks if value satisfies domain constraints. (Sections 3.1) */
    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. (Sections 3.2 and 3.3) */
    public int cardinality(int length);

    /** Optional function – updates any state for this domain using the given value. ( Sections 3.1 and 3.3) */
    public void updateStats(char* value);

    /** Optional function – checks if a given value is a discrepancy, with a certain probability. Typically needs
        to know the total number of tuples in the data set (e.g. see [14]). (Section 3.1) */
    public float matchWithConfidence(char *value, int dataSize);

    /** Optional function – checks if one pattern is redundant after another. (Section 3.2) */
    public boolean isRedundantAfter(Domain d);
}
```

Figure 3: API for user-defined domains. These functions are explained in the sections indicated.

dateStats 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 also 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 matchWithConfidence method is helpful for probabilistic and incremental discrepancy detection algorithms, such as sampling based algorithms (e.g. [14]). The isRedundantAfter method is used while enumerating structures, as described in Section 3.2.

Potter's Wheel provides the following default domains: arbitrary ASCII strings (henceforth called ξ^*), character strings (called *Words*; likewise *AllCapsWords* and *CapWords* refer to words with all capitals and capitalized words respectively), *Integers*, sequences of *Punctuation*, C-style *Identifiers*, floating point values (henceforth called *Doubles*), English words checked according to ispell (*IspellWords*), common *Names* (checked by referring to the online 1990 census results), *Money*, and a generic regular-expression domain that checks values using the PCRE library.

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 [*achrM*]* [*17*]*, [*20*]*, to name a few possible structures. 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 suitable structure $S = d_{s_1} d_{s_2} \ldots d_{s_p}$, where $1 \leq s_1 \ldots s_p \leq m$.

All that we know about these domains is from the functions defined in Figure 3 – even among these, only the set-membership function (match) may be available. In general the inferred structure must be approximate, since the data could have errors in the structure itself. We first describe how to evaluate the appropriateness of a structure for a set of values and then look at ways of enumerating all structures so as to choose the best one.

3.2.1 Evaluating the Suitability of a Structure

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

- *Recall:* The structure should match as many of the column values as possible.
- *Precision:* The structure should match as few other values as possible.
- *Conciseness:* The structure should have minimum length.

The first two criteria are standard IR metrics for evaluating the effectiveness of a pattern [27]. 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*.

This last example highlights the importance of allowing user-defined domains in the alphabet from which we create the structure. For instance if we did not have *Word* and *Integer* as domains in the alphabet, *M a r c h 1 7 , 2 0 0 0* would be the better structure than *[A-Za-z]* [0-9]*, [0-9]** 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* [A-Za-z]* *as the domain Word, rather than as the Kleene closure of a set of 56 characters.*

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 make the tradeoff between over-fitting and under-fitting is through the Minimum Description Length (MDL) principle [24], that minimizes the total length required to encode the data using a structure.

Description Length: A metric for structure quality

We now derive the description length (DL) for encoding a set of values with a structure, as a measure of the appropriateness of the structure; better structures result in smaller DLs. According to the MDL principle, the DL for using a structure

to describe a set of column values is defined as: the length of the theory (the structure definition) plus the length required to encode the values given the structure.

We need a DL that can encapsulate the goals of Recall, Precision, and Conciseness (as penalties). 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, forcing values to be written out explicitly. Thereby we capture Recall.

Example: Consider a structure of *Word Integer Integer* and a value of May 17 2025. The number of bits needed to encode the structure is $3\log(\text{number of domains})$. Then we encode the value by first specifying the length of each sub-component and then, for each sub-component, specifying the actual value from all values of the same length. In this case, the sub-component lengths are 3, 2, and 4. The domains are strings over alphabets of size 52, 10, and 10 ([a-zA-Z] and [0-9]). Thus the description length is: $3\log(\text{number of domains}) + 3\log(\text{maximum length of values in each sub-component}) + \log 52^3 + \log 10^2 + \log 10^4$.

In the above example, we are able to calculate the lengths of the value encodings for integers and words because we know the properties of the domains. We now look at encodings for structures of arbitrary domains. Consider a structure S of p domains, $d_{s_1} d_{s_2} \ldots d_{s_p}$. Let $|T|$ denote the cardinality of any set T. The description length of a string v_i of length $len(v_i)$ using S is $DL(v_i, S) =$

$$\text{length of theory for } S \ + \ \text{length to encode } \ v_i \text{ given } S$$

Given m domains, we can represent each domain with $\log m$ bits. Let f be the probability that v_i matches the structure S. If v_i does not match, we encode it explicitly. Thus,

$$DL(v_i, S) = p\log m + (1-f)(\log|\xi^{len(v_i)}|) + f(\text{space to express } v_i \text{ with } S)$$

with the three parts of the right hand side representing penalties for Conciseness, Recall, and Precision respectively. Let $v_1, v_2, \ldots v_n$ be the values in the column, and $AvgValLen = \sum_{1\le j\le n} len(v_j)$ be the average length of the values in the column. It is easy to see that the average space needed to encode the values is

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

Just as in the example, we express the values v_i using S by first encoding the lengths of their 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 the structure $S = d_{s_1} d_{s_2} \ldots d_{s_p}$ through the concatenation of sub-components $v_i = w_{i,1} w_{i,2} \ldots w_{i,p}$, with $w_{i,j} \in d_{s_j}\ \forall 1 \le j \le p$ (this parsing of v_i is itself not easy; we discuss efficient parsing in Section 4.3.2). 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,

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

After some transformation this becomes

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

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 approximate the fraction directly by repeatedly choosing random strings of appropriate length and checking if they satisfy d_{s_j}. Since these fractions are independent of the actual values, they can be pre-computed and cached.

If the length is too high we may need to check many values before we can estimate this fraction. Hence in the absence of a user-defined cardinality function, we compute the fraction of matches for a few small lengths, and extrapolate it to larger lengths assuming that the number of matches is a strict exponential function of the string length.

3.2.2 Choosing the best structure

We have seen how to compute a description length that measures the suitability of a structure for a given set of values. We now want to 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, and it will be too expensive to enumerate all such strings.

We apply the algorithm of Figure 4 on a set of sample values from the column, and take the union of all structures enumerated thus. We use 100 values as a default; this has proven adequate in all the cases we have encountered. During this enumeration, we prune the extent of recursion by not handling structures with 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 isRedundantAfter(Domain d) method of Domain that determines whether this domain is redundant immediately after the given domain.

Even though this is an exponential algorithm, pruning reduces the number of structures we enumerate for a column considerably. As shown in Figure 5, the number of enumerated structures is typically less than 10.

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 examples of Figure 5.

Potter's Wheel currently detects two parameterizations automatically: domains with constant values and domains

```
/** Enumerate all structures of 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 matching prefix w_1 ... w_k of v_i
    do enumerate(w_{k+1} ... w_m , d_{s_1}, ... d_{s_p})
      – avoid structures beginning with domains
        d' that satisfy d'.isRedundantAfter(d)
    prepend d to all structures enumerated above
}
```

Figure 4: Enumerating various structures for a set of values

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

Figure 5: Structures extracted for different kinds of columns, using the default domains listed in Section 3.1. Structure parameterizations are given in parenthesis.

with values of constant length. Such parameterized structures are especially useful for automatically parsing the values in a column, when inferring Split transforms by example (Section 4.3).

In addition, users can define domains that infer custom parameterizations, using the updateStats method. These domains could use specialized algorithms to further refine the structure of the sub-components that fall within their domain. For example, the default *Integer* domain in Potter's Wheel computes the mean and standard deviation of its values and uses these as parameters, to flag values that are more than 2 standard deviations away as potential anomalies. Likewise a domain can accept all strings by default, but parameterize itself by inferring a regular expression that matches the sub-component values.

The description length for values using a structure often reduces when the structure is parameterized. For the default parameterizations of constant values and constant lengths it is easy to adjust the formulas given in the previous section. For custom parameterizations like the regular expression inference discussed above, the user must define the cardinality function based on the parameterization.

3.4 Example Structures Extracted

Consider the snapshot shown in Figure 1 containing flight delay statistics. Figure 5 shows the structures extracted for some of its column values, and also for some columns from a web access log. We see that the dominant structure is chosen even in the face of inconsistencies; thereby the system can flag these structural inconsistencies as errors to the user, and parse and apply suitable detection algorithms for other values that match the structure.

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 of Figure 1 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 of Figure 1 as structural anomalies.

Figure 5 also shows that a column of IP addresses with values like 12.8.15.147 has its structure inferred as *Double.Double*, rather than *Integer.Integer.Integer.Integer*. This arises because *Double* is a more concise structure than *Integer.Integer*. This could be avoided either by defining a *Short* domain for values less than 255 (to form *Short.Short.Short.Short*), 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 the next best structure, *AllCapsWord*. The system flags TWA as an anomaly – the user could choose to ignore this, 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.

Figure 10 gives more examples of inferred structures.

4 Interactive Transformation

Having seen how Potter's Wheel infers structures and identifies discrepancies, we turn our attention to its support for interactive transformation. We want users to construct transformations gradually, adjusting them based on continual feedback. This breaks down into the following sub-goals:

Ease of specification: Transforms must be specifiable through graphical operations rather than custom programming. Moreover, in these operations, we want to avoid use of regular-expressions or grammars and instead allow users to specify transforms by example as far as possible.

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 it.

Undos and Data Lineage: Users must be able to easily undo transforms after seeing their effect. In addition, the lineage of errors must be clear – *i.e.*, errors intrinsic to the data must be differentiable from those resulting from other transforms.

4.1 Transforms supported in Potter's Wheel

The transforms used in Potter's Wheel are adapted from existing literature on transformation languages (*e.g.* [16, 7]). We describe them briefly here before proceeding to discuss their interactive application and graphical specification. Table 1 gives formal definitions for these transforms. Additional illustrative examples and proofs of expressive power are given in the full version of the paper [22].

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	$\delta((a_1, \ldots, a_n), i, \text{pred})$	$=$	$\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, a_i, \text{null}) \mid (a_1, \ldots, a_n) \in R \wedge \text{pred}(a_i)\} \cup$ $\{(a_1, \ldots, a_{i-1}, a_{i+1}, \ldots, a_n, \text{null}, a_i) \mid (a_1, \ldots, a_n) \in R \wedge \neg\text{pred}(a_i)\}$
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, $\text{left}(x, \text{splitter})$ is the left part of x after splitting by splitter. pred is a function returning a boolean.

Table 1: Definitions of the various transforms. Unfold is defined in the full paper [22].

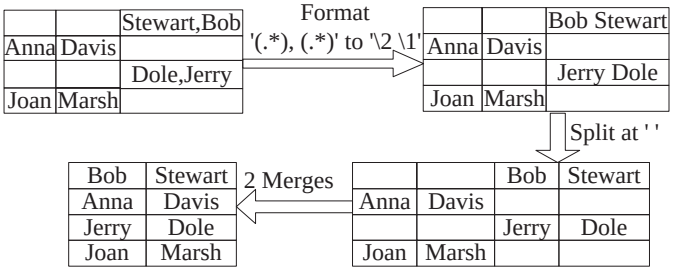

Figure 6: Using Format, Merge and Split to clean name format differences

Value Translation

The Format transform applies a function to every value in a column. We provide built-in 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.

One-to-one Mappings of Rows

One-to-one transforms are column operations that transform individual rows. As illustrated in Figures 6 and 7, they can be used to unify data collected from different sources.

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 parts. The split positions are often difficult to specify if the data is not well structured. We allow splitting by specifying character positions, regular expressions, or by interactively performing splits on example values (Section 4.3).

Drop, Copy, and Add allow users to drop or copy a column, or add a new column. Occasionally, logically different values (maybe from multiple sources) are bunched into the same column, and we want to transform only some of them. Divide conditionally divides a column, sending values into one of two new columns based on a predicate.

Many-to-Many Mappings of Rows

Many-to-Many transforms help to tackle higher-order *schematic heterogeneities* [18] where information is stored partly in data values, and partly in the schema, as shown in Figure 8. Fold "flattens" tables by converting one row into multiple rows, folding a set of columns together into one column and replicating the rest. Conversely Unfold "unflattens" tables; it takes *two* columns, collects rows that have the same values for all the other columns, and unfolds the two chosen columns. Values in one column are used as column names to align the values in the other column. Figures 8 and 9 show 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 and UnFold are adapted from the restructuring operators of SchemaSQL [16], and are discussed in more detail in the full paper [22].

Power of Transforms: As we prove in the full paper [22], these transforms can be used to perform all one-to-many row mappings of rows. Fold and Unfold can also be used to *flatten* tables, converting them to a form where column and table names are all literals and do not have data values. For a formal definition of (un)flattening and an analysis of the power of Fold and Unfold, see [16].

4.2 Interactive Application of Transforms

We want to apply the transforms on tuples incrementally, as they stream in, so that the effects of transforms can be immediately shown on tuples visible on the screen of the UI. It also lets the system pipeline discrepancy detection on the results of the transforms, thereby giving the interactivity advantages described in the introduction.

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 incrementally.

However Unfold operates on a set of rows with matching values. Since this could potentially involve scanning the entire data, we do not allow Unfold to be specified graphically. For displaying records on the screen we can avoid this problem by not showing a complete row but instead showing more and more columns as distinct values are found, and filling data values in these columns as the corresponding input rows are read. Such progressive column addition in the spreadsheet interface could confuse the user; hence we plan to implement an abstraction interface where all newly created columns are shown as one rolled up column. When

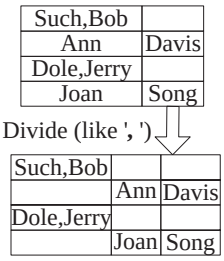

Figure 7: Divide-ing to separate various name formats

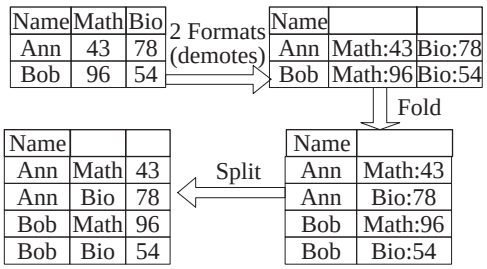

Figure 8: Fold-ing to fix higher-order variations

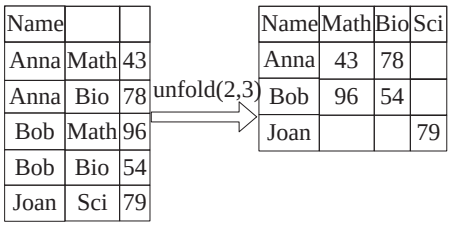

Figure 9: Unfold-ing into three columns

the user clicks to unroll the column it expands into a set of columns corresponding to the distinct values found so far.

4.3 Graphical Specification of Transforms

Transforms like Add, Drop, Copy, Fold, and Merge are simple to specify graphically. Users can highlight the desired columns and pick the appropriate transform.

However Split is often hard to specify precisely. Splits are needed to parse values in a column into consituent parts, as illustrated in Figure 10. This is an important problem for commercial integration products. Tools like Microsoft SQL Server and Access have wizards for parsing ASCII data files with constant delimiters. There are many research and commercial "wrapper-generation" tools (e.g. Araneus [12], Cohera Net Query (CNQ) [8], Nodose [2]) that tackle this 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 10, 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 want users to be able to parse and split values without specifying complex regular expressions or writing programs. Instead 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 [25] way shows the positions at which these values are to be split, into sub-components $(x_{1,1}x_{1,2}\ldots x_{1,m})$, $(x_{2,1}\ldots x_{2,m}), \ldots,$ $(x_{n,1}\ldots x_{n,m})$ respectively. As is 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 "*Word(len 3) Word(len 3) Integer(len 2) time Integer(len 4)*" (for the output of the UNIX date command). Therefore they are better than simple regular expressions at identifying split positions.

Figure 10 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, it can extract good structures that can be used to split unambiguously.

/** Split a string q (of characters $w_1 \ldots w_m$) using structures * $S_1, S_2, \ldots S_k$.*/

void LeftRight(q,$S_1, \ldots S_k$) {
  If $k == 0$, check if q is empty
  for all prefixes $w_1 \ldots w_j$ of q satisfying S_1 do
    LeftRight($w_{j+1} \ldots w_m$, $S_2 S_3 \ldots S_k$)
}

void DecSpecificity(q,$S_1, \ldots S_k$) {
  Let $v_1, v_2, \ldots v_n$ be the example values used
  to infer the structures for the split.
  Let $x_{i,1}x_{i,2}\ldots x_{i,k}$ be the user-specified split for each v_i.
  As in Section 3.2, compute for all structures S_j
  sp_j = space needed to express $x_{1,j}, \ldots x_{n,j}$ using S_j
  Choose the structure S_j with the least value of sp_j.
  for all substrings $w_a \ldots w_b$ of q satisfying S_j do
    DecSpecificity($w_1 \ldots w_{a-1}$, $S_1 \ldots S_{j-1}$)
    DecSpecificity($w_{b+1} \ldots w_m$, $S_{j+1} \ldots S_p$)
}

Figure 11: Two methods of splitting a value.

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 errors to the user, who can apply other transforms to split them, or clean the data further.

4.3.2 Splitting based on inferred structures

Since the structures inferred involve domains with Turing-complete match functions, splitting a value based on these is not easy. The first algorithm of Figure 11, LeftRight, is a simple recursive algorithm for parsing a value that considers the inferred structures from left to right, and tries to match them against all prefixes of the unparsed value. This algorithm is potentially 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 called DecSpecificity (the second algorithm of 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 motivation is that in the initial stages the most specific structures (typically constant delimiters) will match only a few substrings, 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 sum of the description lengths of the (appropri-

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

Figure 10: Parse structures inferred from various split-by-examples

ate substrings) of the example values using the structure. The less specific structures need to be used only after the value has been decomposed into much smaller substrings, and the splitting is not too expensive on these.

To study the effect of parsing according to specificity we ran DecSpecificity, LeftRight, and IncSpecificity on a few structures. IncSpecificity is the exact opposite of DecSpecificity and considers structures starting with the least specific one; it illustrates how crucial the choice of starting structure is. Figure 12 compares the throughput at which one can split values using these methods. We see that DecSpecificity performs much better than the others, with the improvement being dramatic at splits involving many structures.

4.4 Undoing Transforms and Tracking Data Lineage

The ability to undo incorrect transforms is an important requirement for interactive transformation. However, if the specified transforms are directly applied on the input data, many transforms (such as regular-expression-based substitutions and some arithmetic expressions) cannot be undone unambiguously – there exist no "compensating" transforms. Undoing these requires "physical undo", *i.e.*, the system has to maintain multiple versions of the (potentially large) dataset.

Instead Potter's Wheel never changes the actual data records. 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 done "logically," by removing the concerned transform from the sequence and "redoing" the rest before repainting the screen.

This approach also solves the ambiguous data lineage problem of whether a discrepancy is due to an error in the data or because of a poor transform. 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.

5 Related Work

The commercial data cleaning process is based on ETL tools and auditing tools, as described in the introduction. [6, 9] give good descriptions of the process and some popular tools.

There is much literature on transformation languages, especially for performing higher-order operations on relational data [1, 7, 16, 18]. Our horizontal transforms are very similar to the restructuring operators of SchemaSQL [16]. However our focus is on the ease of specification and incremental application, and not merely on expressive power.

The research literature on finding discrepancies in data has focused on two main things: general-purpose algorithms for finding outliers in data (*e.g.* [3]), and algorithms for finding approximate duplicates in data [13, 17, 10]. There has also been some work on finding hidden dependencies in data and correspondingly their violations [14]. Such general purpose algorithms are useful as default algorithms for Potter's Wheel's discrepancy detector. However we believe that in many cases the discrepancies will be domain-specific, and that data cleaning tools must handle these domains extensibly.

A companion problem to data cleaning is the integration of schemas from various data sources. We intend to extend Potter's Wheel with a system that handles interactive specification of schema mappings (such as Clio [19]).

Extracting structure from poorly structured data is increasingly important for "wrapping" data from web pages, and many tools exist in both the research and commercial world (*e.g.* [2, 12, 8]). As discussed in Section 4.3, these tools typically require users to specify regular expressions or grammars; even these are often not sufficient to unambiguously parse the data, so users have to write custom scripts. There have also been some learning-based approaches for automatic text wrapping and segmentation [15, 4]. We believe, however, that a semi-automatic, interactive approach using a combination of graphical operations and statistical methods is more powerful.

There has been some work in the machine learning literature [20, 5] and the database literature [11] on inferring regular expressions from a set of values. However as argued before, for detecting discrepancies it is important to infer structures in terms of generic user-defined domains, in a way that is robust to structural data errors.

6 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 trans-

Example Values	Structure to Split by (Int=Integer, Dbl=Double)	Split Throughput (usecs/value) (DecSpec	LeftRight	IncSpec)
8:45	$<Int><:><Int>$	5.96	9.18	9.18
1997/10/23	$<Int></><Int></><Int>$	11.52	17.95	57.89
12.8.15.14 - - [01/May/2000 ... "GET ... 404 306	$<Dbl '.' Dbl><'- - ['><\xi^*>$	144.8	539.8	27670
12.8.15.14 - - [01/May/2000 ... "GET ... 404 306	$(<Dbl '.' Dbl><'- - ['><\xi^*><Int ' ' Int>)$	219.38	943.8	1525590
12.8.15.14 - - [01/May/2000 ... "GET ... 404 306	$(<Dbl '.' Dbl><- - [><Int/Word/Int><"GET><\xi^*><Int ' ' Int>)$	233.55	1960.95	1036090

Figure 12: Comparison of split throughputs using three methods.

formation, and complex transformation interfaces that often require user programming.

We have described Potter's Wheel, an interactive system for data transformation and cleaning. By integrating discrepancy detection and transformation, Potter's Wheel allows users to gradually build a transformation to clean the data by adding transforms as discrepancies are detected. Users can specify transforms through graphical operations or through examples, and see the effect instantaneously, thereby allowing easy experimentation with different transforms.

We have seen that parsing strings using structures of user-defined domains results in a general and extensible discrepancy detection mechanism for Potter's Wheel. Such domains also provide a powerful basis for specifying Split transformations through example values. In future we would like to investigate specification of other complex transforms such as the Format transform, through examples.

Our focus with Potter's Wheel has so far been on flat, tabular data. However, nested data formats like XML are becoming increasingly common. While much there are many research efforts on transformation and query languages for such data, it would be interesting to investigate graphical and example-based approaches for specifying these.

While we have so far looked at Potter's Wheel as a data cleaning tool, we would like to investigate its effectiveness as a client interface to a interactive query processing system. The transformations applied at the client interface can be viewed as refinements to the ongoing query, and can be fed back into the query processor, thereby combining query specification, execution, and result browsing.

Acknowledgments

The scalable spreadsheet interface that we used was developed along with Andy Chou. Renee Miller and Subbu Subramanian gave us pointers to related work on handling schematic heterogeneities. This work was supported by a grant from IBM Corporation, a California MICRO grant, NSF grants IIS-9802051 and RI CDA-9401156, a Microsoft Fellowship, and a Sloan Foundation Fellowship.

References

[1] S. Abiteboul et al. Tools for data translation and integration. *Data Engg. Bulletin*, 22(1), 1999.

[2] B. Adelberg. NoDoSE — A tool for semi-automatically extracting structured and semistructured data from text documents. In *SIGMOD*, 1998.

[3] A. Arning et al. A linear method for deviation detection in large databases. In *Proc. KDD*, 1996.

[4] V. Borkar, K. Deshmukh, and S. Sarawagi. Automatic segmentation of text into structured records. In *SIGMOD*, 2001.

[5] A. Brazma. Efficient algorithm for learning simple regular expressions from noisy examples. In *Intl. Wksp. on Algorithmic Learning Theory*, 1994.

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

[7] 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.

[8] Cohera Corp. http://www.cohera.com.

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

[10] H. Galhardas, D. Florescu, D. Shasha, and E. Simon. AJAX: An extensible data cleaning tool. In *SIGMOD*, 2000.

[11] M. N. Garofalakis et al. A system for extracting document type descriptors from XML documents. In *SIGMOD*, 2000.

[12] S. Grumbach and G. Mecca. In search of the lost schema. In *ICDT*, 1999.

[13] 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.

[14] J. Kivinen et al. Approximate dependency inference from relations. *Theoretical Computer Science*, 149(1), 1995.

[15] N. Kushmerick. Wrapper induction: efficiency and expressiveness. *Artificial Intelligence*, 118, 2000.

[16] V.S. Lakshmanan et al. SchemaSQL: A language for intereoperability in relational multi-database systems. In *VLDB*, 1996.

[17] M. Lee et al. Cleansing data for mining and warehousing. In *DEXA*, 1999.

[18] R. J. Miller. Using schematically heterogeneous structures. In *SIGMOD*, 1998.

[19] R. J. Miller, L. Haas, and M. A. Hernandez. Schema mapping as query discovery. *VLDB*, 2000.

[20] L. Pitt. Inductive inference, DFAs, and computational complexity. *Analogical and Inductive Inference*, 1989.

[21] V. Raman et al. Scalable spreadsheets for interactive data analysis. In *DMKD Workshop*, 1999.

[22] V. Raman and J. M. Hellerstein. Potter's Wheel A-B-C. At control.cs.berkeley.edu/abc, also UCB CSD-00-1110, 2000.

[23] V. Raman, B. Raman, and J. Hellerstein. Online dynamic reordering for interactive data processing. In *VLDB*, 1999.

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

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

[26] M. Stonebraker and J. M. Hellerstein. Content integration for E-Commerce. In *SIGMOD*, 2001.

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