Towards Automatic Data Format Transformations: Data Wrangling at Scale

DOI:
10.1093/comjnl/bxy118

Towards Automatic Data Format Transformations: Data Wrangling at Scale

Alex Bogatu, Norman W. Paton, Alvaro A.A. Fernandes and Martin Koehler

School of Computer Science, University of Manchester, Manchester, M13 9PL, UK
Email: alex.bogatu@manchester.ac.uk, norman.paton@manchester.ac.uk,
alvaro.a.fernandes@manchester.ac.uk, martin.koehler@manchester.ac.uk

Data wrangling is the process whereby data is cleaned and integrated for analysis. Data wrangling, even with tool support, is typically a labour intensive process. One aspect of data wrangling involves carrying out format transformations on attribute values, for example so that names or phone numbers are represented consistently. Recent research has developed techniques for synthesising format transformation programs from examples of the source and target representations. This is valuable, but still requires a user to provide suitable examples, something that may be challenging in applications in which there are huge data sets or numerous data sources. In this paper we investigate the automatic discovery of examples that can be used to synthesise format transformation programs. In particular, we propose two approaches to identifying candidate data examples and validating the transformations that are synthesised from them. The approaches are evaluated empirically using data sets from open government data.

Keywords: format transformations; data wrangling; program synthesis

1. INTRODUCTION

Data wrangling is the process of data collation and transformation that is required to produce a data set that is suitable for analysis. Although data wrangling may be considered to include a range of activities, from source selection, through data extraction, to data integration and cleaning [1], here the focus is on *format transformations*. Format transformations carry out changes to the representation of textual information, with a view to reducing inconsistencies.

As an example, consider a scenario in which information about issued building permits is aggregated from different data sources. There can be different conventions for most of the fields in a record (e.g. the format of the date when the permit was issued: *2013-05-03* vs. *05/03/2013*; the cost of the building: *83319* vs. *$83319.00*; or the address of the building: *730 Grand Ave* vs. *Grand Ave, Nr. 730*). Such representational inconsistencies are rife within and across data sources, and can usefully be reduced during data wrangling.

Data wrangling is typically carried out manually (e.g. by data scientists) with tool support; indeed data wrangling is often cited as taking a significant portion of the time of data scientists[1]. An example of a tool that supports the authoring of format transformations is Wrangler [2], commercialised by Trifacta, in which data scientists author transformation rules with support from an interactive tool that can both suggest, and illustrate the effect of, the rules. Such an approach to data wrangling should lead to good quality results, but is labour intensive where there are multiple data sources that manifest numerous inconsistencies.

In this paper we address the question *can the production of such format transformations be automated?* Automatic solutions are unlikely to be able to match the reach or quality of transformations produced by data scientists, but any level of automation provides the possibility of added value for minimal cost. We build on some recent work on the synthesis of transformation programs from examples, which was originally developed for use in spreadsheets (e.g. FlashFill [3], BlinkFill [4]). In the commercial realisation of FlashFill as a plugin to Excel, the user provides example pairs of values that represent source and target representations, from which a program is synthesised that can carry out the transformations. The published evaluations have shown that effective transformations can often be produced from small numbers of examples.

[1] http://nyti.ms/1Aqif2X

An issue with this is that there is a need for examples to be provided by users. While this is potentially fine for spreadsheets, where there is typically a single source and target, and the source is of manageable size, identifying suitable examples seems more problematic if there are large data sets or many sources. How do we scale program synthesis from examples to work with numerous sources? The approach investigated here is to identify examples automatically. To this end, we propose two techniques for identifying pairs of values from two different data sources that represent the same real world entity. The first proposal makes use of matching candidates and hypothesised functional dependencies to identify pairs of equivalent values. Specifically, given two datasets S and T in which we identify two functional dependency candidates, $S.a \rightarrow S.b$ and $T.c \rightarrow T.d$, and two matching candidates $(S.a, T.c)$ and $(S.b, T.d)$, we pair together values from the right-hand sides of the functional dependencies, $S.b$ and $T.d$, where their corresponding left-hand sides, $S.a$ and $T.c$, have equal values. For example, in Figure 2 (a) and (b) from Section 3, in order to pair together the values from $S.Date$ and $T.Date$, we need the functional dependency relationships $S.Permit\_Nr. \rightarrow S.Date$ and $T.Permit\_Nr. \rightarrow T.Date$, and the matching instances $(S.Permit\_Nr., T.Permit\_Nr.)$ and $(S.Date, T.Date)$. Notice that the values of the $Permit\_Nr.$ columns have equal values. We argue that the resulting pairs can be used as examples for synthesis algorithms to transform the format(s) of the values from $S.Date$ to the format represent in $T.Date$.

While the previous solution proves to be effective when certain conditions are met, functional dependency candidates are often hard to obtain using specialised tools due to data inconsistencies or simply because such relationships do not exist. To address this possibility, we propose a second, less restrictive technique, based on string similarities and candidate matching relationships, which proves to have comparable effectiveness with better scalability. Specifically, for each matching pair candidate $(S.a, T.b)$, we do a string similarity-based fuzzy pairing of values from $S.a$ and $T.b$. Then we assign a confidence measure to each pair of values and use it to select a subset of pairs as examples for synthesis algorithms. For instance, in Figure 2 from Section 3, we pair the values of $S.Date$ and $T.Date$ that represent the same date using their similarity, without requiring a common tuple identifier such as $Permit\_Nr.$

In summary, the main contributions of this paper are:

(i) the identification of the opportunity to deploy format transformation by program synthesis more widely through automatic discovery of examples;

(ii) the description of two approaches that support (i) - an effective, but conditional algorithm, and a less effective, more scalable technique;

(iii) the evaluation of the above approaches with real world data sets.

Although this paper focuses on a fully automated approach, in which the user is not in-the-loop, the automated approach could be used to support the bootstrapping phase of pay-as-you-go approaches, in which users subsequently provide feedback on the results of the automatically synthesised transformations.

The rest of the paper is organised as follows. Section 2 reviews the work on synthesis programming on which this paper builds. Section 3 describes the first proposal for generating examples based on functional dependency candidates. In Section 4 the previously mentioned technique is evaluated with real world datasets. Section 5 describes an alternative to functional dependency-based examples generation in the form of a string similarity-based technique. We evaluate this method in Section 6 and finally discuss the related work in Section 7 and conclude in Section 8.

This paper is an invited extended version of [5], the main additional contributions being the inclusion of an additional method for generating examples in Section 5, and its evaluation in Section 6.

2. TECHNICAL CONTEXT

In this section, we briefly review the work on synthesis of programs for data transformation on which we build; full details are provided with the individual proposals (e.g. [3, 4]). In essence, the approach has the following elements: (i) a *domain-specific* language within which transformations are expressed; (ii) a *data structure* that can be used to represent a collection of candidate transformation programs succinctly; (iii) an algorithm that can generate candidate transformation programs that correspond to examples; and (iv) a ranking scheme that supports the selection of the more general programs. Starting from a set of user provided examples, where an example represents a pair (e^{in}, e^{out}), with e^{in} the value to be transformed, and e^{out} the expected transformation result, this language can express a range of operations for performing syntactic manipulations of strings such as *concatenation*, e.g. **Concatenate**, which links together the results of two or more *atomic expressions*, e.g. **SubStr, Pos**, or more complex expressions that involve *conditionals* such as **Switch** and *loops*, e.g. **Loop**. A brief description of the available language constructors is provided in Table 1 - more details are provided in [3].

To illustrate the approach in practice, in Figure 1, the user provided the first two rows as examples for transforming the *Address* column and the synthesizer will try to learn one or more programs consistent with the examples. To this end, for each example e_i provided, the algorithm will start by *tokenizing* the input value, e_i^{in}, and the output value, e_i^{out}, using the character classes depicted in Table 2. Then, it will generate all possible programs, expressed using the domain-

Expressions	Description
$Concatenate(e_1 ... e_n)$	Concatenates the results of multiple expressions
$SubStr(v, p_1, p_2)$	The sub-string of v at $p_1 : p_2 - 1$
$Pos(r_1, r_2, c)$	The index t of a given string s such that r_1 matches some prefix $s[0 : t - 1]$, r_2 matches some suffix $s[t : length(s) - 1]$ and t is the c^{th} such match.
$SubStr2(v, r, c)$	The c^{th}occurrence of regular expression r in v
$ConstStr(s)$	The constant string s

TABLE 1: FlashFill expressions

	Address	Transformed Address
1	730 Grand Ave	Grand Ave, Nr. 730
2	5257 W Eddy St	W Eddy St, Nr. 5257
3	362 Schmale Rd	
4	612 Academy Drive	
5	3401 S Halsted Rd	

FIGURE 1: Transformation scenario

specific language, that are consistent with the provided examples. Given that the number of such expressions can be huge, the algorithm chooses the most suitable expressions according to the ranking scheme. For instance, for row 1, in FlashFill, the following expression is a possible inferred program:

```
Concatenate(t1, ConstStr(" ,\ Nr.\ "), t2)
  where:
    t1 ≡ SubStr(v1, Pos(Alph, ε, 1),
                    Pos(EndTok, ε, 1)),
    t2 ≡ SubStr2(v1, Num, 1),
    v1 ≡ "730 Grand Ave"
    EndTok ≡ the end of string
```

The logic here is to extract the street name, as t_1, using the **SubStr** function, i.e., the sub-string that starts at the index specified by the first occurrence of an alphabet token and ends at the end of string, then to extract the street number, as t_2, using the **SubStr2** function, i.e., the first occurring number, and concatenate these two sub-strings separated by the constant string *", Nr. "* using the **Concatenate** function.

Primitive	Regex	Description
Alph	[*a–zA–Z*]+	One or more letters
LowAlph	[*a–z*]+	One or more lowercase letters
UppAlph	[*A–Z*]+	One or more uppercase letters
Num	[0–9]+	One or more digits
AlphNum	[*a–zA–Z*0–9]+	One ore more letters or digits
Punct	[\p{*Punct*}]+	One or more punctuation signs
Space	[\s]+	One or more spaces

TABLE 2: Token primitives

Although for the scenario in Figure 1, the above program is consistent with both examples, it is possible for the algorithm to synthesise more than one expression for a set of provided examples. This happens, especially, when the examples describe different formats according to the primitives from Table 2. In such cases, the final program will include all the synthesised expressions, joined by a conditional directive, e.g. **Switch**. This enables the interpretation of data that is in multiple formats. For the rest of the paper, we describe the formats of the examples using the notion of **format descriptor**:

DEFINITION 2.1. *A **format descriptor** is a unique sequence of regular expression primitives, i.e., from Table 2, that describes one or more values from the column to be transformed, i.e., the source column.*

For example, the format descriptor for values such as *730 Grand Ave* would be <*Num Space Alph Space Alph*> describing a number followed by a space, followed by two words separated by another space.

While the above example illustrates a case in which the program learned is able to correctly transform all the values, this is not always the case. In general, the generated transformations are more likely to succeed to the extent that the following hold: (a) the correct transformation program is expressible in the underlying transformation language, (b) both elements in the example pairs denote values in the domain of the same real-world property, and (c) taken together, the example pairs cover all or most of the formats used for the column.

3. DISCOVERING EXAMPLES - FD BASED SCHEME

The approach described in Section 2 synthesizes transformation programs from examples, where an *example* consists of pairs of source and target values, $\langle s, t \rangle$, where s is a literal from the source and t is a literal from the target. In our running example, s = *730 Grand Ave* and t = *Grand Ave, Nr. 730*. In the spreadsheet setting, given a column of source values, the user provides target examples in adjacent columns, and FlashFill synthesises a program to transform the remaining values in ways that are consistent with the transformations in the examples. Of course, it may take several examples to enable a suitable transformation (or suitable transformations) to be synthesised. In fact, the user needs to provide enough examples to cover all the relevant patterns existing among the values to be transformed.

3.1. Examples Generation

In this section we propose an approach to the automatic identification of examples, drawing on data from existing data sets. Our aim is to use transformation program synthesis in more complex scenarios than spreadsheets. For example, consider a scenario in which we would like to integrate information about issued building permits from several different sources, and for the result to be standardised to a single format per column. Specifically, we want to represent the columns of the resulting dataset using the formatting conventions used in one of the original datasets, which acts as the target. Providing manual examples to synthesise the transformations needed can be a tedious task that requires knowledge about the formats of the values existing in the entire dataset. In our first approach, the basic idea is to identify examples from the different sources, where the source and target values for an attribute can be expected to represent the same information.

To continue with our running example, assume we have two descriptions of an issued building permit, as depicted in Figure 2(a) and (b). To generate a transformation that applies to the *Address* columns, we need to know which (different) values in the source and target *Address* columns are likely to be equivalent. There are different types of evidence that could be used to reach such a conclusion. In the approach described here, we would draw the conclusion that *730 Grand Ave* and *Grand Ave, Nr. 730* are equivalent from the following observations: (i) the names of the first columns in the two tables match (because of the identical sub-string *Permit_Nr.* they share); (ii) there is a functional dependency, on the instances given, *Permit_Nr.* → *Address* in each of the tables; (iii) the names of the fifth columns (*Address*) of the two tables match; and (iv) the values for the first columns in the two tuples are the same. Note that this is a heuristic that does not guarantee a correct outcome – it is possible for the given conditions to hold, and for the values not to be equivalent; for example, such a case could occur if the *Address* attributes of the source and target tables had different semantics.

More formally, assume we have two data sets, source S and target T. S has the attributes $(sa_1, ...sa_n)$, and T has the attributes $(ta_1, ...ta_m)$. We want values from S to be formatted as in T. Further, assume that we know instances for S and T. Then we can run a functional dependency discovery algorithm (e.g. [6]) to hypothesise the functional dependencies that exist among the attributes of S and T. This gives rise to collections of candidates functional dependencies for S and T, $S.FD = \{sa_i \rightarrow sa_j, ...\}$ and $T.FD = \{ta_u \rightarrow ta_v, ...\}$. Note that, in general, sa_i and ta_u can be lists of attributes.

In addition, assume we have a function, $Matches$, that given S and T, returns a set of pairwise matches between the attribute names in S and T, $Matches(S,T) = \{\langle sa_i, ta_j \rangle, ...\}$. $Matches$ can be implemented using a schema matching algorithm, most likely in our context making use of both schema and instance level matchers [7]. In the case where the left hand sides of the two functional dependencies $sa_i \rightarrow sa_j$ and $ta_u \rightarrow ta_v$ are lists of attributes, we say that sa_i *matches* ta_u if both lists have the same number of elements and there are pairwise matches between the attributes of the two lists. Then Algorithm 1 can be used to compute a set of examples for transformations between S and T. If *S.sa* and *T.ta* are list of attributes, then the *join condition* used in the SQL query at lines $8 - 10$ is a conjunction of comparisons between the matching elements. While this may seem restrictive, it is a necessary condition for finding suitable example candidates.

Algorithm 1 Example discovery using functional dependencies.

```
 1: function FdEgsGen(S,T)
 2:     $Egs \leftarrow \{\}$
 3:     for all $sa \in S$ and $ta \in T$ do
 4:         if $\langle sa, ta \rangle \in Matches(S,T)$ then
 5:             for all $(sa \rightarrow sad) \in S.FD$ and
 6:                 $(sa \rightarrow sad) \in T.FD$ do
 7:                 if $\langle sad, tad \rangle \in Matches(S,T)$ then
 8:                     $EgVals \leftarrow$ select distinct
 9:                 S.sad, T.tad from S, T where
10:                 S.sa = T.ta
11:                     $EgPairs \leftarrow (sad, tad, \langle EgVals \rangle)$
12:                     $Egs \leftarrow Egs \cup \{EgPairs\}$
13:                 end if
14:             end for
15:         end if
16:     end for
17:     return $Egs$
18: end function
```

For the example in Figure 2(a) and (b), (c) illustrates the intermediate results used by or created in Algorithm 1.

3.2. Examples Validation

Although Algorithm 1 returns sets of examples that can be used for synthesising transformations (for example, using FlashFill [3] or BlinkFill [4]) there is no guarantee that the transformation generation algorithm will produce effective transformations. The synthesised transformations may be unsuitable for various reasons, e.g., (a) the required transformation cannot be expressed using the available transformation language, (b) the data is not amenable to homogenisation (e.g. because there is no regular structure in the data), or (c) there are errors in the data. As a result, there is a need for an additional validation step that seeks to determine (again automatically) whether or not a

S.Permit_Nr.	S.Date	S.Contractor	S.Cost	S.Address	S.Permit_Type
100484472	2013-05-03	BILLY LAWLESS	83319	730 Grand Ave	Renovation

(a) Source

T.Permit_Nr.	T.Date	T.Cost	T.Contractor	T.Address	T.Type
100484472	05/03/2013	$83319.00	BILLY LAWLESS	Grand Ave, Nr. 730	Renovation

(b) Target

$Matches(S,T)$	$\{\langle S.Permit\_Nr., T.Permit\_Nr.\rangle, \langle S.Date, T.Date\rangle,$ $\langle S.Cost, T.Cost\rangle, \langle S.Address, T.Address\rangle, ...\}$
$S.FD = T.FD$	$\{Permit\_Nr. \rightarrow Date, Permit\_Nr. \rightarrow Cost,$ $Permit\_Nr. \rightarrow Address, Permit\_Nr. \rightarrow Contractor, ...\}$
Generated Examples	$\{\langle S.Date, T.Date, \langle\text{"2013} - 05 - 03\text{"}, \text{"05/03/2013"}\rangle\rangle,$ $\langle S.Cost, T.Cost, \langle\text{"83319"}, \text{"\$83319.00"}\rangle\rangle,$ $\langle S.Address, T.Address, \langle\text{"730 } Grand\ Ave\text{"}, \text{"}Grand\ Ave,\ Nr.\ 730\text{"}...\}$

(c) Partial intermediate and final results from the algorithm

FIGURE 2: Building permits example

suitable transformation can be synthesised.

In our approach, the set of examples returned by Algorithm 1 is discarded unless a k-fold cross validation process is successful. In this process, the set of examples is randomly partitioned into k equally sized subsets. Then, transformations are synthesised, in k rounds, using the examples from the other $k-1$ partitions, and the synthesised transformation is tested on the remaining partition. Although k-fold cross validation can be used with different thresholds on the fraction of correct results produced, in the experiments, we retain a set of examples only if the synthesised transformations behave correctly throughout the k-fold validation process. In our experiments, we used $k = 10$.

4. EVALUATION OF FD BASED SCHEME

In this paper, we hypothesise that the process of transforming data from one format into another using the recent work on program synthesis can be automated by replacing the user-provided examples with those identified by Algorithm 1. In a scenario in which information from multiple, heterogeneous data sets is to be integrated, in Wrangler[2] or FlashFill[3], it falls on the data scientists either to identify values that need to be transformed in one data set and their corresponding versions in a second data set, or to provide the target versions for some of the source values. Our approach removes the user from the process by automating the identification of examples for use by program synthesis.

To illustrate and evaluate this method we use open government web data, available as CSV files, and we build a process pipeline which makes use of several existing pieces of work. Specifically, the process starts by using an off-the-shelf matcher to identify column level matches between a source data set and a target one (*Match*), then for each data set an off-the-shelf profiling tool is used to identify functional dependencies, which are then used together with the previously discovered matching column pairs to generate input-output examples for the desired transformations using Algorithm 1. The resulting sets of examples are then validated using a k-fold cross-validation process, as described in Section 3. Note that, in practice, the first two steps of our pipeline are necessary only if the input data doesn't explicitly contain pre-discovered matching correspondences or functional dependency candidates, e.g. a relational database that contains explicit FDs for its relations.

Finally, the validated pairs are fed to a synthesis algorithm. Note that only the generation of examples is claimed as our main contribution and, hence, evaluated in this section. Although they are relevant to the use of the approach in practice, we do not seek to directly evaluate the off-the-shelf components we use: (i) the effectiveness of the implementation of *Match*; (ii) the effectiveness of the functional dependency detector; or (iii) the effectiveness of the format transformation synthesizer. In all cases, these works have been evaluated directly by their original authors and/or in comparative studies, and we do not seek to replicate such studies here. Rather, we carry out experiments that investigate example discovery, validation and the quality of the synthesised program, for the approach described in Section 3.

4.1. Experimental Setup

In the experiments we use data from the open government data sets listed in Table 3. For each domain, the URLs represent the location of the source and target datasets, respectively. The last two columns illustrate the size (cardinality and arity) of each dataset. Recall from Section 3 that the FD-based technique relies on the information overlap existing between source and target datasets. Consequently, for the evaluation we picked data on similar domains from different sources that have different levels of overlap. For example, the

Domain	URLs	Card.	Arity
Food Hygiene	ratings.food.gov.uk, data.gov.uk	12323 61	7 10
Lobby	data.cityofchicago.org, data.cityofchicago.org (different years)	7542 210	12 19
Doctors/ Addresses	www.nhs.uk, data.gov.uk	7696 17867	19 3
Building Permits	app.enigma.io, data.cityofchicago.org	10000 1245	13 13
Citations	dl.acm.org/results.cfm, ieeexplore.ieee.org	1072 2000	16 11

TABLE 3: Data sources

source dataset for the *Food Hygiene* domain contains details about food hygiene ratings from 2015 across the whole U.K., while the target dataset illustrates information from a certain area, e.g. a single council or county. Therefore the overlap between source and target datasets might not be very large. In other words, the size of the source/target is not an accurate estimator of the expected overlap.

For Matching, we used COMA 3.0 Community Edition [2], with schema and instance level matchers, and the following parameter values: *MaxN = 0* and *Threshold = 0.2*. The first parameter specifies that attributes can only be in 1:1 matching relationships with each other. The second value specifies the minimal confidence a correspondence between two elements must reach in order to be accepted as a match. For functional dependency discovery we used HyFD[3][6] with the *Null Semantics* setting set to $null \neq null$. This last setting is needed because real-world data often contains *null* values. So for a schema $S(A, B)$, if there are two tuples $r1 = (null, 1)$ and $r2 = (null, 2)$, if $null = null$ then $A \rightarrow B$ is not a functional dependency. In order to avoid discarding functional dependencies in such situations we assume $null \neq null$. For synthesising and running transformations, we used a Java implementation of FlashFill described in [8]. All experiments were run on Ubuntu 16.04.1 LTS installed on a 3.40 GHz Intel Core i7-6700 CPU and 16 GB RAM machine.

4.2. Experiment 1

Can Algorithm 1 identify candidate column pairs? To evaluate this, we report the precision ($\frac{TP}{TP+FP}$) and recall ($\frac{TP}{TP+FN}$) of the algorithm over data sets where there are known suitable candidates. For this experiment, *true positive* (TP), *false positive* (FP) and *false negative* (FN) are defined as: TP – the two columns represent the same concept; FP – the two columns do not represent the same concept; and FN – a correct result that is not returned. The ground truth for computing precision and recall in this experiment was manually created for the domains in Table 3 - a column from the source and a column from the target create a pair in the ground truth if they represent the same concept, e.g. *Business Name* columns in Food Hygiene, *Author Names* columns in Citations, etc.

Examples of candidate columns pairs that are identified by Algorithm 1, are given in Table 4. The last column illustrates the number of unique example pairs generated by Algorithm 1 for each case.

The results for this experiment are presented in Table 5, for the data sets in Table 3. In general, both precision and recall are high. In the case of the Building Permits domain, there are 3 attributes in each dataset representing a cost (see Row 9 in Table 4 for an example). Although there are 9 possible source column-target column alignments, Algorithm 1 was able to identify the correct ones and returned no false positives. Food Hygiene precision is *0.86* due to a FP match identified by COMA. An example of this match is on the second row of Table 4. The two columns have the same name (*AddressLine1*), but different semantics. The precision and recall for Citation are reduced by one FP match identified by COMA (represented in the last row of Table 3) and 2 FNs (i.e. 2 pairs of columns that were not reported as matches by COMA).

4.3. Experiment 2

Is the validation process successful at identifying problematic column pairs? A column pair is problematic if we cannot synthesise a suitable transformation. Note that in *Experiment 1* we measured the effectiveness of Algorithm 1 at identifying correct column pairs. Here we check whether the alignment of values, i.e. the result of the select at lines 8-10 in Algorithm 1, denotes suitable examples for synthesizing a correct transformation. To evaluate this we run a k-fold cross validation task, with $k = 10$, on each pair of columns from Experiment 1. We investigate here the candidate pairs for which validation has failed; the cases for which validation has passed are discussed in Experiment 3. A pair of columns is considered to pass the validation step if all transformations are correct for each of the 10 iterations.

The fraction of the candidate column pairs that pass validation is reported in the last column of Table 5. Column pairs have failed validation for the following reasons: (i) The matched columns have different semantics, and thus incompatible values, for which no transformation can be produced. This is the case for *2* of the *8* column pairs that fail validation (for example see Row 2 and Row 14 in Table 4). (ii) The matched columns have the same semantics, but FlashFill has failed to synthesise a suitable transformation. This is the case for *2* of the column pairs that fail validation (for example, consider the lists of author names from Row 11 in Table 4). (iii) There are issues with the specific values in candidate columns. This is the case for *4* of

[2] http://bit.ly/2fLVvtl

[3] http://bit.ly/2f5DwJW

#	Domain	Src Semantics	Source Example	Target Semantics	Tgt Example	# egs.
1	Food Hyg.	Rating Date	2015-12-01	Rating Date	01/12/2015	57
2	Food Hyg.	Building Name	Royal Free Hospital	Street	Pond Street	59
3	Food Hyg.	Business Name	Waitrose	Business Name	Waitrose	60
4	Lobby	Person Name	Mr. Neil G Bluhm	Person Name	Bluhm Neil G	206
5	Lobby	Phone Nr.	(312) 463-1000	Phone Nr.	(312) 463-1000	191
6	Docs/.Addrs.	Address	55 Swain Street,Watchet	Street	Swain Street	41
7	Docs./Addrs.	City	Salford	City	Manchester	28
8	Build. Permits	Address	1885 Maud Ave	Address	Maud Ave,Nr. 1885	26
9	Build. Permits	Cost	6048	Cost	$6048.00	22
10	Build. Permits	Issue Date	2014-06-05	Issue Date	06/05/2014	26
11	Citations	Author Names	Sven Apel and Dirk Beyer	Author Names	S. Apel; D. Beyer	56
12	Citations	Conf. Date	” ”	Conf. Date	2-8 May 2010	32
13	Citations	Pub. year	2011	Pub. year	2011	56
14	Citations	Nr. of pages	10	Start page	401	56

TABLE 4: Transformation candidates

Domain	Candidates	Precision	Recall	Valid
Food Hyg.	6	0.83	1.00	5/6
Lobby	9	1.00	1.00	9/9
Docs./ Addrs.	2	1.00	1.00	1/2
Build. Perm.	12	1.00	1.00	12/12
Citations	7	0.86	0.75	1/7

TABLE 5: Experiments 1 and 2

the column pairs that fail validation (as an example consider the missing information from Row 12 in Table 4) and inconsistent values (in Row 7 in Table 4).

It is important to note that in practice, the effectiveness of the off-the-shelf tools that we used here can be impacted by characteristics of the data such as the ones exemplified above. This evaluation shows that the validation method we employ is able to filter out example sets that otherwise would produce invalid transformations or no transformations at all.

4.4. Experiment 3

Do the synthesised transformations work on the complete data sets (and not just the training data)? To evaluate this, we report the precision and recall of validated transformations in Table 6; the missing row numbers are the examples from Table 4 for which the transformations failed validation. In computing the precision and recall, we use the following definitions: TP – the transformation produces the correct output; FP – the transformation produces an incorrect output; FN – the transformation results in an empty string.

Of the *28* validated transformations from Table 5, all but *6* are identity transformations, i.e. the source and target data values are the same (e.g.Rows 3, 5 and 13 in Table 6). This can often happen in practice. For instance, Row 13 represents the year for a publication which is most commonly represented as a four digit number. In such cases, FlashFill proved to be able to identify that the transformation is only a copying operation. Of the *6* cases where the values are modified, the precision and recall are both *1.0* in 3 cases (Rows 1, 8 and 10 in Table 6). For rows 1 and 10 the transformations rearrange the date components and replace the separator. Our experiments confirmed the results of [3] according to which FlashFill is effective in such simple cases. For Row 8, the transformation is more complicated given that the street name does not always have a fixed number of words, or that the street number can have several digits. In this case Algorithm 1 was able to identify enough examples to cover all the existing formats. There were problems with the transformations generated in 3 cases. For Row 4, a few source values do not conform to the usual pattern (e.g. the full stop is missing after the salutation). For Row 9, not all source values are represented as integers, giving rise to incorrect transformations. For Row 6, similarly to Row 9, the examples do not cover all the relevant address formats, i.e. 41 examples are used to synthesise a program to transform a rather large number of values (approx. 7700).

4.5. Discussion

The technique evaluated above proves to be effective in scenarios where certain conditions are met. The most important of these is that the source and target contain overlapping information on some of the tuples, i.e. the left-hand sides of the functional dependencies used in Algorithm 1, and format diversity on the corresponding right-hand side.

Another important condition is for FlashFill to be able to synthesise transformations from the pairs of generated examples. The evaluation shows that as long as these conditions are met, we can delay the need for user intervention in the cleaning process by synthesizing and applying some transformations automatically.

#	Src Semantics	Src Value	Tgt Semantics	Tgt Value	Precision	Recall
1	Rating Date	2015-12-01	Rating Date	01/12/2015	1.0	1.0
3	Business Name	Waitrose	Business Name	Waitrose	1.0	1.0
4	Person Name	Mr. Neil G Bluhm	Person Name	Bluhm Neil G	0.98	0.84
5	Phone Nr.	(312) 463-1000	Phone Nr.	(312) 463-1000	1.0	1.0
6	Address	55 Swain Street,Watchet	Street	Swain Street	0.68	1.0
8	Address	1885 Maud Ave	Address	Maud Ave,Nr. 1885	1.0	1.0
9	Cost	6048	Cost	$6048.00	0.97	1.0
10	Issue Date	2014-06-05	Issue Date	06/05/2014	1.0	1.0
13	Pub. year	2011	Pub. year	2011	1.0	1.0

TABLE 6: Experiment 3

5. DISCOVERING EXAMPLES - WEIGHTED SCHEME

The technique described in the previous section uses a set of hypothesised functional dependencies to pair values from a source dataset S : $(sa_1, ...sa_n)$ and a target dataset $T : (ta_1, ...ta_m)$, that represent the same real world entity. While the evaluation from Section 4 showed that Algorithm 1 can be effective in some scenarios, there are cases where the conditions required by the algorithm are not satisfied. For instance, candidate functional dependencies may be missing or state-of-the-art algorithms are unable to find them due to inconsistencies in values. Furthermore, the number of example pairs generated by Algorithm 1 can be very large, while the number of cases covered by the examples is small. For instance, for row 1 in Table 4, the algorithm generated 57 example pairs, all describing a single date format. Since the complexity of synthesis algorithms such as FlashFill is known to be *exponential in the number of examples and high degree polynomial in the size of each example*[9], not only is it the case that many such examples are useless, but they increase the runtime of the synthesis process.

To address these scenarios, in this section we propose an approach to the automatic identification of examples using syntactic similarities existing between values of matching column pairs, as returned by the $Matches$ function introduced in the previous section. We also propose an incremental example selection algorithm which, once the example pairs are generated, selects a subset from which there is evidence that FlashFill can synthesise effective transformations.

The objective of the example discovery technique, described in subsection 5.1, is, given two columns from different tables, to heuristically identify pairs of values that are equivalent. To this end, we start by *tokenizing* each column value, where each token is a substring of the original value delimited by punctuation or spaces. The tokens enable the *grouping* of values into blocks, where each block contains values from both columns with common tokens. An intra-block, pair-wise *comparison* of values is conducted to determine potentially equivalent instances. Given the fact that the syntactic comparison does not guarantee the optimal pairing of equivalent values, each candidate pair obtained will have an associated weight. This measure is used in the second phase, described in subsection 5.2, to determine the minimal sub-set of candidate value pairs that can be used as examples to synthesise an effective transformation program.

5.1. Weighted Examples Generation

Consider a pair of columns $(S.sa, T.ta)$, described in Figure 3, between which there is a matching correspondence as returned by the $Matches$ function. The objective is to identify which values in sa and ta are likely to be equivalent. Starting from the assumed matching relationship between the two columns, we can pair values based on their string representations and determine that, for example, *730 Grand Ave* and *Grand Ave, Nr. 730* represent the same address.

More formally, assume we have two datasets, source S and target T. S has the attributes $(sa_1, ...sa_n)$, and T has the attributes $(ta_1, ...ta_m)$. We want values from S to be formatted as in T. Further, assume we have a function, $Matches$, that returns a set of pairwise matches between the attribute names in S and T. Then, we can use Algorithm 2 to pair values from sa_i and ta_j that are likely to be equivalent. The obtained pairs can then be used as examples for synthesis algorithms to generate transformation programs that will format values from sa_i as in ta_j. Figure 3 depicts a pair of matching columns, each with 4 values. Notice that in a real world case, the number of tuples of sa and ta will most likely be different. For the rest of this paper, we define the **example set** as the collection of pairs (e_i^{in}, e_i^{out}), with e_i^{in} a value from sa and e_i^{out} its equivalent value from ta. For example, if e_i^{in} is row 1 from sa, then e_i^{out} is row 4 from ta. Analogously, we define the **test set** as the collection of values from sa for which there are no corresponding values identified in ta, i.e. the set of values that will be transformed by the transformation program synthesised using the example set.

When applied on the column pair from Figure 3, Algorithm 2 comprises of the following steps:
Tokenise - lines 4,5: For each value in columns sa and ta, the $Tokenise$ method transforms the string

Algorithm 2 Weighted examples discovery using string similarities.

```
 1: function wEgsGen(S,T)
 2:     $Egs \leftarrow \{\}$
 3:     for all $\langle sa, ta \rangle \in Matches(S, T)$ do
 4:         $Tok_s \leftarrow Tokenise(sa)$
 5:         $Tok_t \leftarrow Tokenise(ta)$
 6:         $Idx \leftarrow Index(Tok_s, Tok_t)$
 7:         $Egs \leftarrow Egs \cup WeightedPairing(Idx)$
 8:     end for
 9:     return $Egs$
10: end function
```

#	sa
1	730 Grand Ave
2	93 Roland St
3	21 Duke Ave
4	44 Park Rd

#	ta
1	Robinson St, Nr. 14
2	Park Rd, Nr. 44
3	Grand Central Ave, Nr. 331
4	Grand Ave, Nr. 730

FIGURE 3: Matching column pair

Algorithm 3 The WeightedPairing function of Alg. 2

```
 1: function WeightedPairing(Idx)
 2:     for all $e \in Idx.entries$ do
 3:         $pairs \leftarrow SimPairing(e)$
 4:         $maxPair \leftarrow null$
 5:         for all $p \in pairs$ do
 6:             if $maxPair.weight < p.weight$ then
 7:                 $maxPair \leftarrow p$
 8:             end if
 9:         end for
10:         $Egs \leftarrow Egs \cup \{maxPair\}$
11:     end for
12:     return $Egs$
13: end function
```

#	sa - tokenised
1	[730,Grand,Ave]
2	[93,Roland,St]
3	[21,Duke,Ave]
4	[44,Park,Rd]

#	ta - tokenised
1	[Robinson,St,Nr,14]
2	[Park,Rd,Nr,44]
3	[Grand,Central,Ave,Nr,331]
4	[Grand,Ave,Nr,730]

FIGURE 4: Tokenised values

representation of the value into an array representation, where each element of the array is a token as defined by the list of regular-expression-based primitives from Table 2. The last primitives, *punctuation* and *space*, are being used as separators. The intuition is that a punctuation sign has small significance in determining the similarity of two values, e.g. the separators have a small weight in determining the equivalence of *24/04/1989* and *04.24.1989*. To illustrate this, Figure 4, describes the tokenised representations of values from Figure 3.

Index - line 6: The tokens are then used to create an inverted index $I(sa, ta)$ for the pair of matching columns. For each token t, the inverted list $I[t]$ is a list of all values from sa and ta which contain t. For example, if t =*Ave*, $I[t]$ = [sa_i.*'730 Grand Ave'*,ta_j.*'Grand Ave, Nr. 730'*, ...] - all values from *sa* and *ta* containing token *Ave*. Figure 5 shows three index entries for tokenised values from Figure 4 - each row denotes an index entry with the *Token* column being the key and *Inverted list* being the list of values containing the token.

Weighted Pairing - line 7: Each source value in each index entry is paired, by Algorithm 3, with the most similar target value according to a confidence measure described below. For instance, if t = *Ave* and $I[t]$ = [*sa.'730 Grand Ave'*, *ta.'Grand Ave, Nr. 730'*, *ta.'Grand Central Ave, Nr. 331'*], then the returned pair would be (*sa.'730 Grand Ave'*, *ta.'Grand Ave, Nr. 730'*). Similar examples can be seen in Figure 6 for indexed values from Figure 5.

Weight - Each pair of values $(sa.x, ta.y)$, returned at the previous step, will have a weight σ assigned to it, computed in the *SimPairing* method (line 3 in Algorithm 3), as described by Equation 1. We define θ, in Equation 2, as the overlap coefficient between two strings of characters which divides their intersection by the size of the smaller of the two sets[4]; M as the number of tokens under which the pair is indexed; IDF_{rk}, in Equation 3, as the inverse document frequency of a token r_k, under which the pair $(sa.x, ta.y)$ has been indexed, computed as a logarithmically scaled fraction obtained by dividing the total number of values from both columns by the number of values containing token r_k; and ϕ as a similarity measure[5] of two strings obtained by removing token r_k from the two original values (if by removing the token from one of the values the result is the empty string, then ϕ is computed using the original strings). In Equation 1, θ is used to penalise pairs with very dissimilar values, and IDF_{rk} is used to weight down pairs indexed under a very common token, e.g. *St, Ave.* For instance, pair (*sa.'730 Grand Ave'*,*ta.'Grand Ave, Nr. 730'*) from Figure 6 has been indexed under two tokens, *Grand* and *Ave.* The first entry will have a higher weight because there are four occurrences of *Ave* in Figure 3, while only three of *Grand.* Notice that the proposed weight is not intended to be a normalised similarity measure between two strings, but identify the pairs of value instances that are more likely to be valid examples for synthesis algorithms.

$$\sigma = \theta(sa.x, ta.y) \times \max_{1 \leq k \leq M} (IDF_{rk} \times \phi(sa.x, ta.y)) \quad (1)$$

[4] In computing the overlap we only consider alpha-numeric, lowercase characters

[5] In our experiments we used the Euclidean distance metric

Algorithm 4 Examples selection

```
function EgsSelection(Egs)
    $Egs \leftarrow Sort(Egs)$
    $minEgs \leftarrow InitEgs(Egs)$
    $p \leftarrow Synthesise(minEgs, Egs \setminus minEgs)$
    while $p \neq \epsilon$ do
        if $GetFormat(p.in) \notin GetFormats(Egs \setminus minEgs)$ then
            return $REJECT\ Egs$
        end if
        $minEgs \leftarrow minEgs \cup \{p\}$
        $p \leftarrow Synthesise(minEgs, Egs \setminus minEgs)$
    end while
    return $minEgs$
end function
```

Token	Inverted list
Grand	[*sa*.’730 Grand Ave’,*ta*.’Grand Central Ave, Nr. 331’,*ta*.’Grand Ave, Nr. 730’]
Ave	[*sa*.’730 Grand Ave’,*sa*.’21 Duke Ave’, *ta*.’Grand Central Ave, Nr. 331’, *ta*.’Grand Ave, Nr. 730’]
Park	[*sa*.’44 Park Rd’,*ta*.’Park Rd, Nr. 44’]

FIGURE 5: Indexed values

Algorithm 5 The *Synthesise* method of Alg. 4

```
function Synthesise(Egs,TestEgs)
    $\tau \leftarrow FlashFill(Egs)$
    for all $p \in TestEgs$ do
        if $\tau(p.in) \neq p.out$ then
            if $IsFirst(p)$ then
                return $p$
            end if
        end if
    end for
    return $\epsilon$
end function
```

Token	Value pairs
Grand	[(*sa*.’730 Grand Ave’,*ta*.’Grand Ave, Nr. 730’)]
Ave	[(*sa*.’730 Grand Ave’,*ta*.’Grand Ave, Nr. 730’), (*sa*.’21 Duke Ave’, *ta*.’Grand Central Ave, Nr. 331’)]
Park	[(*sa*.’44 Park Rd’,*ta*.’Park Rd, Nr. 44’)]

FIGURE 6: Paired values

$$\theta(X,Y) = \frac{|X \cap Y|}{\min(|X|,|Y|)} \tag{2}$$

$$IDF_{rk} = \log\left(\frac{N}{n_{rk}}\right) \tag{3}$$

To illustrate the behaviour of Equation 1 in practice, Table 7 shows examples of cases we encountered and their respective coefficient values. In the table, the first column depicts the token, i.e., key, under which the pair from the second column has been stored in the index, as presented in the *Index* method above. The last three columns are the values for θ - overlap coefficient; IDF_{r_k} - IDF of the key token; and ϕ - string similarity. For both θ and ϕ we only considered alpha-numeric characters, i.e., non-separators. In computing ϕ, the key token is also removed from the strings, unless the result is the empty string. All coefficients are computed using lowercase strings.

Of particular interest in Table 7 is row 2 where, while the two strings represent the same address, the key token is very common, therefore having a small IDF. Thankfully, the same pair will appear in a bucket identified by token *morgan* as well, which appears less frequent than *st*, hence having a higher IDF - note that in equation (1) we only consider the maximum product of IDF and ϕ. The difference in string sizes for the third row leads to a relatively small ϕ, but the high overlap and token IDF compensate. The last two rows indicate pairs that should not be considered examples, i.e. false positives. Here the dissimilarity of strings is being demonstrated by a low value for at least one coefficient. In general, high values for θ and/or ϕ denote strong evidence of the validity of the example instance, while a combination of low/mid-range values suggests false positives. Whenever the dissimilarity is not caught by equation (1), we rely on Algorithm 4 to eliminate false positives, as described in the next section.

The final example set, returned by Algorithm 3 for the two columns from Figure 3 will include three pairs, i.e. the set of distinct pairs from Figure 6, each one having an assigned weight, which will be used in the selection process as described in the next section.

5.2. Incremental Examples Selection

Algorithm 2 returns sets of examples that can be used for synthesizing transformation programs using FlashFill [3] or BlinkFill [4], but, as with the functional dependency based technique, there is no guarantee that effective transformations will be produced from these examples. Furthermore, Algorithm 2 can generate a high number of example pairs (the smallest number of examples obtained in our experiments was 21) which can exponentially increase the synthesis time. As a result, we propose a selection technique that aims to refine the example set produced by Algorithm 2 by selecting only the smallest subset for which there is evidence that it will produce effective transformations. We define the evidence as the effectiveness of the synthesis algorithms, using the selected subset, to produce a transformation program that will correctly transform the original set of examples generated by Algorithm 2.

More formally, given an example set $E = \{e_1, e_2, ..., e_n\}$ produced by Algorithm 2, with $e_i =$

Token r_k	Value pair	θ	IDF_{r_k}	ϕ
2016	*(2016-01-04, 04/01/2016)*	1.0	1.27	1.0
st	*(71 Morgan St, Morgan St, Nr. 71)*	1.0	0.32	0.60
mckeon	*(Howard P. Mckeon (R-Calif), Mckeon)*	1.0	2.67	0.55
robert	*(Kruzel, Robert, Robert L Danley Jr)*	0.5	3.84	0.36
neal	*(Tatum O'Neal, neal_h_brian)*	0.33	2.55	0.29

TABLE 7: Weight examples

#	**sb**
1	Jim K. Lee
2	Billy W. Tauzin
3	Clarence Thomas
4	Joe Ackerman-Specter
5	Tommy Thompson
6	George W. Bush
7	Alphonso Jackson
n	...

#	**tb**
1	Benjamin-Stock
2	Bush
3	Millender-Cramer
4	Jackson
5	Thomas
6	Lee
7	Ackerman
m	...

FIGURE 7: Matching column pair

Source Value	**Target Value**
Condoleezza Rice	Rice
Juanita Millender-Cramer	Millender-Cramer
George W. Bush	Bush

FIGURE 8: Initialization phase

$\left(e_i^{in}, e_i^{out}\right)$, the incremental selection technique, described in Algorithm 4, returns an example set $F = \{f_1, f_2, ...f_m\}$, with $F \subseteq E$ and $m \leq n$, such that when F is used as the input to a synthesis algorithm produces a set of transformation expressions $T = \{\tau_1, \tau_2, ..., \tau_p\}$ so that $\forall e_i \in E, \exists \tau_j \in T$ with $\tau_j\left(e_i^{in}\right) = e_i^{out}$.

The objective of Algorithm 4 is twofold: (i) to purge pairs of values returned by Algorithm 2 which are not equivalent; and (ii) to minimise the set of examples by purging redundant pairs. For (ii), recall from Section 2 that a synthesised program is more likely to correctly transform all the test values if the example pairs cover all of the formats existing in the test data. But if there are too many example pairs covering the same format, this will exponentially increase the complexity of the synthesis. In Algorithm 4, we consider two examples to be *redundant* if they cover the same format, e.g. the first 3 rows of Figure 9 describe the same format and a single transformation: *extract the second token from a string of two tokens.*

To illustrate the technique in practice, consider the pair of columns depicted in Figure 7. The values of *sb* represent full names of U.S. congress members, while *tb* contains person last names. The objective is to generate examples for synthesizing a transformation program that will extract the last names from *sb*. For this purpose, we start by applying Algorithm 2, which will produce a set of example pairs, 11 of which are represented in Figure 9, together with their corresponding weights. Next, Algorithm 4 follows the steps described below:

Sort - line 2: The set of example pairs returned by Algorithm 2 is sorted into descending order based on the pair weights. The result of this step is shown in Figure 9.

Initialise Examples - line 3: The algorithm starts by selecting one example pair, i.e. the pair with the highest weight, *for each format present in the example set* - where a format is defined by its format descriptor - see the format descriptor definition from Section 2. Specifically, the pairs depicted in Figure 9 describe three formats, i.e. rows 1-3 (< *Alph Space Alph* >), rows 4-7 (< *Alph Space Alph Punct Alph* >), and rows 8-9 (< *Alph Space UppAlph Punct Space Alph* >). Note that the pairs with the same format are grouped together for clarity - this might not be the case in a real world scenario. Note also that both the punctuation and space count as token types rather than separators (as was the case for the tokenization step in previous section). The three examples from Figure 8 have the highest weights for their respective formats.

Synthesise - lines 4,10: This method, illustrated in Algorithm 5, takes as input a set of example pairs, e.g. the pairs with the highest weights as returned by the *InitEgs* method, and a set of test pairs, e.g. the pairs with lower weights than the ones considered as examples. Then, a transformation program is synthesised using FlashFill from the example set (line 2 in Algorithm 5). The resulting program is then tested against each pair from the test set. If the result of a transformation is different from the expected test output (line 4 in Algorithm 5), and if there has not been a previous value describing the same format which was correctly transformed (line 5 in Algorithm 5), then that failing pair is returned. The *IsFirst*(p) method at line 5, checks if pair p has the highest weight for the format it describes.

Increment - line 9: Every time the *Synthesise* method returns a failing pair, it is added to the previous set of examples. The intuition is that the initial set of examples did not cover the format described by the failing pair, therefore, by considering the pair an example, the format will be covered.

Halting Condition - line 6: Algorithm 4 ends when there are no failing test pairs, or when the halting condition is met: all the pairs describing a failing format have been used as examples. This means that there are not enough examples for that format, the algorithm returns and the example set is rejected.

To continue with our US Congress example, when *Synthesise* is called at line 4 in Algorithm 4, the

#	sb	tb	W
1	Condoleezza Rice	Rice	2.21
2	Michael Leavitt	Leavitt	2.16
3	Clarence Thomas	Thomas	2.09
4	Juanita Millender-Cramer	Millender-Cramer	1.76
5	Joe Ackerman-Specter	Ackerman	1.34
6	Tammy Liu-Vitter	Liu-Vitter	1.32
7	Walter Ben-Stock	Benjamin-Stock	0.84
8	George W. Bush	Bush	0.81
9	Joe K. Pitts	Pitts	0.79

FIGURE 9: Algorithm 2 sorted results

pairs from Figure 8 will be used as the example set to synthesise a transformation program τ, which will then be tested against the rest of the values from Figure 9. Notice that for row 5 of Figure 9, it is likely that the transformation synthesised from the examples depicted in Figure 8 will fail, i.e., τ(*Joe Ackerman-Specter*) $\neq$ *Ackerman*, because *Ackerman* is not the last name. The pair at row 4 has the highest weight from the test pairs describing the same format f, meaning that f has not been covered by the previous example set, so the pair will be added to the example set at line 9 in Algorithm 4, and a new iteration started. Of particular interest is row 7 in Figure 9. Notice that, given the values exemplified so far, there is no transformation program that can transform *Walter Ben-Stock* to *Benjamin-Stock*. If the previous pair (row 6), which has a higher weight, is correctly transformed, i.e. τ(*Tammy Liu-Vitter*) =*Liu-Vitter*, then the pair at row 7 will be considered a *false-positive* returned by Algorithm 2, ignored, and the algorithm will continue towards a successful result. This result will include rows 1, 4, 5, and 8 of Figure 9, i.e. the minimal set of examples that can produce a transformation program that can correctly transform the rest of pairs. Otherwise, if τ(*Tammy Liu-Vitter*) $\neq$*Liu-Vitter* and τ(*Walter Ben-Stock*) $\neq$*Benjamin-Stock*, the halting condition is met, i.e. there are no more pairs for the format described by these two failing pairs, and the example set generated by Algorithm 2 is rejected.

At this point we consider that there is no need for further validation of the returned set of examples, e.g., employing a k-fold cross-validation step similar to the one from Section 3. The intuition is that if Algorithm 4 was successful, there is enough evidence to consider that FlashFill can synthesise a transformation program that will correctly transform the source values covered by the obtained examples. Furthermore, if applied on the output of Algorithm 4, the cross-validation, as described in Section 3, will fail because it requires at least two example pairs per format in the candidate example set. In other words, the validation technique requires *redundant* example candidates which is exactly what Algorithm 4 tries to remove.

Domain	URLs	Card.	Arity
Food Hygiene	ratings.food.gov.uk, data.gov.uk	12323 61	7 10
Lobbyists	data.cityofchicago.org, data.cityofchicago.org	7542 210	12 19
Doctors/ Addresses	www.nhs.uk, data.gov.uk	7696 17867	19 3
Building Permits	app.enigma.io, data.cityofchicago.org	10000 1245	13 13
US Congress	opensecrets.org govtrack.us/congress	620 11870	9 10
Restaurants	app.enigma.io data.ny.gov	25000 21000	19 18
Movies	imdb.com imdb.com	75000 15000	4 5
Employment	data.cityofchicago.org data.cityofchicago.org	31000 12000	8 7

TABLE 8: Data sources used in experiments

6. EVALUATION OF WEIGHTED SCHEME

In this section we evaluate the effectiveness of Algorithms 2 and 4. As was the case with the functional dependency based proposal, we hypothesise that the task of providing examples by the user, which is often required in current wrangling tools, can be partly replaced by Algorithms 2 and 4. Note that only the examples generation technique, from Algorithms 2 and 3, and the examples selection technique from Algorithms 4 and 5, are claimed as our contributions. We use an off-the-shelf schema matcher to identify column level matches between a source data set and a target data set, and the implementation of a synthesis algorithm described in [8] to determine the transformation programs, but we do not evaluate the effectiveness of these solutions. Rather, we carry out experiments that investigate the approaches on examples generation and selection described in the previous section, and the quality of the transformation programs synthesised from the results of our approaches.

6.1. Experimental setup

In the experiments we used publicly available data from 9 different domains, describing 5 types of data in different formats, such as *addresses*, *monetary values*, *person names*, *dates*, and *numbers*. Table 8 summarises the used datasets. For each domain, the URLs represent the location of the source and target datasets, respectively. The last two columns illustrate the size (cardinality and arity) of each dataset.

Once again, similarly to the experiments in Section 4, for *Matching* we used COMA 3.0 Community Edition with a similar set of parameters. For synthesis, we used the FlashFill implementation described in [8]. All experiments were run on Ubuntu 16.04.1 LTS installed on a 3.40 GHz Intel Core i7-6700 CPU and 16 GB RAM machine.

6.2. Experiment 1

The question we try to answer in this section is: *does the synthesis algorithm produce effective transformation programs from the example pairs generated by Algorithms 2 and 4?* This is equivalent to the question analysed in Experiment 3 of Section 4. Analyzing how effective the weighted scheme is at identifying candidate column pairs, i.e., similar to Experiment 1 from Section 4, would mean to measure the effectiveness of *Match*, since Algorithm 2 produces example instances for each matching pair returned by *Match*. While the matching function is an essential part of our end-to-end method, schema matching evaluation is not our primary focus in this paper. As for the validation part, the weighted approach doesn't employ a k-fold cross validation stage and relies on Algorithm 4 to discard candidate pairs if there is not enough evidence for synthesizing transformations (as described in the previous section). Some candidate pairs for which example selection failed are discussed below.

For each matching column pair returned by the *Matches* function, we first run Algorithm 2, the result is passed to Algorithm 4, then, using the output from the latter and the FlashFill implementation, a transformation program is synthesised and applied on the test set. Recall that, for each matching column pair (sa, ta), the test set represents the set of values from sa for which there are no corresponding values from ta determined by Algorithm 2. We report the precision and recall of the result of transformations applied on each test set using the following definitions: TP - the transformation produces the correct output; FP - the transformation produces an incorrect output; FN - the transformation resulted in an empty string. The results of the experiment are illustrated in Table 9. Note that the cases illustrated here include only candidate column pairs for which the synthesised transformation is different from the identity transformation, e.g., rows 3 and 13 from Table 4. Our proposed algorithms can trivially identify examples for the identity transformation, i.e. where source and target strings are the same, and we have seen in Section 4 that FlashFill is able to synthesise such simple transformations. Therefore, our focus here is on more challenging ones.

The columns from Table 9 (starting with the second column) represent the domain of the matching column pair, their data type, a source instance value, the corresponding target instance value, the number of example pairs returned by Algorithm 2, the number of example pairs returned by Algorithm 4, the precision, and the recall, respectively.

6.3. Discussion

Of the 14 cases exemplified in Table 9, Algorithm 4 rejected 3 example sets: rows 3, 9, and 11. For rows 3 and 11 the halting condition was met, while in the case of row 9, FlashFill was unable to synthesise a transformation program. For the rest of the matching column pairs, the recall descended below 0.98 only in one case: row 2. This is due to the fact that person names often contain common tokens which can lead to highly weighted false positives returned by Algorithms 2 and 4. This is also true for precision. In fact, all of the cases with precision lower than 0.80 describe person names as well. This means that providing false positive examples to FlashFill decreases the accuracy of transformations.

Notice that columns 6 and 7 from Table 9 denote large differences between the number of example pairs generated by Algorithm 2 and the pairs selected by Algorithm 4. This suggests that many of the pairs generated by the former are covering a relatively small number of formats. For instance, for the *Food Hygiene* domain, all source values of the 57 example pairs generated by Algorithm 2 are describing a single date format, e.g., row 1. Therefore, Algorithm 4 returned a single pair. It is important to understand that while the source values of the example pairs contain dates in a single format, the test set might contain values in different formats. If that was to be the case, then we would see a decrease in recall, e.g. row 12: some names from the source column have a middle name, but this format is not represented in the examples. This suggests that, for all cases exemplified in Table 9 that returned a perfect recall, each format present in the test set is represented at least once in the examples set.

6.4. Scheme comparison

In this section we provide a comparative study of the two schemes for generating examples presented in this paper. We analyse the computational cost of each technique, i.e., Algorithms 1, 2, and 4, the computational cost of transformation synthesis using examples generated by each method, and the consequences (if any) of removing *redundant* examples - Algorithm 4. We do not analyse here the complexity of the other algorithms used in our experiments, e.g. *Match*, *FlashFill*, as this has been done in their original papers.

Table 10 gives details on cases from Table 9 for which examples have been generated using both schemes. The other cases from Table 9 have only been used with the weighted technique because the conditions for the FD-based approach were not met, e.g. FDs could not be discovered. We report the row identifier from Table 9 on the first column, while the rest of the columns represent the source/target cardinality, the number of example pairs returned by Algorithm 1, the number of example pairs returned by Algorithm 2 and Algorithm 4, and the time in seconds required to generate the examples by each algorithm. Note that we only analyse cases for which the validation stage has been passed (for FD-based scheme) and for which the selection stage

#	Domain	Type	Source	Target	Alg.2	Alg.4	Prec	Rec
1	Food Hyg.	Date	2015-12-01	01/12/2015	57	1	1.00	1.00
2	Lobbyists	Pers. Name	Mr. Neil G Bluhm	Bluhm Neil G	206	25	0.75	0.71
3	Docs./Addr.	Address	55 Swain Street, Watchet	Swain Street	2819	N/A	N/A	N/A
4	Build. Perm.	Address	1885 Maud Ave	Maud Ave,Nr. 1885	1051	14	0.90	1.00
5	Build. Perm.	Cost	6048	$6048.00	257	3	0.99	1.00
6	US Congress	Pers. Name	Pete Stark (D-Calif)	Stark	473	34	0.77	1.00
7	US Congress	Pers. Name	Pete Stark (D-Calif)	Pete	279	25	0.75	1.00
8	US Congress	Pers. Name	Pete Stark (D-Calif)	Calif	23	6	0.94	1.00
9	Restaurants	Address	41 Page Avenue, Delhi	Page Avenue	1401	N/A	N/A	N/A
10	Restaurants	Date	2014-06-23 00:00:00+00	06/23/2014	1148	1	1.00	1.00
11	Movies	Actor Name	James Brown	james_brown	1872	N/A	N/A	N/A
12	Movies	Pers. Name	Perry Lang	perry_lang	1680	17	0.79	0.98
13	Employment	Pers. Name	OUTTEN, MIA G	Mia G Outtent	8903	22	0.97	0.99
14	Addresses	Address	2440 N Cannon Drive	Cannon Dr	21	4	0.87	1.00

TABLE 9: Evaluation results

#	Source/Target Size	Alg.1	Alg.2	Alg.4	Alg.1 (s)	Alg.2 (s)	Alg.4 (s)
1	12323/61	57	57	1	0.006	0.2	2.3
2	7542/210	206	206	25	0.005	0.15	42.2
4	10000/1245	26	1051	14	0.006	12.5	8.9
5	10000/1245	22	257	3	0.005	0.15	2.9

TABLE 10: Scheme comparison

(weighted scheme) returned at least one example pair.

The complexity of Algorithm 1 is given by the number of matching relationships of the source and target and by the number of functional dependency candidates for each dataset (lines 3 and 5-6). In practice, the numbers of matches and FDs tend to be small, meaning that the time required to run Algorithm 1 is dominated by the *select* query at lines 8-10. Conversely, the complexity of Algorithm 2 is defined by the number of records of the two datasets, i.e., the more instance values two columns have, the more entries the index will contain, which translates into more pairwise string similarity comparisons. The first two rows of Table 10 show that if we don't use functional dependencies and resort to fuzzy string matching, generating the same number of examples takes 30 times longer. Often (last two rows), the pairs generated by Algorithm 2 contain false positives, i.e. example instances for which the two strings don't represent the same thing. To mitigate such cases we employ Algorithm 4, the complexity of which is given by FlashFill - exponential in the number of examples and highly polynomial in the length of examples [9]. The number of seconds required to refine the examples generated by Algorithm 2 using Algorithm 4 is reported in the last column of Table 10.

The benefit of minimizing examples can be observed in Figure 10, where the synthesis time, i.e. the time it takes FlashFill to synthesise a transformation program, is reported. We compare synthesizing transformations from the examples generated by Algorithm 1 against synthesizing transformations from examples generated by Algorithm 2 and refined by Algorithm 4. It can be observed that eliminating *redundant* example instances substantially improves synthesis time, especially when there are many examples per format: the fact that for row 2 Algorithm 4 reduced 206 examples to 25 suggests that there are almost 9 examples per format.

With respect to the transformations synthesised, for cases 1 and 2 from Table 10 the transformation resulting from the examples returned by Algorithm 1 was equivalent to the transformation obtained from the examples returned by Algorithm 4. For the last two cases, the transformation synthesised from the examples returned by Algorithm 4 was more complex[6]. This is unsurprising, as the number of candidate pairs generated by Algorithm 2 contains more formats that the ones from Algorithm 1, and these formats are successfully identified in Algorithm 4. Recall from Section 2 that FlashFill is able to synthesise transformations that cover multiple cases, i.e., formats, through conditional expressions, e.g. **Switch**.

Overall, the experiments from this section, together with the evaluation from Section 4, show that the functional dependency based approach efficiently generates more effective example pairs, as long as FDs can be discovered. This approach can lead to an increase in synthesis complexity due to redundant example instances, but the problem can be solved in practice by combining Algorithm 1 with the example selection technique from Algorithm 4. For cases where

[6] We consider a transformation to be complex if it contains more conditional branches.

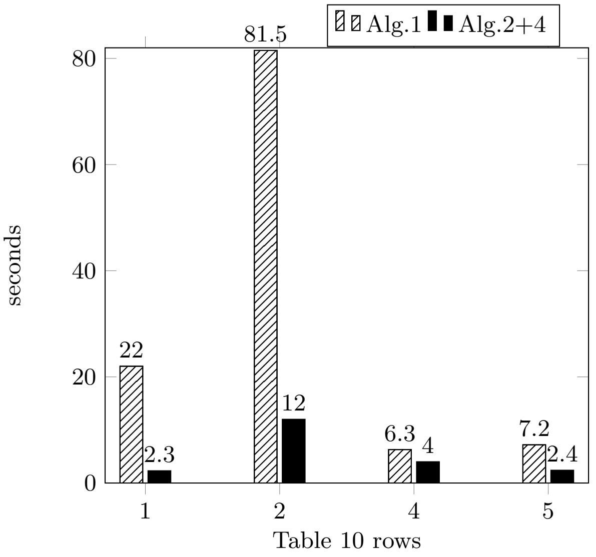

FIGURE 10: Synthesis time (s)

functional dependencies are not available, the weighted solution can prove viable at the expense of effectiveness. For example, row 2 of Table 10 represents a case where the weighted scheme proves less effective than the FD-based scheme: the precision decreased with 23% and the recall decreased with 15% (row 4 of Table 6 compared with row 2 of Table 9).

7. RELATED WORK

In recent years, there has been an increasing number of proposals that use declarative, constraint-based quality rules to detect and repair data problems (e.g. [14, 15, 12, 13], see [10] and [11] for surveys). For many of these heuristic techniques, rule-based corrections are possible as long as the repairing values are present either in an external reference data set or in the original one. For example, in [13] the correct values are searched for in a master data set using editing rules which specify how to resolve violations, with the expense of requiring high user involvement. While the semantics of editing rules can be seen as related to our approach described in Algorithm 1, there are at least two essential differences. First, editing rules address instance-level repairs, i.e. every tuple is checked against every rule and the values of the attributes covered by the rule are replaced with correct ones from the reference data set (if they exist). Our approach determines pairs of values from which we learn transformations that hold for entire columns, so we do not search for the correct version of every value that needs to be cleaned, but for a small number of values that describe the syntactic pattern of the correct version. Second, we determine these transformations automatically, without any intervention from the user.

A related proposal on program synthesis is the work by L. Contreras-Ochando et al. [16]. In their solution, they address the problem of generality characteristic to domain specific languages, such as the one used by FlashFill. Specifically, they propose the use of domain-specific background knowledge to synthesise more specific transformations for fixed data types such as dates, emails, names, etc.

An important body of work close to our proposal for example selection, i.e., Algorithm 4, and built on the transformation synthesis methods, is the research by B. Wu et al. [17]. In their approach, they propose an example recommending algorithm to assist the user in providing enough examples for the entire column to be transformed. To this end, the approach samples a set of records for automatic inspection. It then uses machine learning techniques to identify potentially incorrect records and presents these records for the users to examine. This proposal can be considered orthogonal to our approach. In fact, one can see our technique as an automatic initialization phase of a pay-as-you-go process, while the work by B. Wu et al. [17] could represent the refinement phase where previous results are improved with user support. We leave this idea for future work.

Recent work on transformation-driven join operations by E. Zhu et al. [18] resulted in a technique for automatically joining two tables using fuzzy value pairing and synthesis of transformation programs. Their approach leverages sub-string indexes to efficiently identify candidate row pairs that can potentially join, and then it synthesises a transformation program whose execution can lead to equi-joins. Although their principle is similar to our approach, their focus is on join operations which can be considered as part of the mapping generation phase of a data wrangling pipeline, while our proposal is presented as being part of the format transformation and normalization task of the pipeline. Moreover, their focus on joining operations requires that the columns from which examples for the transformation are being searched, have to be candidate keys in their respective datasets (or one key and one foreign-key). Our approaches, especially the weight based one, aim to enable normalization for any column which has a corresponding match in the target dataset.

Closer to our solution are the tools for pattern enforcement and transformation, starting with traditional ETL tools like Pentaho[7] or Talend[8], but especially Data Wrangler [2], its ancestor Potter's Wheel [19], and OpenRefine[9]. Data Wrangler stands out by proposing a transformation language and an inference algorithm that aids the user in transforming the data. Although the automated suggestion mechanism, described in [20], avoids manual repetitive tasks by actively learning from the decisions the user makes, or from the transformations the user writes, the user must know what transformation is required and how to express that operation. Our work is a first step towards a solution in which

[7] http://www.pentaho.com/

[8] https://www.talend.com/

[9] http://openrefine.org/

such transformations are synthesised without up-front human involvement.

An important body of related work is the research on synthesis programming introduced in Section 2. The algorithms and languages proposed in [3] and [4], and extended in [8] have been developed with spreadsheet scenarios in mind. We build on these solutions and argue that such techniques can be applied on real world big data as well, where the amount of inconsistency and format heterogeneity is higher.

8. CONCLUSIONS

Recent advancements in areas such as big data management have increased the importance of data integration and data wrangling techniques. The challenges caused by data volume and variety require (semi)automatic, cost-effective processes to prepare the data before the analysis process. In this respect, data wrangling is important, as a precursor to data analysis, but is often labour intensive. The creation of data format transformations is an important part of data wrangling and in the data cleaning landscape there are important recent results on techniques to support the creation of such transformations (e.g. [2, 3]). These solutions are user-centric and many of the transformations that can be automatically learned from examples provided by Algorithms 1 and 2 can be obtained using the above solutions as well, but with a much greater level of human involvement. In this paper we build on and complement these results by describing two approaches that can automate the creation of format transformations. Thus we do not replace human curators, but we reduce the number of cases in which manual input is required. In several domains, we have described how candidate sets of examples can be discovered and refined automatically, how these examples can be used to synthesise transformations using FlashFill, and how the resulting transformations can be validated automatically. The evaluation showed that, when certain conditions are met, Algorithm 1 can generate effective examples from which transformation programs can be synthesised. When the conditions are not fulfilled, a heuristic technique was proposed, which proved to have comparable effectiveness while being aligned with synthesis's performance requirements.

The approaches described in this paper can potentially be applied in different settings. For example, transformations could be identified in the background by a platform such as Trifacta's Wrangler[10], and offered as candidate solutions to users. However, the approach can also be used as part of more general extract-transform-load platforms, in which format transformation is only one aspect. Indeed, the methods described here have been incorporated into the VADA system, to support a pay-as-you-go approach to wrangling in which the results from several initially automated steps are refined in the light of data context [22] and user feedback on the data product [21].

ACKNOWLEDGEMENTS

This work has been made possible by funding from the UK Engineering and Physical Sciences Research council, whose support we are pleased to acknowledge.

REFERENCES

[1] Furche, T. and Gottlob, G. and Libkin, L. and Orsi, G. and Paton, N. W. (2016) Data wrangling for big data: Challenges and opportunities. Advances in Database Technology EDBT 2016: Proceedings of the 19th International Conference on Extending Database Technology, Bordeaux, France, March 15-16, pp. 473–478. OpenProceedings.org.

[2] Kandel, S and Paepcke, A. and Hellerstein, J.M. and Heer, J. (2011) Wrangler: Interactive visual specification of data transformation scripts. CHI '11 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Vancouver, BC, Canada, May 07-12, pp. 3363–3372. ACM New York, NY, USA.

[3] Gulwani, S. (2011) Automating string processing in spreadsheets using input-output examples. POPL' 11: Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Austin, Texas, USA, January 26 - 28, pp. 317–330. ACM New York, NY, USA.

[4] Singh, R. (2016) Blinkfill: Semi-supervised programming by example for syntactic string transformations. Proceedings of the VLDB Endowment, Volume 9 Issue 10, June 2016, 816–827.

[5] Bogatu, A. and Paton, N.W. and Fernandes, A.A.A. (2017) Towards automatic data format transformations: Data wrangling at scale. BICOD Proceedings of the 31st British International Conference on Databases, London, United Kingdom, July 10-12, pp. 36–48. Springer International Publishing, Basel.

[6] Papenbrock, T. and Naumann, F. (2016) A hybrid approach to functional dependency discovery. SIGMOD '16 Proceedings of the 2016 International Conference on Management of Data, San Francisco, California, USA, June 26 - July 01, pp. 821–833. ACM New York, NY, USA.

[7] Rahm, E. and Bernstein, P.A. (2001) A survey of approaches to automatic schema matching. The VLDB Journal, Volume 10 Issue 4, December 2001, 334–350.

[8] Wu, B. and Knoblock, C.A. (2015) An iterative approach to synthesize data transformation programs. Proceedings of the 24th International Joint Conference of Artificial Intelligence, Buenos Aires, Argentina, July 25-31, pp. 1726–1732. AAAI Press.

[9] Raza, M. and Gulwani, S. and Milic-Frayling, N. (2014) Programming by example using least general generalizations. AAAI'14 Proceedings of the 28th AAAI Conference on Artificial Intelligence, Quebec, Canada, July 27-31, pp. 283–290. AAAI Press.

[10] Fan, W. (2008) Dependencies revisited for improving data quality. PODS '08 Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium

[10] https://www.trifacta.com/products/wrangler/

on Principles of database systems, Vancouver, Canada, June 09-12, pp. 159–170. ACM New York, NY, USA.

[11] Fan, W. (2015) Data quality: From theory to practice. ACM SIGMOD Record. Volume 44 Issue 3, September 2015, 7–18.

[12] Chu, X. and Ilyas, I.F. and Papotti, P. (2013) Holistic data cleaning: Putting violations into context. ICDE IEEE 29th International Conference on Data Engineering, Brisbane, QLD, Australia, April 8-12, pp. 458–469. IEEE.

[13] Fan, W. and Li, J. and Ma, S. and Tang, N. and Yu, W. (2012) Towards certain fixes with editing rules and master data. The VLDB Journal, Volume 21 Issue 2, April 2012, 213–238.

[14] Fan, W. and Geerts, F. and Jia, X. and Kementsietsidis, A. (2008) Conditional functional dependencies for capturing data inconsistencies. ACM TODS, Volume 33 Issue 2, June 2008, Article No. 6.

[15] Yakout, M. and Elmagarmid, A.K. and Neville, J. and Ouzzani, M. and Ilyas, I.F. (2011) Guided data repair. Proceedings of the VLDB Endowment, Volume 4 Issue 5, February 2011, 279–289.

[16] Contreras-Ochando, L. and Ferri, C. and Hernandez-Orallo, J. and Martinez-Plumed, F. and Ramirez-Quintana, M.J. and Katayama, S. (2017) Domain specific induction for data wrangling automation. AutoML@ICML, Sydney, Australia, Aug 10.

[17] Wu, B. and Knoblock, C.A. Maximizing correctness with minimal user effort to learn data transformations. IUI '16 Proceedings of the 21st International Conference on Intelligent User Interfaces, Sonoma, California, USA, March 07-10, pp. 375–384. ACM New York, NY, USA.

[18] Zhu, E. and He, Y. and Chaudhuri, S. (2017) Auto-join: Joining tables by leveraging transformations. Proceedings of the VLDB Endowment, Volume 10 Issue 10, June 2017, 1034–1045.

[19] Raman, V. and Hellerstein, J.M. (2001) Potter's wheel: An interactive data cleaning system. VLDB '01 Proceedings of the 27th International Conference on Very Large Data Bases, Roma, Italy, September 11-14, pp. 381–390. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

[20] Heer, J. and Hellerstein, J.M. and Kandel, S. (2015) Predictive interaction for data transformation. CIDR 7th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7.

[21] Konstantinou, N. et al. (2017) The VADA architecture for cost-effective data wrangling. SIGMOD '17 Proceedings of the 2017 ACM International Conference on Management of Data, Chicago, Illinois, USA, May 14-19, pp. 1599–1602. ACM New York, NY, USA.

[22] Koehler, M. et al. (2017) Data context informed data wrangling. 2017 IEEE International Conference on Big Data, Boston, MA, USA, December 11-14, pp. 956–963.