Unifacta: Profiling-driven String Pattern Standardization

Zhongjun Jin* Michael Cafarella* H. V. Jagadish* Sean Kandel† Michael Minar†
*University of Michigan †Trifacta Inc.
{markjin,michjc,jag}@umich.edu {skandel,mminar}@trifacta.com

ABSTRACT

Data cleaning is critical for effective data analytics on many real-world data collected. One of the most challenging data cleaning tasks is *pattern standardization*–reformatting ad hoc data, e.g., phone numbers, human names and addresses, in heterogeneous non-standard patterns (formats) into a standard pattern–as it is tedious and effort-consuming, especially for large data sets with diverse patterns. In this paper, we develop UNIFACTA, a technique that helps the end user standardize ill-formatted ad hoc data. With minimum user input, our proposed technique can effectively and efficiently help the end user synthesize high quality explainable pattern standardization programs. We implemented UNIFACTA, on TRIFACTA, and experimentally compared UNIFACTA with a previous state-of-the-art string transformation tool, FLASHFILL, along with TRIFACTA and BLINKFILL. Experimental results show that UNIFACTA produced programs of comparable quality, but more explainable, while requiring substantially less user effort than FLASHFILL, and other related baseline systems. In a user effort study, UNIFACTA saved 30% - 70% user effort compared to the baseline systems. In an experiment testing the user's understanding of the synthesized transformation logic, UNIFACTA users achieved a success rate about twice that of FLASHFILL users.

PVLDB Reference Format:
Zhongjun Jin, Michael Cafarella, H. V. Jagadish, Sean Kandel, and Michael Minar. Unifacta: Profiling-driven String Pattern Standardization. *PVLDB*, 11 (5): xxxx-yyyy, 2018.
DOI: https://doi.org/TBD

1. INTRODUCTION

Much of the raw data we collect today are like "tar sands": everyone believes that they are potentially valuable, but it takes enormous effort to clean them for downstream use, such as data analysis or visualization. A common problem in raw data is non-standard representation. For example, a set of medical billing codes may look messy like column "Raw Data" in Table 1. Any data integration or analytics on such *ad hoc data* can be difficult unless they are reformatted into a standard, unified format, as in column "Standardized Data" in Table 1.

Raw data	Standardized data
CPT-00350	[CPT-00350]
[CPT-00340	[CPT-00340]
[CPT-11536]	[CPT-11536]
CPT115	[CPT-115]

Table 1: Standardizing messy medical billing codes

Data scientists struggle to convert such diverse data representations into a standard pattern. Traditionally, they write ad hoc scripts to perform this data standardization. These scripts are laborious to specify correctly, and hard to understand or maintain.

This problem has drawn interest from both academia and industry. Previous attempts [23, 19] have been made to automate the process of string transformation program generation through a Programming By Demonstration (PBD) interaction paradigm: the end user demonstrates the desired transformation by either specifying the correct operations or by giving hints of the correct operations. While these tools have shown some success in layout transformation tasks (e.g., TRIFACTA), their value for text manipulation tasks is less clear. In addition, they are often difficult for some non-experts to use [15, 12, 22, 11].

Researchers [14, 32] have adopted another approach, Programming By Example (PBE), to automate the string transformation process: the end user provides explicit input-output example pairs to describe the desired transformation, and the system outputs a domain-specific language (DSL) program satisfying all provided examples. Although no domain knowledge about the string transformation is required, the end users of PBE systems, based on our observation, may still face two usability challenges while performing PBE string transformation:

- *Specification*: To provide meaningful input-output example pairs, the user has to identify data records that are ill-formatted, which can be painstaking, especially when the raw data is large or very messy. Moreover, PBE systems usually depend on entirely error-free examples for correct operation, whereas the user is likely to make mistakes (e.g., spelling errors, typos) while creating example pairs.
- *Verification*: As the generated transformation program is usually not explained, the user has low confidence in its correctness. The only way to verify the trans-

Proceedings of the VLDB Endowment, Vol. 11, No. 5

DOI: https://doi.org/TBD

formation is by checking if each string has been correctly transformed, which can quickly overwhelm the user if there is a large volume of data. Furthermore, if the system is not expressive enough to yield a desired transformation, the user won't realize it until after performing many rounds of example-providing and checking.

Our Approach — In this paper, we propose our solutions to address the above two challenges: **1)** automatic *pattern profiling* and *transformation program synthesis* for the *Specification* challenge, **2)** *program explanation* for the *Verification* challenge.

To address the *Specification* challenge in the PBE systems, we propose a new interaction model that allows the user to describe the desired transformation to the system at pattern level, not string level: the user indicates her desired transformation through specifying an easy-to-understand standard data representation–the *string pattern*–and our technique will synthesize a transformation program that converts all ill-formatted data into the desired pattern.

The intuition behind this proposed interaction paradigm is that instead of explicit input-output pairs, which is cumbersome and error-prone to provide, we can directly ask for the *target pattern*, i.e., the desired pattern of the target data. If the target pattern can be identified in the input data, the user can simply point it out[1]. For example, if the user were to tell us that they like the format of the first row in Figure 1, that could be a sufficient hint for the system to transform the remaining rows of the table into the form in Figure 2.

Actually implementing this intuition requires us to address two major technical challenges. ***First, we need to help the user quickly identify the target pattern from the raw data.*** Letting the user select strings in the target pattern from the input data is not acceptable, since the input data may be large making these strings hard to find when they are few in number. Instead, we discover all distinct patterns from the input data and let the user choose among them, which is easier. The pattern information presented must not be too sophisticated. Plus, the generation of the patterns should also be "hassle-free" for the user. We propose an automatic data profiling technique that discovers pattern descriptions of the underlying raw data meant for non-experts to understand.

Second, we need to synthesize transformation programs from each of the observed patterns (patterns automatically discovered) to the desired pattern. We propose a string transformation language, UniFi, to represent such transformations (Section 4); it is expressive enough to handle many transformation tasks in our representative workload. Given a set of source patterns (observed patterns that are not the target pattern) and the target pattern, if we can correctly map correspondences between the various components in the source and target patterns [2], the system could, in theory, develop the transformation program without additional help from the user. We propose a sound and complete algorithm to efficiently identify the component correspondences between patterns and synthesize the UniFi program in Section 5, using the pattern descriptions discovered previously, and apply a graph data structure and search-based algorithm to represent and infer the satisfying programs.

[1]If no input data matches the target pattern, the user could alternatively choose to manually specify the target pattern.
[2]E.g., if we use the format of the first string in Figure 1 as the target pattern, we need to figure out how to obtain "(", area code, etc., from the source pattern

(734) 645-8397
(734)586-7252
734-422-8073
734.236.3466
...

Figure 1: Phone numbers with diverse formats

(734) 645-8397
(734) 586-7252
(734) 422-8073
(734) 236-3466
...

Figure 2: Phone numbers in a standard format

The second challenge–*Verification*–is addressed by explaining synthesized transformation logic. The synthesized program is expressed as a set of Replace operations parameterized by natural language regular expressions to the end user (Section 5.4), which makes the program easy to understand. We also explain each of these *Replace* operations using a visualization–*Preview Table*. If the initially inferred transformation is incorrect, the user can quickly identify the error and adjust the transformation programs easily.

In addition, the earlier step of pattern generation also helps simplify the *Verification* process. Once the synthesized transformation code is executed on the input data, the system will generate new pattern representations for the transformed data. The user can verify the transformation through checking the new pattern information, which is less work than verifying each individual string value.

Through the above means, we are able to greatly ameliorate the *Specification* and *Verification* usability challenges in the traditional PBE string transformation systems. Compared to FlashFill, our proposed technique does require a bit more expertise, but our user study shows that a non-expert user with a basic understanding of regular expressions can spend no more user effort and achieve a more explainable transformation program using our technique.

Organization — After motivating our problem with an example in Section 2, we discuss the following contributions:

- We define the pattern standardization problem and present our novel interaction model driven by string pattern profiling. (Section 3)
- We present our methodology for pattern profiling. (Section 4)
- We develop algorithms for pattern standardization, which can transform any given input pattern to the desired standard pattern. (Section 5)
- We experimentally evaluate the Unifacta prototype and other baseline systems using a benchmark suite we collected. (Section 6)

We explore Related Work in Section 7 and finish with a discussion of future work in Section 8.

2. MOTIVATING EXAMPLE

Bob is a technical support at the customer service department. He had a set of 10,000 phone numbers in various formats (as in Figure 1) and wanted to have these numbers in a unified format. Given the volume and the heterogeneity of the data, neither manual fixing them or hard-coding a transformation script was convenient for Bob. He decided to see if there was any automated solution to this problem.

\({digit}3\){digit}3\-{digit}4 **(734)586-7252** ... *(2572 rows)*
{digit}3\-{digit}3\-{digit}4 **734-422-8073** ... *(3749 rows)*
\({digit}3\)\ {digit}3\-{digit}4 **(734) 645-8397** ... *(1436 rows)*
{digit}3\.{digit}3\.{digit}4 **734.236.3466** ... *(631 rows)*
...

Figure 3: Raw data patterns before standardization

```
1 Replace '/^\(({digit}{3})\)({digit}{3})\-({digit}{4})$/'
      in column1 with '($1) $2-$3'
2 Replace '/^({digit}{3})\-({digit}{3})\-({digit}{4})$/'
     in column1 with '($1) $2-$3'
3 ...
```

Figure 4: Suggested pattern standardization operations in UNIFACTA

\({digit}3\)\ {digit}3\-{digit}4 **(734) 645-8397** ... *(10000 rows)*

Figure 5: Patterns after standardization

Bob found that Excel 2013 had a new feature named FLASHFILL that could perform pattern standardization. He loaded the new data set into Excel and performed FLASHFILL on them.

EXAMPLE 1. *Initially, Bob found using* FLASHFILL *was straightforward: he simply needed to provide an example of the transformed form of each data entry in the input data that is ill-formatted and copy the exact value if the data entry is already in the correct format. However, in practice, it was not so easy. First, Bob needed to carefully check each phone number entry before deciding whether it is ill-formatted or not, which is not easy for large volume heterogeneous data. Second, he must be very careful not making any mistake when typing a new example. Otherwise,* FLASHFILL *would simply fail (*Specification Challenge*). After obtaining a new input-output example pair,* FLASHFILL *would update the transformation results for the rest of the input data, and Bob had to carefully examine if any of the transformation result was incorrect, which was also laborious (*Verification Challenge*). Another issue is that the customer phone numbers were critical information for Bob's company and must not be damaged during the transformation. Nevertheless, Bob got little insights from* FLASHFILL *regarding the transformation program generated and hence was not sure if the transformation was reliable (*Verification Challenge*).*

Bob heard about UNIFACTA and decided to give it a try.

EXAMPLE 2. *He loaded his data into* UNIFACTA *and* ***it immediately presented a list of the string patterns for phone numbers in the input data (Figure 3)****, which helped Bob quickly tell which part of the data are ill-format or not. To describe the desired transformation, Bob did not need to specifically type the example values but* ***simply selected the third pattern as the desired pattern****. So far, Bob did not have any* Specification *issue he had when using* FLASHFILL. UNIFACTA *then inferred a program transforming all ill-formatted strings to this pattern.* ***The program is presented as a set of*** **Replace** ***operations for each raw pattern in Figure 3, each with a small picture visualizing the transformation effect****. Bob was not a regular expressions guru, but these operations seemed simple to understand and verify. He was positive that the inferred transformation was correct.* ***After executing this script,*** UNIFACTA ***generated another pattern information*** *which showed all data followed the selected pattern (Figure 5). This further convinced Bob that the program was correct.* UNIFACTA *made* Verification *simpler as well.*

Notation	Description
$\mathcal{S} = \{s_1, s_2, \dots\}$	A set of ad hoc strings $s_1, s_2, \dots$ needed to be standardized.
$\mathcal{P} = \{p_1, p_2, \dots\}$	String patterns derived from $\mathcal{S}$.
$p_i = \{t_1, t_2, \dots\}$	Pattern made from a sequence of tokens t_i
$\mathcal{T} \in \mathcal{P}$	The target pattern chosen by the user indicating that all strings in $\mathcal{S}$ needed to be transformed into this format.
$\mathcal{L}$	The program synthesized in UNIFACTA pattern standardization language UNIFI that can transform the formats of $\mathcal{S}$ into $\mathcal{T}$.
$\mathcal{E}$	The expression $\mathcal{E}$ in $\mathcal{L}$, which is a concatenation of Extract and/or ConstStr operations. It is a transformation plan for a source pattern. We also refer to it as an *Atomic Transformation Plan* in the paper.
$\mathcal{Q}(\widetilde{\mathfrak{t}}, p)$	Token quantity of token $\widetilde{\mathfrak{t}}$ in pattern p
$\mathcal{G}$	Potential expressions represented in Directed Acyclic Graph.

Table 2: Frequently used notation

3. OVERVIEW

3.1 Problem Definition

Using notations summarized in Table 2, we formally define the pattern standardization problem:

DEFINITION 3.1 (PATTERN STANDARDIZATION PROBLEM). *Given a set of strings $\mathcal{S} = \{s_1, \dots, s_n\}$, generate a program $\mathcal{L}$ that transforms each string in $\mathcal{S}$ to an equivalent string matching the user-specified target pattern $\mathcal{T}$.*

In practice, some strings may already be in the desired target format. The user can just point to one or more of these strings to specify the desired target format. This may be easier than the user specifying the format from scratch.

However, manually finding and labeling data records that are already in standard format in the raw data can still be burdensome, especially when the standard-formatted data records are scarce. What the user needs is a more efficient way to group and label those standard-formatted data. We accomplish this through the use of *string patterns*, which are usually much smaller in size than data records.

We ask the end user to label the patterns rather than the actual data records. Thus, our solution to the pattern standardization problem–UNIFACTA–is decomposed into two system steps, with an intervening user step: 1) Discover a pattern information representation for the given set of strings. 2) Have the user select a target pattern $\mathcal{T} \in \mathcal{P}$. 3) Synthesize a UNIFI program that transforms each string from its original pattern into a string that follows pattern $\mathcal{T}$.

Note that if the target format does not exist in the raw data, our proposed pattern standardization solution will still be usable. In such a case, the user could manually specify the target pattern $\mathcal{T}$, instead of selecting it from the discovered patterns.

Token Name	Regular Expression	Example
digit	`[0-9]`	"12"
lower	`[a-z]`	"car"
upper	`[A-Z]`	"IBM"
alpha	`[a-zA-Z]`	"Excel"
alpha-numeric	`[a-zA-Z0-9]`	"Excel2013"

Table 3: Token names and their descriptions in UNIFACTA

In the following two subsections, we define the string pattern language and the transformation language we use in this paper. The main ideas in this paper can be carried through with different choices for these languages. However, these choices do matter for the performance evaluation, and are described here for concreteness.

3.2 String Patterns

A string pattern is a representation that describes the structure of the string representation of the attribute value. A natural way to describe a pattern could be a regular expression over the characters that constitute the string. For our reformatting task, we find that groups of contiguous characters are often transformed together as a group. Further, these groups of characters are meaningful in themselves. For example, in a date string "11/02/2017", it is useful to cluster "2017" into a single group, because these four digits are likely to be manipulated together. We call such meaningful groups of characters as *tokens*.

Table 3 presents all tokens we currently support in UNIFACTA, including their token names, regular expressions. In addition, we also support tokens of constant values (e.g., ",", ":"). In the rest of the paper, we represent and handle these tokens of constant values differently from the 5 tokens defined in Table 3. For convenience of presentation, we denote such tokens with constant values as ***literal tokens*** and 5 tokens defined in Table 3 as ***base tokens***.

A pattern is written as a sequence of tokens, each followed by a quantifier indicating the number of occurrences of the preceding token. In UNIFACTA, a quantifier is either a single natural number or "+", indicating that the token appears at least once. In the rest of the paper, to be succinct, a token will be denoted as "$\{\widetilde{\mathtt{t}}\}\mathtt{q}$" if $\mathtt{q}$ is a number (e.g., {digit}3) or "$\{\widetilde{\mathtt{t}}\}+$" otherwise (e.g., {digit}+). If $\widetilde{\mathtt{t}}$ is a literal token, it will be surrounded by a single quotation mark, like ':'.

3.3 Pattern Standardization Programs

In UNIFACTA, we propose the pattern standardization language UNIFI shown in Figure 6.

As with FLASHFILL [14] and BLINKFILL [32], we only focus on syntactic transformation, where strings are manipulated as a sequence of characters and no external knowledge is accessible, in this paper. Allowing semantic transformation (e.g., converting "IBM" to "International Business Machines") will be our future work. Further–again like BLINKFILL [32]–our proposed pattern standardization language UNIFI does not support loops.

The top-level of program UNIFI is a Switch statement that conditionally maps strings to a transformation. Match checks whether a string s is an *exact match* of a certain pattern p we discover previously. Once a string matches this pattern, it will be processed by an ***Atomic Transformation Plan*** (denoted by $\mathcal{E}$) defined below.

$$\begin{aligned}
\text{Program } \mathcal{L} &:= \mathsf{Switch}((b_1, \mathcal{E}_1), \ldots, (b_n, \mathcal{E}_n)) \\
\text{Predicate } b &:= \mathsf{Match}(s, p) \\
\text{Expression } \mathcal{E} &:= \mathsf{Concat}(f_1, \ldots, f_n) \\
\text{String Expression } f &:= \mathsf{ConstStr}(\widetilde{s}) \mid \mathsf{Extract}(\widetilde{\mathtt{t}}_i, \widetilde{\mathtt{t}}_j) \\
\text{Token Expression } \mathtt{t}_i &:= (\widetilde{\mathtt{t}}, \mathtt{r}, \mathtt{q}, i)
\end{aligned}$$

Figure 6: Pattern Standardization Language UNIFI

DEFINITION 3.2 (ATOMIC TRANSFORMATION PLAN). *An Atomic Transformation Plan is a transformation strategy, composed of a sequence of parametrized string operators, for a given source pattern.*

The available string operators include ConstStr, Extract. $\mathsf{ConstStr}(\widetilde{s})$ denotes a constant string $\widetilde{s}$. $\mathsf{Extract}(\widetilde{\mathtt{t}}_i,\widetilde{\mathtt{t}}_j)$ extracts from the i^{th} token to the j^{th} token in a pattern. We denote an Extract operations as $\mathsf{Extract}(i,j)$, or $\mathsf{Extract}(i)$ if $i = j$, in the rest of the paper. A token $\mathtt{t}$ is represented as $(\widetilde{\mathtt{t}}, \mathtt{r}, \mathtt{q}, i)$: $\widetilde{\mathtt{t}}$ is the token name in Table 3; $\mathtt{r}$ represents the corresponding regular expression of this token; $\mathtt{q}$ is the quantifier of the token expression; i denotes the index (one-based) of this token in the source pattern.

We use the following two examples used by FLASHFILL [14] and BLINKFILL [32] to briefly demonstrate the expressive power of UNIFI, and the more detailed expressive power of UNIFI would be examined in the experiments in Section 6.2. For simplicity, $\mathsf{Match}(s, p)$ is shortened as $\mathsf{Match}(p)$ as the input string s is fixed for a given task.

EXAMPLE 3. *The goal of this problem modified from test case "**Example 3**" in* BLINKFILL *is to transform all messy in the medical billing codes into the correct form "[CPT-XXXX]" as in Table 1.*

The UNIFI *program for this standardization task is*

```
Switch((Match("\[{upper}+\-{digit}+"),
     (Concat(Extract(1,4), ConstStr(']'))))),
   (Match("{upper}+\-{digit}+"),
     (Concat(ConstStr('['), Extract(1,3),
       ConstStr(']'))))
   (Match("{upper}+{digit}+"),
     (Concat(ConstStr('['), Extract(1),
       ConstStr('-'), Extract(2) ConstStr(']')))))
```

EXAMPLE 4. *This problem is borrowed from "**Example 9**" in* FLASHFILL. *The goal is to transform all names into a unified format as in Table 4.*

Raw data	**Standardized data**
Dr. Eran Yahav	Yahav, E.
Fisher, K.	Fisher, K.
Bill Gates, Sr.	Gates, B.
Oege de Moor	Moor, O.

Table 4: Standardizing messy employee names

A UNIFI *program for this task is*

```
Switch((Match("{upper}{lower}+\.\ {upper}{lower}+\
              {upper}{lower}+"),
    Concat(Extract(8,9),ConstStr(','), ConstStr(' '),
           Extract(5))),
   (Match("{upper}{lower}+\ {upper}{lower}+\,\
                 {upper}{lower}+\."),
    Concat(Extract(4,5), ConstStr(','), ConstStr(' '),
```

```
        Extract(1)),
(Match("{upper}{lower}+\ {lower}+\ {upper}{lower}+"),
 Concat(Extract(6,7), ConstStr(','), ConstStr(' '),
        Extract(1))))
```

While being expressive enough to cover many pattern standardization tasks, our designed pattern standardization language UniFi can be easily explained to the end user. We discuss how we explain the language in Section 5.4.

4. AUTOMATIC PATTERN PROFILING

In Unifacta, we need to discover the pattern structure information of the ad hoc data without user intervention. LearnPADS [9, 8] is an important project relevant to our interest here. Yet, a significant impediment to applying its approach to pattern discovery in our project is that they are usually good at handling system-generated data following a single high-level pattern (e.g., log files), whereas heterogeneous data (e.g., human names, phone numbers, dates) in our project may be dominated by multiple high-level patterns. Plus, the PADS language [7] itself is known to be hard for a non-expert to read [38]. Our interest is to derive simple patterns of the form described above in Sec. 3.

In this section we propose a completely automated means to discover patterns given a set of strings. We design a two-phase approach for pattern profiling: 1) tokenization: tokenize the given set of strings of ad hoc data, 2) refinement: construct a ***pattern tree*** that allows the end user to view/understand the pattern structure information in a simpler and more systematic way and Unifacta generates a simple transformation program.

4.1 Tokenization

Tokenization is a common process in string processing when string data needs to be manipulated in chunks larger than single characters. A simple parser can do the job.

Below are the rules we follow in the tokenization phase.

- Non-alphanumeric characters carry important hints of the string structure. Each such character is identified as an individual literal token.
- We always choose the most precise base type to describe a token. For example, a token with string content "cat" can be categorized as "lower", "alphabet" or "alphanumeric" tokens. We choose "lower" as the token type for this token.
- The quantifiers are always natural numbers.

Here is an example of the token description of a string data record discovered in tokenization phase.

Example 5. *Suppose the string "Bob123@gmail.com" is to be tokenized. The result of tokenization becomes [{upper}, {lower}2, {digit}3, '@', {lower}5, '.', {lower}3].*

4.2 Refinement

The initial set of patterns discovered in the tokenization phase are usually too specific. Since we distinguish different patterns by token names, token positions and quantifiers, the actual number of patterns discovered in the ad hoc data in tokenization phase could be huge. It can be unacceptably expensive to develop pattern standardization programs separately for each pattern. Worse, user effort is proportional to the number of patterns. Furthermore, user comprehension is inversely related to the number of patterns.

Algorithm 1: Refine Pattern Representations

Data: Pattern set $\mathcal{P}$, generalization strategy $\tilde{g}$
Result: Set of more generic patterns $\mathcal{P}_{final}$

1 $\mathcal{P}_{final}$,$\mathcal{P}_{raw} \leftarrow \emptyset$;
2 $\mathcal{C}_{raw} \leftarrow \{\}$;
3 **for** $p_i \in \mathcal{P}$ **do**
4 $\quad p_{parent} \leftarrow getParent(p_i, \tilde{g})$;
5 $\quad$ add p_{parent} to $\mathcal{P}_{raw}$;
6 $\quad \mathcal{C}_{raw}\ [p_{parent}\ ] = \mathcal{C}_{raw}\ [p_{parent}\ ] + 1$;
7 **for** $p_{parent} \in \mathcal{P}_{raw}$ *ranked by* $\mathcal{C}_{raw}$ *from high to low* **do**
8 $\quad p_{parent}.child \leftarrow \{p_j | \forall p_j \in \mathcal{P}, p_j.isChild(p_{parent})\}$;
9 $\quad$ add p_{parent} to $\mathcal{P}_{final}$;
10 $\quad$ remove $p_{parent}.child$ from $\mathcal{P}$;
11 **Return** $\mathcal{P}_{final}$;

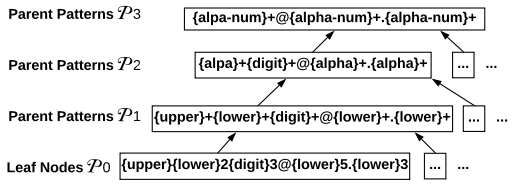


Figure 7: Pattern tree example

Therefore, we build a hierarchical pattern representation, ***pattern tree***, with the leaf nodes being the patterns discovered through tokenization, and every internal node being a parent pattern, i.e., a generalization of the child patterns. With this hierarchical pattern description, the user can understand the pattern information at a high level without being overwhelmed by many details, and the system can generate simpler programs.

Pattern Tree Construction — We successively obtain more general patterns and construct each layer in the pattern tree bottom-up using Algorithm 1, with different generalization strategy $\tilde{g}$ each time, and the child pattern set $\mathcal{P}$.

Specifically, in the Unifacta implementation, we perform three rounds of refinement to construct *parent patterns* (more generic patterns) in the new layer using $getParent$ in Algorithm 1, each with a particular generalization strategy:

1. natural number quantifier to '+'
2. {lower}, {upper} tokens to {alpha}
3. {alpha}, {digit} tokens to {alpha-numeric}

Line 3-5 synthesizes all the parent patterns of the current set of patterns using the generalization strategy $\tilde{g}$.

For each child pattern, we obtain a parent pattern. Some of these parent patterns might have overlapping expressive power, or simply identical to others. Keeping all these parent patterns in the same layer of the pattern tree is unnecessary and increases the complexity of the pattern tree generated. Therefore, we only keep a small subset of the parent patterns initially discovered and make sure they together can cover any child pattern in $\mathcal{P}$.

To do so, we use a counter $\mathcal{C}_{raw}$ counting the frequencies of the obtained parent patterns (line 6). Then, we iteratively add the parent pattern that covers the most patterns in $\mathcal{P}$ into the set of more generic patterns to be returned (line 7-10). The returned set covers all patterns in $\mathcal{P}$ (line 11).

Algorithm 2: Synthesize UniFi Program

Data: String pattern tree root $\mathcal{P}_R$, target pattern $\mathcal{T}$
Result: Synthesized program $\mathcal{L}$

1 $Q_{unsolved}$, $\mathcal{Q}_{unsolved} \leftarrow [\,]$;
2 $\mathcal{L} \leftarrow \emptyset$;
3 $Q_{unsolved}$.push($\mathcal{P}_R$);
4 **while** $Q_{unsolved} \neq \emptyset$ **do**
5 $p \leftarrow Q_{unsolved}$.pop();
6 **if** *ifValidSource(p, $\mathcal{T}$)* **then**
7 $\mathcal{G} \leftarrow findTokenAlignment(p, \mathcal{T})$;
8 $\mathcal{Q}_{unsolved}$.push($\{p, \mathcal{G}\}$);
9 **else**
10 $Q_{unsolved}$.push(p.children);
11 $\mathcal{L} \leftarrow createProgs(\mathcal{Q}_{unsolved})$;
12 **Return** $\mathcal{L}$

Example 6. *Given the pattern we obtained in Example 5, we successively apply Algorithm 1 with Strategy 1, 2 and 3 to generalize parent patterns $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ and construct the pattern tree as in Figure 7.*

Find Constant Tokens — Some of the tokens in the discovered patterns have constant values. Discovering such constant values and representing them using the actual values rather than base tokens helps improve the quality of the program synthesized. For example, if most entities in a faculty name list contain "Dr.", it is better to represent a pattern as ['Dr.','\ ', '{upper}', '{lower}+'] than ['{upper}','{lower}','.','\ ', '{upper}', '{lower}+']

Similar to [9], we take a heuristic approach to find tokens with constant values using the statistics over tokenized strings in the data set. In the future, we plan to utilize external knowledge to make this process more reliable.

5. UNIFI PROGRAM SYNTHESIS

After obtaining the patterns through pattern profiling, we discuss how to find a UniFi program using the pattern tree. Algorithm 2 presents an overview of our synthesis technique.

Although Section 4 gives us a tree of patterns, it might not be wise to create an *atomic transformation plan* (Definition 3.2) for every pattern in the pattern tree. Some patterns are too general, it can be hard to determine how to transform these patterns into the desired target pattern. For instance, if a pattern is "{alpha-numeric}+,{alpha-numeric}+", it would be hard to tell if or how it could be transformed into the desired pattern of "{upper}{lower}+:{digit}+". By comparison, a child pattern "{upper}{lower}+,{digit}+" seems to be a better fit as the *source pattern*. One the other hand, if a pattern is a good fit as a *source pattern*, it is unnecessary to make atomic transformation plans for its child patterns. Another important reason that a pattern might not be suitable as a source pattern is that it could be a description of garbage values in an ad hoc data set. For example, a data set of phone numbers may contain "N/A" as a data record because the customer does not want to reveal this information. Such data records cannot possibly be transformed into the specified target phone number format. In this case, creating an atomic transformation plan is pointless.

Motivated by the above reasons, we traverse the pattern tree top-down to find valid *candidate source patterns*, i.e., suitable source patterns that could possibly be transformed into the target pattern (line 6, see Section 5.1). Once a source candidate is identified, we discover all *token correspondences* between this source pattern in Q_{solved} and the target pattern (line 7, see Section 5.2). With the generated *token correspondence* information, we synthesize the pattern standardization program including an atomic transformation plan for every source pattern (line 11, see Section 5.3).

5.1 Identify Source Candidates

The input data set is not guaranteed to be of high quality; it can be quite noisy especially when the data is not system-generated. Therefore, prior to synthesizing a program for a source pattern, we need to check whether this source pattern is a *candidate source pattern*, that is, it is possible to find a transformation from it into the target pattern. Any input data matching no candidate source pattern is left unchanged and flagged for manual review.

In UniFacta, we assume we have no access to any external knowledge. This implies that we should not introduce any *semantic transformation* (e.g., transforming "September" to "9") during the course of transformation, in other words, all *semantic data* (usually alpha-numeric values represented by *base tokens*) in the target pattern (usually represented by *base tokens*) must be extracted from the source pattern. Thus, *candidate source pattern is defined as*:

Definition 5.1 (Candidate Source Pattern). *Given the target pattern $\mathcal{T}$, a source pattern is a candidate source pattern if all semantic data in the target pattern can be extracted from this source pattern.*

To determine if a source pattern qualifies as a candidate source pattern, we check if all semantic data in the target pattern can possibly be extracted from this source pattern. We examine if there are sufficient base tokens of each kind in the source pattern matching the base tokens in the target tokens. To do so, we create a *token histogram* for every kind of token for each pattern, with x axis representing the quantifier $\mathtt{q}$ of different kinds of tokens $\{\widetilde{\mathtt{t}}\}$ and y axis representing number of such token $\{\widetilde{\mathtt{t}}\}\mathtt{q}$. Figure 8 shows such a token histogram for pattern ['(', {digit}3, ')', {digit}3, '-', {lower}4].

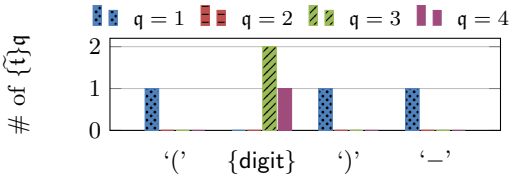


Figure 8: Token histogram for '({digit}3){digit}3-{digit}4'

With this histogram, we compare the *token quantity* for every kind of base token for each pair of source and target pattern. Given a pattern p and a token $\{\widetilde{\mathtt{t}}\}$, the *token quantity* of this token $\widetilde{t}$ in p is defined as

$$\mathcal{Q}(\widetilde{\mathtt{t}}, p) = \sum_{i=1}^{n} \{t_i.\mathtt{q} | t.name = \widetilde{\mathtt{t}}\}, p = \{t_1, \ldots, t_n\} \quad (1)$$

If a quantifier is not a natural number but "+", we treat it as 1 in computing $\mathcal{Q}$.

Algorithm 3: Token Alignment Algorithm

Data: Target pattern $\mathcal{T} = \{t_1, \ldots, t_m\}$, candidate source pattern $\mathcal{P}_{cand} = \{t'_1, \ldots, t'_n\}$, where t_i and t'_i denote base tokens
Result: Directed acyclic graph $\mathcal{G}$

1 $\widetilde{\eta} \leftarrow \{0, \ldots, n\}$;
2 $\eta^s \leftarrow 0$;
3 $\eta^t \leftarrow n$;
4 $\xi \leftarrow \{\}$;
5 **for** $i \in \{1, \ldots, n\}$ **do**
6 **for** $t'_j \in \mathcal{P}_{cand}$ **do**
7 **if** *ifSyntacticallySimilar*(t_i, t'_j) **then**
8 $e \leftarrow \mathsf{Extract}(t'_j)$;
9 add e to $\xi_{(i-1,i)}$;
10 **if** $t_i.type =$ *'literal'* **then**
11 $e \leftarrow \mathsf{ConstStr}(t_i.name)$;
12 add e to $\xi_{(i-1,i)}$;
13 **for** $i \in \{1, \ldots, n-1\}$ **do**
14 $\xi_{in} \leftarrow \{\forall e_p \in \xi_{(i-1,i)}, e_p \text{ is } \mathsf{Extract}\}$;
15 $\xi_{out} \leftarrow \{\forall e_q \in \xi_{(i,i+1)}, e_k \text{ is } \mathsf{Extract}\}$;
16 **for** $e_p \in \xi_{in}$ **do**
17 **for** $e_q \in \xi_{out}$ **do**
18 **if** $e_p.srcIdx + 1 = e_q.srcIdx$ **then**
19 $e \leftarrow \mathsf{Extract}(e_p.t_i, e_q.t_j)$;
20 add e to $\xi_{(i-1,i+1)}$;
21 $\mathcal{G} \leftarrow Dag(\widetilde{\eta}, \eta^s, \eta^t, \xi)$;
22 **Return** $\mathcal{G}$

Suppose $\mathfrak{T}$ is the set of all kinds of tokens (in our project, $\mathfrak{T}$ =[{digit}, {lower}, {upper}, {alpha}, {alpha-num}]), *ifValidSource* (denoted as $\mathcal{C}$) is defined as

$$\mathcal{C}(p_1, p_2) = \begin{cases} \text{true} & \text{if } \mathcal{Q}(\widetilde{\mathfrak{t}}, p_1) \geq \mathcal{Q}(\widetilde{\mathfrak{t}}, p_2), \forall \widetilde{\mathfrak{t}} \in \mathfrak{T} \\ \text{false} & \text{otherwise} \end{cases} \quad (2)$$

$\mathcal{Q}(\widetilde{\mathfrak{t}}, p)$ is the sum of the quantifiers of all tokens named $\widetilde{\mathfrak{t}}$ in pattern p, where $p = \{t_1, \ldots, t_n\}$ is a sequence of tokens.

Example 7. *Suppose the target pattern $\mathcal{T}$ in Example 3 is ['[', {upper}+, '-', {digit}+, ']'], we know*

$$\mathcal{Q}(\{digit\}, \mathcal{T}) = \mathcal{Q}(\{upper\}, \mathcal{T}) = 1$$

A pattern ['[', {upper}{3}, '-', {digit}{5}] derived from data record "[CPT-00350" will be identified as a ***source candidate*** *by* ***ifValidSource****, because*

$$\mathcal{Q}(\{digit\}, p) = 5 > \mathcal{Q}(\{digit\}, \mathcal{T}) \wedge$$
$$\mathcal{Q}(\{upper\}, p) = 3 > \mathcal{Q}(\{upper\}, \mathcal{T})$$

Another pattern ['[', {upper}{3}, '-'] derived from data record "[CPT-" will be denied because

$$\mathcal{Q}(\{digit\}, p) = 0 < \mathcal{Q}(\{digit\}, \mathcal{T})$$

5.2 Token Alignment

Once a source pattern is identified as a source candidate in Section 5.1, we need to synthesize an atomic transformation plan between this source pattern and the target pattern, which explains how to obtain the target pattern using the source pattern. To do this, we need to find the *token correspondences* for each token in the target pattern: discover all possible operations that yield a token. This process is denoted as *Token Alignment*.

For each token in the target pattern, there might be multiple different toke correspondences. Inspired by [14], we store the results of the token alignment as a $DAG(\widetilde{\eta}, \eta^s, \eta^t, \xi)$ using Directed Acyclic Graph (DAG). $\widetilde{\eta}$ denotes all the nodes in DAG with η^s as the source node and η^t as the target node. Each node corresponds to a position in the pattern. ξ are the edges between the nodes in $\widetilde{\eta}$ storing the source information, which yield the token(s) between the starting node and the ending node of the edge. Our proposed solution to token alignment in a DAG is presented in Algorithm 3.

Align Individual Tokens to Sources — To discover sources, given the target pattern $\mathcal{T}$ and the candidate source pattern $\mathcal{P}_{cand}$, we iterate through each token t_i in $\mathcal{T}$ and compare t_i with all the tokens in $\mathcal{P}_{cand}$.

For any source token t'_j in $\mathcal{P}_{cand}$ that is *syntactically similar* (defined in Definition 5.2) to the target token t_i in $\mathcal{T}$, we create a *match* between t'_j and t_i with an Extract operation on an edge from t_{i-1} to t_i (line 5-12).

Definition 5.2 (Syntactically Similar). *Two tokens t_i and t_j are syntactically similar if: 1) they have the same name, 2) their quantifiers are identical natural numbers or one of them is '+' and the other is a natural number.*

When t_i is a literal token, it is either a symbolic character or a constant value. To build such a token, we can simply use a ConstStr operation (line 10-12), instead of extracting it from the source pattern. This does not violate our previous assumption of not introducing any semantic data during the transformation.

Example 8. *Let the candidate source pattern be [digit{3}, '.', digit{3}, '.', digit{4}] and the target pattern be ['(', digit{3}, ')', ' ', digit{3}, '-', digit{4}]. Token alignment result for the source pattern $\mathcal{P}_{cand}$ and the target pattern $\mathcal{T}$, generated by Algorithm 3 is shown in Figure 9.*

Combine Sequential Extracts — The Extract operator in our proposed language UniFi is designed to extract one or more tokens sequentially from the source pattern. Line 7-12 only discovers sources composed of an Extract operation generating an individual token. *Sequential extracts* (Extract operations extracting multiple consecutive tokens from the source) are not discovered, and this token alignment solution is not complete. We need to find the *sequential extracts*.

Fortunately, discovering sequential extracts is not independent of the previous token alignment process; sequential extracts are combinations of individual extracts. With the alignment results ξ generated previously, we iterate each state and combine every pair of Extracts on an incoming edge and an outgoing edge that extract two consecutive tokens in the source pattern (line 13-20). The Extracts are then added back to ξ. Figure 10 visualizes combining two sequential Extracts, Extract({upper},0) and Extract({lower}+,0)

A benefit of discovering sequential extracts is it helps yield a "simple" program, as described in Section 5.3.

Correctness — This token alignment algorithm is *sound* and *complete*.

Theorem 5.1 (Soundness). *If the token alignment algorithm (Algorithm 3) successfully discovers a token correspondence, it can be transformed into a* UniFi *program.*

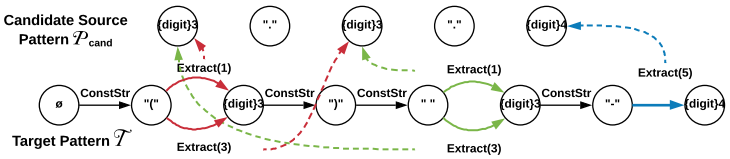


Figure 9: Token alignment for the target pattern $\mathcal{T}$

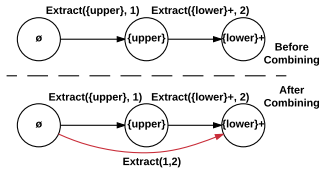


Figure 10: Combine Extracts

Figure 11: Preview Table

PROOF. Recall that an atomic transformation plan for a pair of source pattern and target pattern is a concatenation of Extract or ConstStr operations that sequentially generates each token in the target pattern. Every token correspondence discovered in Algorithm 3 corresponds to either a ConstStr operation or a Extract, both of which will generate one or several tokens in the target pattern. Hence, a token correspondence is can be possibly admitted into an atomic transformation plan, which will end up becoming part of a UNIFI program. The soundness is true. □

THEOREM 5.2 (COMPLETENESS). *The token alignment algorithm (Algorithm 3) can discover the corresponding token correspondence information for any* UNIFI *program.*

PROOF. Given the definition of the UNIFI and candidate source patterns, the completeness is true only when the token alignment algorithm can discover all possible parameterized Extract and/or ConstStr operations which combined will generate all tokens for the target pattern. In Algorithm 3, line 7-9 is certain to discover any Extract operation that extracts a single token in the source pattern and produces a single token in a target pattern; line 10-12 guarantees to discover any ConstStr operation that yields a single constant token in a target pattern. Given the design of our pattern profiling, an Extract of a single source token can not produce multiple target tokens, because such multiple target tokens, if exist, must have the same toke name, and should be merged as one token whose quantifier is the sum of the all these tokens. Similarly, the reverse is also true. What remains to prove is whether Algorithm 3 is guaranteed to generate an Extract of multiple tokens, a.k.a., Extract$(p, q)(p < q)$, in the source pattern that produces multiple tokens in the target pattern. In Algorithm 3, line 7-9 is guaranteed to discover Extract(p), Extract$(p+1)$, ..., Extract(q). With these Extracts, when performing line 14-20 when $i = p + 1$ in Algorithm 3, it will discover the incoming edge representing Extract(p) and the output edge representing Extract$(p + 1)$ and combine them, generating Extract$(p, p + 1)$. When $i = p + 2$, it will discover the incoming edge representing Extract$(p, p+1)$ and the outgoing edge representing Extract$(p + 2)$ and combine them, generating Extract$(p, p + 2)$. If we repeat this process, we will definitely find Extract(p, q) in the end. Therefore, the solution is complete. □

Complexity — Suppose the lengths of the source pattern and the target pattern is m and n, the time complexity of line 5-12 is $\mathcal{O}(mn)$. Line 13-20 iteratively checks the incoming and outgoing edges of each state, both of which could be of size $\mathcal{O}(m)$ in the worst case. Theoretically, line 14-20 could be as complex as $\mathcal{O}(m^2)$. However, the "nested for loops" are for convenience of presentation. In the actual implementation, we first sort both sets of the edges based on their source token id $srcIdx$ (complexity is $\mathcal{O}(m \log(m))$), and then find sequential extracts in linear time (complexity is $\mathcal{O}(m)$). Therefore, the complexity of line 14-20 becomes $\mathcal{O}(mn \log(m))$, which is also the overall time complexity.

5.3 Program Synthesis using Token Alignment Result

Given all token correspondence information discovered in token alignment, we formulate an atomic transformation plan for each candidate source pattern $\mathcal{P}_{cand}$, converting it into the desired target pattern $\mathcal{T}$.

As we represent all token correspondences for a source pattern as a DAG (Algorithm 3), finding a good transformation plan is to find a short path from state 0 to state l, where l is the length of the target pattern $\mathcal{T}$.

There can be multiple paths from the source to target in the DAG, indicating there are different possible atomic transformation plans. Obviously, not all of these plans are equally likely to be correct and desired by the end user. We hope to prioritize the plans expected by the user. To rank the possible plans, we follow **Occam's razor principle**: the simplest explanation is usually correct. From our observation, a program is simple if it is **1)** composed of fewer operations **2)** composed of fewer ConstStr operations. Despite being subjective, these criteria are helpful in prioritizing correct programs as shown in our experiments. We use a variant of Dijkstra's algorithm to find top k "simplest" paths in the DAG.

Based on our knowledge, extracting the same token multiple times in an actual transformation almost never happens. In this case, we assume a generated atomic transformation plan must follow a one-to-one match between the source and target tokens. For example, two Extract(1) operations cannot co-exist in a generated plan. We remove any plan discovered by the above algorithm that violates this assumption, which helps improve the quality of the synthesized program.

In the end we will generate k possible atomic transformation plans, ranked by the *simplicity*, for every candidate source pattern. The first plan is the *initial* solution UNIFACTA believes is correct for each pattern. The final UNIFI program synthesized is essentially the set of patterns with their initial atomic transformation plans.

5.4 UNIFI Programs Explanation

As stated in Section 1, the *Verification* challenge in existing PBE string transformation systems partially results from the lack of explanation of the synthesized transformation logic. The end user neither understands nor is able to adjust the underlying transformation program being generated, which sometimes makes the system unusable. Although there are other PBE systems (e.g., [21]) that present the synthesized logic to the user, their programs are usually hard to read for non-experts.

UNIFACTA in the end translates the synthesized UNIFI program into a set of Replace [3] operations parameterized by *natural language regular expressions* used by Wrangler [23] (e.g., Figure 4). To allow the user understand each atomic Replace operation in the blink of an eye, we also added a *Preview Table* (e.g., Figure 11) to visualize the transformation effect in our prototype in a sample of the input data.

The user study in Section 6.6 demonstrates that our effort adding an explanation step is helpful for the end user to understand what transformation is generated by the system.

5.5 Program Adjustment

Sometimes, the atomic transformation plan we *initially* select for each source pattern in Section 5.3 can be incorrect. The reason is that the target pattern $\mathcal{T}$ as the sole user input so far is more ambiguous compared to input-output example pairs used in most other PBE systems. For instance, in Figure 10, the first "{digit}3" token in the target pattern can be extracted from either the first or second "{digit}3" token in the source pattern. Although our algorithm described in Section 5.3 often makes good guesses about the right correspondences, the system still infers an imperfect transformation about 50% of the time (Section 6.2). In such cases, the user could manually *adjust* the atomic transformation plan: replace the initial atomic transformation plan with another atomic transformation plans among the ones Section 5.3 suggests for a given source pattern.

The Token Alignment Algorithm and Program Synthesis Algorithm from Section 5.2 and 5.3 have already made the space of possible plans reasonably small, we want to further prune the space of possible programs using the concept of an *equivalence class* to minimize the user effort.

Prune Equivalent Transformations — Before the adjustment phase, we delete extra transformation plans within the same *equivalence class*. Two transformation plans are *equivalent* if they conform to Definition 5.3.

DEFINITION 5.3 (EQUIVALENT PLANS). *Two Transformation Plans are* equivalent *if, given the same source pattern, they always yield the same transformation result for any string matching the source pattern.*

For instance, suppose the source pattern is [digit{2}, '/', digit{2}], if transformation plans $\mathcal{E}_1$ is [Extract(3), Const('/'), Extract(1)] and transformation plans $\mathcal{E}_2$ is [Extract(3), Extract(2), Extract(1)], their final transformation result should be exactly the same and the only difference between $\mathcal{E}_1$ and $\mathcal{E}_2$ is the source of '/'. Presenting such equivalent transformations to the user will not be helpful but increase the user effort. Hence, we only pick the simplest plan in the same equivalence class and prune the rest.

Overall, the adjustment process does not significantly increase the user effort. In those cases where the initial program is imperfect, 75% of the time the user made just a single adjustment (Section 6.2).

6. EXPERIMENTS

In this section, we report on our experimental evaluation of UNIFACTA. We implemented UNIFACTA on TRIFACTA, which is the commercial version of the well-known data wrangling tool WRANGLER, and compared it against two state-of-the-art PBE systems FLASHFILL and BLINKFILL. We also compared it with unmodified TRIFACTA. All experiments were performed on a 4-core Intel Core i7 2.8G CPU with 16GB RAM.

We first present our benchmarks and then our evaluation using the benchmarks to answer several questions:

- How effective and expressive are UNIFACTA and the proposed pattern standardization language UNIFI (Section 6.2)?
- How difficult it is to use UNIFACTA (Section 6.2, 6.4)?
- Does UNIFACTA save user effort compared to the baseline systems? (Sections 6.3, 6.4)
- Does the system scale when the size and the heterogeneity of the data increase (Section 6.5)?
- Do the programs UNIFACTA generates help the user understand the transformation logic? (Section 6.6)

6.1 Benchmarks

We created a benchmark of 47 pattern standardization test cases using a mixture of public string transformation test sets and example tasks from related research publications[4]. 27 test tasks are from SyGus (Syntax-guided Synthesis Competition), which is a program synthesis contest held every year. In 2017, SyGus revealed 108 string transformation tasks in its Programming by Examples Track: 27 unique scenarios and 4 tasks of different sizes for each scenario. We collected the task with the longest data set in each scenario and formulated the pattern standardization benchmarks of 27 tasks. We collected 10 tasks from FlashFill [14]. There are 14 in their paper. Four tests (Example 4, 5, 6, 14) require a loop structure in the transformation program which is not supported in UNIFI and we filter them out. Additionally, we collected 4 tasks from BLINKFILL [32], 3 tasks from PredProg [34], 3 tasks from Microsoft PROSE SDK.

For test scenarios with very little data, we asked a Computer Science student not involved with this project to synthesize more data. Thus, we have sufficient data for evaluation later. The average sizes of the collected benchmark test cases are shown in Table 5. Also, the current UNIFACTA prototype system requires at least one data record in the target pattern. For any benchmark task, if the input data set violated this assumption, we randomly converted a few data records into the desired format and used these transformed data records and the original input data to formulate the new input data set for the benchmark task. The heterogeneity of our benchmark tests comes from the input data and their diverse pattern representations in the pattern language described previously in the paper.

SyGus	FlashFill	BlinkFill	PredProg	Prose	Overall
63.3	10.3	10.8	10	39.3	43.6

Table 5: Average size (# of rows) of benchmark test cases

6.2 Effectiveness and Efficiency of UNIFACTA

In this section we study whether UNIFACTA is able to infer transformation programs correctly (effectiveness), and how much user effort is required to get there (efficiency).

[3]Replace a string matching the given regular expression with another string

[4]https://github.com/markjin1990/unifacta-benchmarks

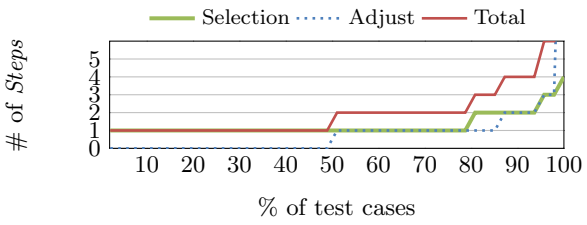


Figure 12: Percentage of test cases costing $\leq$ Y *Steps* in different UNIFACTA interaction phases

Overview — We evaluated UNIFACTA against 47 benchmark tests. As conducting an actual user study on all 47 benchmarks is impossible, we simulated a user following the "lazy approach" used by Gulwani et al. [17] and evaluated the effectiveness and efficiency. The user selected a target pattern or multiple target patterns and then adjusted the atomic transformation plan for each source pattern if the default one was imperfect.

Metrics — In experiments, we measured how much user effort UNIFACTA system consumed. To quantify the user effort, we define its unit as *Step*. In each test, the total *Steps* (Total) is defined as the sum of the number of correct patterns the user chooses (Selection) and the number of adjustments for the source patterns whose default atomic transformation plans are incorrect (Adjust). In the end, we also check if the system has synthesized a "perfect" program: a program that successfully standardizes all the data.

Results — Overall, UNIFACTA synthesized perfect transformations for 42/47 (∼90%) test cases, which suggests UNIFACTA, including the UNIFI language, is quite expressive.

There are five test cases where UNIFACTA failed to yield a perfect transformation. "Example 13" in FlashFill requires the inference of advanced conditionals (Contains keyword "picture") that UNIFI can not express. Supporting more advanced conditionals is in the future plan. The failures in the remaining four test cases were mainly caused by the lack of the target pattern examples in the data set. For example, one of the test cases we failed is a name standardization task, where there is a last name "McMillan" to extract. However, all data in target pattern contain last names of one uppercase letter followed by multiple lowercase letters and hence our system did not realize "McMillan" needed to be extracted. We think if the input data is large and representative enough, we should be able to successfully capture all desired data patterns.

Figure 12 is a breakdown of the user effort. The y-axis is the number of *Steps*; the x-axis denotes the percentage of test cases that costs less than or equal to the number of *Steps* indicated by y-axis. For around 79% of the test cases, UNIFACTA is able to infer a perfect pattern standardization program within two steps. Also, the user needs to select only one target pattern in the initial step for about 79% of the test cases, which proves that our pattern profiling technique is usually effective in grouping data under the same pattern.

Additionally, we observe that the user needs to make no adjustment for the suggested transformations in about 50% of the test cases and ≤ 1 adjustment in about 85% of the test cases. This shows that Occam's razor principle we follow and the algorithm we design is effective in prioritizing the correct transformations and producing quality results.

Note that, in test case "popl-13.ecr" from PROSE, UNIFACTA consumed tremendous user effort because the data are a combination of human names, organization names and country names. All these names do not share a distinctive syntax for us to identify. For example, if the user wants to extract both "INRIA" and "Univ. of California", the user might have to select both "{upper}+" and "{upper}{lower}+ {lower}+ {upper}{lower}+" as the target patterns, and adjust more later. This increases the user selection and adjustment effort.

However, this problem can be easily solved by suggesting a single operation of Extract between two commas for all source patterns once we identify the "comma" is a "StructProphecy" [5] using the methodology proposed by [9]. In this case, the user effort is substantially reduced.

6.3 Comparison with Related Work

In this section, we compare the efficiency of PBE systems FLASHFILL and BLINKFILL, and PBD system TRIFACTA with the UNIFACTA prototype.

Overview — We tested all three baseline systems against 47 test cases in the benchmark test suite. Similarly, we defined an user on FLASHFILL and BLINKFILL which provided the first positive example on the first data record in non-standard pattern, and then recursively provided positive examples on the data record on which the synthetic string transformation program failed. On TRIFACTA, as Replace is the most appropriate operator for string transformation, the simulated user specified a Replace operation with two regular expressions indicating the matching string pattern and the transformed pattern, and recursively specified new parameterized Replace operations for the next ill-formatted data record until all data were in the correct format.

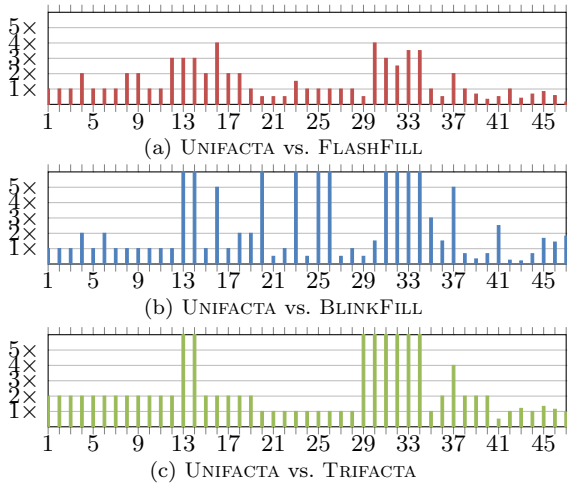
Evaluation Metrics — Because UNIFACTA follows an interaction model different from all three baselines, a direct comparison of the user effort is impossible. We redefine the user effort unit *Step* for all baselines. Defining a *Step* that costs the same amount of user effort for different intersection phases for one system is very difficult, and doing so among all systems is even harder. In this experiment, we only intend to have a coarse estimation of the user effort in all four systems on the 47 benchmarks. Later in Section 6.4, we present our experiment results of an actual user study on a sample of the benchmarks.

For FLASHFILL and BLINKFILL, the total *Steps* is the sum of the number of input examples to provide and the number of data records that the system fails to transform. As for TRIFACTA, each specified Replace operation is counted as 2 *Steps* as the user needs to type two regular expressions for each Replace, which is about twice the effort of giving an example in FLASHFILL and BLINKFILL.

In the end of each test, for each system, we add the number of data records that the system fails to transform correctly to its total *Step* value as a punishment.

Results — Figure 13 shows the overall *speedup* of UNIFACTA over all three other baselines. The y-axis is the value of *speedup*: number of *Steps* cost in UNIFACTA over the

[5] A pattern with "StructProphency" tokens is a pattern of k fields separated by $k-1$ struct tokens; in "popl-13.ecr" from PROSE, all source data are three name fields separated by two commas, and we want to extract the field in the middle



(a) UNIFACTA vs. FLASHFILL

(b) UNIFACTA vs. BLINKFILL

(c) UNIFACTA vs. TRIFACTA

Figure 13: Speedup: # of *Step*s ratio for 47 test cases

Baselines	UNIFACTA Wins	Tie	UNIFACTA Loses
vs. FLASHFILL	17 (36%)	17 (36%)	13 (28%)
vs. BLINKFILL	23 (49%)	14 (30%)	10 (21%)
vs. TRIFACTA	33 (70%)	12 (26%)	2 (4%)

Table 6: User effort simulation comparison.

other baseline system. The x-axis denotes the benchmark id. Since the *Step* metric is a potentially noisy measure of user effort, it is more reasonable to check whether UNIFACTA costs more or less effort than other baselines, rather than to compare absolute *Step* numbers. The result is summarized in the result is summarized in Table 6. It suggests UNIFACTA often requires less or at least equal user effort than both PBE systems. Compared to the PBD system TRIFACTA, UNIFACTA almost always costs less or equal user effort.

BLINKFILL failed to outperform FLASHFILL in many cases. The reason was that BLINKFILL is less expressive than FLASHFILL and cannot successfully express all necessary transformations for our workload, which contains multiple heterogeneous patterns. In fact, BLINKFILL failed to infer a perfect transformation in 19 benchmarks (40%). Because of this, we choose FLASHFILL, instead of BLINKFILL, as the only PBE system evaluated in the user study in Section 6.4.

6.4 User Study

We used a crude metric of counting *Steps* to measure user effort in the previous section. The user effort can vary between steps, so the count of *Steps* is only an estimate of user effort. Furthermore, even the number of steps is estimated based on a specific model of user behavior, which may not reflect reality perfectly. Therefore, we performed a user study to evaluate how efficiently UNIFACTA can help an actual user with the pattern standardization task. The participants were requested to perform pattern standardization tasks on UNIFACTA, FLASHFILL and TRIFACTA.

Overview — Since it was impractical to perform a user study on all 47 tasks, we had to limit this study to just a few tasks. We already have a rough idea of user effort from the *Step* metric used in the last section. The most interesting comparisons are between tasks estimated by the *Step* metric to be equally difficult for users using either UNIFACTA or FLASHFILL. In the benchmark suite, there are 17 tasks for which UNIFACTA and FLASHFILL tied in user effort with the *Step* metric. We randomly chose 3 test cases from these 17: Example 11 from FlashFill (task 1), Example 3 from PredProg (task 2) and "phone-10-long" in SyGus (Task 3).

We invited 9 students in Computer Science with a basic understanding of regular expressions and not involved in our project. Before the experiment, we educated participants on how to use all three systems. We then ask every user solve the above three tests, each on a different system. The task completion time was measured in each case.

Results — The average completion time for each task using the baseline systems and UNIFACTA is presented in Figure 14a. Compared to FLASHFILL, the participants using UNIFACTA spent around 30% less time on average: 70 % less time on task 1 and around 60% less time on task 3, but about 40% more time on task 2. As the chosen tasks are considered to consume same effort on both systems, the result strengthens the conclusion in Section 6.2 that UNIFACTA typically costs less user effort if not equal than FLASHFILL.

The heterogeneity of task 1 is in fact low, and the participants seemed to have no difficulty using UNIFACTA to find a correct transformation. On FLASHFILL, the *Specification* challenge was evident, because the underlying transformation logic is unknown to the normal user. The participants usually provided more examples than necessary and occasionally provided incorrect positive examples which slowed down the whole transformation process.

The heterogeneity of the test data in task 1 and task 3 are about the same but task 1 has 10 data records and task 3 has 100 data records. From our observation, the participants using FLASHFILL typically spent much more time on understanding the data formats at the beginning and verifying the transformation result in solving task 3. This complies to the *Verification* challenge we claim in Section 1. In comparison, due to the pattern information UNIFACTA provides, the participants did not spend much time understanding and verifying the data. Hence, UNIFACTA cost less time.

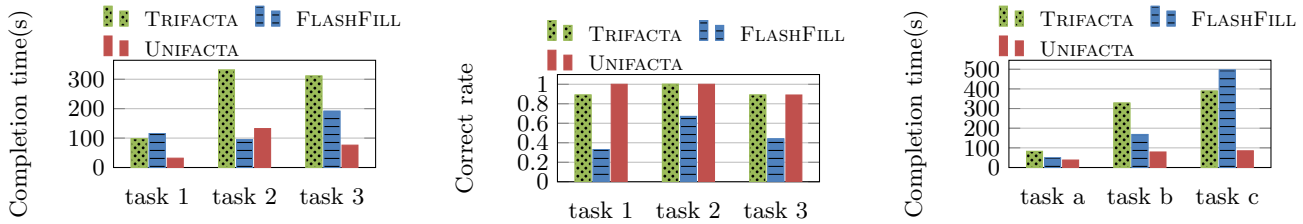
The test data of task 2 is more heterogeneous than that of task 1 and task 3, but has only 10 data records. The participant spent most of the time adjusting the atomic transformation plans for source patterns using UNIFACTA. In comparison, the participants did not spend as much time using FLASHFILL simply because task also has only 10 data records. In fact, the user spent most of time making corrections using FLASHFILL for task 2 because the transformation inferred by FLASHFILL was unsuccessful and the participants often made corrections for a larger portion of the data records.

Compared to TRIFACTA, the effort saved by UNIFACTA is more impressive, about 70% on average. This is because writing and verifying regular expressions on TRIFACTA is are more challenging tasks.

6.5 User Effort on Scalability

In this section, we further investigate the user effort scalability issue: whether increasing the heterogeneity and data size of pattern standardization tasks significantly increases the user effort on FLASHFILL like the *Specification* and *Verification* challenges we claim, and also, how much UNIFACTA could alleviate the situation. Additionally, we also tested TRIFACTA.

Overview — In our benchmark suite, there is no test case with sufficient data records. For this study, we used a col-



(a) User effort test (shorter bars are better)

(b) User comprehension test (taller bars are better)

(c) User effort scalability test (shorter bars are better)

Figure 14: 14a shows the average completion time of 3 user study tasks on both systems; 14b shows the overall correct rates of the 3 user study tasks; 14c shows the completion time as data size and heterogeneity scales ("task a": 10 rows & 2 patterns; "task b": 100 rows & 4 patterns; "task c": 300 rows & 6 patterns)

umn of phone numbers from the "Times Square Food & Beverage Locations" dataset[6].

The pattern standardization task is to transform all phone numbers into the form "{digit}3-{digit}3-{digit}4". We created three test cases with various data size and heterogeneity by randomly sampling the raw dataset: "task a" has 10 data records and 2 patterns; "task b" has 100 data records and 4 patterns; "task c" has 300 data records and 6 patterns.

We let 9 participants each work on one test case on all three systems and measured their completion time.

Results — Figure 14c shows the overall results of this scalability test. Compared to UNIFACTA, FLASHFILL cost 29% more time in "task a", 1.1x more time in "task b", and 4.8x more time in "task c". These numbers suggest that, as the size the data and the heterogeneity increases, the user effort consumed by FLASHFILL grows much faster than UNIFACTA. The reason that UNIFACTA's user effort grew slowly is that the user effort in UNIFACTA is primarily affected by the heterogeneity, less by the size of the data, since the interaction is at the pattern level. In comparison, both the heterogeneity and the size of the data influence user effort in FLASHFILL; the user has to identify the non-standard patterns and verify the transformations row by row. Thus, the user effort scales with data volume and heterogeneity. The experiment results further demonstrate that the *Specification* and *Verification* challenges exist in standardizing real-world large, heterogeneous data, and UNIFACTA largely ameliorates these problems. TRIFACTA, on average, cost 2.6x more time than UNIFACTA, which suggests UNIFACTA scales better than TRIFACTA. The user effort of TRIFACTA seems to grow slower than FLASHFILL in "task c", which we believe is because the synthesized program is presented to the user and easier to verify.

6.6 Effectiveness of Program Explanation

We also want to understand whether the UNIFACTA architecture, and the facilities described in Section 5.4, actually help the user understand the transformation process. We evaluated comprehensibility of the synthetic transformations of UNIFACTA against the baseline system FLASHFILL in the user study.

Overview — We designed 3 multiple choice questions for every task in Section 6.4 examining how well the user understood the transformation regardless of the system he/she interacted with. All the questions are formulated as "Given the input string as x, what is the expected output"? Each participant was asked to answer these questions based on the transformation results or the synthetic programs obtained after completion of the task in the user study in Section 6.4. All questions are shown in Appendix A.

Results — The correct rates for all 3 tasks using all systems are presented in Figure 14b. The result shows that the participants were able to answer these questions almost perfectly using UNIFACTA, but struggled to get even half correct using FLASHFILL. TRIFACTA also achieved a success rate similar to TRIFACTA, but at a higher cost of user effort.

Additionally, we performed a subjective observational study. When participants were answering the questions, we noticed that they tended to have more confidence when they were using UNIFACTA. If they were using FLASHFILL, they hesitated to give an answer and occasionally made random guesses. Also, when FLASHFILL users were shown the correct answer (after they had submitted their own answer to the question), they were sometimes confused and surprised.

7. RELATED WORK

Data Transformation — Data transformation involves syntactic transformations, layout transformation and semantic transformation.

Syntactic transformation manipulates a string as a sequence of characters. FLASHFILL (now a feature in Excel) is an influential work for syntactic transformation by Gulwani [14]. It designed an expressive string transformation language and proposed the algorithm based on version space algebra to discover a program in the designed language. It was recently integrated to PROSE SDK released by Microsoft. Other related work include [32, 36].

Layout transformation mainly relocates the cells in a tabular form. The seminal work WRANGLER created by Kandel et al. [23] provides a user-friendly interaction framework which suggests useful parameterized data cleaning operations when the user works on different part of the data. WRANGLER mainly focus on layout transformation, and some syntactic transformation enabled by Replace, Split and Merge operators. Other related work include [17, 3, 22, 21].

Semantic transformation transforms string based on semantic meanings. Previous work [6, 33, 2] are aimed at addressing transformations as such. We plan to provide support for semantic transformation in future work.

Data Profiling — Data profiling is the discovery of the metadata of an unknown dataset or database [1]. Some

[6]Available at `https://opendata.cityofnewyork.us/`

of the metadata that are commonly profiled in a dataset include number of null values, distinct values, and redundancies. Researchers also have made progress in profiling foreign keys [31], functional dependencies [37, 18] and inclusion dependencies [28, 4] in a relational database.

In our project, we focus on profiling the pattern structure of ad hoc string data. The LearnPADS [9, 8] project shares a similar goal. It proposes a learning algorithm using statistics over symbols and tokenized data chunks to discover pattern structure. LearnPADS assumes that all data entries follow a repeating high-level pattern structure. However, this assumption may not hold for some of the workload elements. In contrast, we create a bottom-up pattern discovery algorithm that does not make this assumption. Plus, the output of LearnPADS (i.e., PADS program [7]) is hard for human to read, whereas our pattern tree is simpler to understand. Most recently, Datamaran[10] has proposed methodologies discovering structure information in a dataset whose record boundaries are unknown, but for the same reasons as LearnPADS, Datamaran is not suitable for our problem.

Program Synthesis — Program synthesis has drawn wide interest in domains where the end users might not have good programming skills or programs are hard to maintain or reuse including data science and database systems. Researchers have built various program synthesis applications to generate SQL queries [35, 30, 26], regular expressions [5, 27], data cleaning programs [14, 21, 36],

Researchers have proposed various techniques for program synthesis [13]. [16, 20] proposed a constraint-based program synthesis technique using logic solvers. However, constraint-based techniques are mainly applicable in the context where finding a satisfying solution is challenging, but we prefer a high quality program rather than a satisfying program. Version space algebra is another important technique that is applied by Mitchell [29, 24, 14, 25]. Most of these projects rely on the user inputs to reduce the search space until a quality program can be discovered and share the same hope that there is one simple solution matching most, if not all, user-provide example pairs. In our case, transformation plans for different heterogeneous patterns can be quite different, and hence applying version space algebra technique is difficult.

8. CONCLUSION AND FUTURE WORK

In this paper, we have presented a novel technique, Unifacta, for standardizing ad hoc data with heterogeneous patterns. Unlike the PBD or PBE interaction models used by previous data cleaning projects, we propose a new model where the end user only selects the desired data pattern and our technique helps the user synthesize high quality, explainable transformations that convert all non-standard patterns into the desired pattern.

To derive pattern descriptions that are easy to understand, we create a bottom-up pattern profiling method. We design graph-based algorithms to synthesize the transformation logic represented by our proposed pattern standardization language, UniFi, which is later explained to the user.

The experiments show that Unifacta is able to handle most of the pattern standardization benchmarks we collected, and often with less user effort compared to the baseline systems FlashFill and Trifacta. In the user study, Unifacta on average cost 56%-70% less user effort than the baselines. The user effort scalability test shows that the user effort of Unifacta increases much slower than FlashFill when the size and the heterogeneity of the data set increases. In the experiment examining the user's understanding of the synthesized transformation logic, Unifacta users achieved a success rate 101% higher than FlashFill users.

In the future, we would like to extend Unifacta with the support for semantic transformation. Additionally, we also plan to expand the pattern standardization language, e.g., introducing new conditionals besides pattern matching.

9. REFERENCES

[1] Z. Abedjan, L. Golab, and F. Naumann. Profiling relational data: a survey. *The VLDB Journal*, 24(4):557–581, 2015.

[2] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: A robust transformation discovery system. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 1134–1145. IEEE, 2016.

[3] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. Flashrelate: extracting relational data from semi-structured spreadsheets using examples. In *ACM SIGPLAN Notices*, volume 50, pages 218–228. ACM, 2015.

[4] J. Bauckmann, U. Leser, F. Naumann, and V. Tietz. Efficiently detecting inclusion dependencies. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1448–1450. IEEE, 2007.

[5] A. Blackwell. Swyn: A visual representation for regular expressions. *Your Wish Is My Command: Programming by Example*, pages 245–270, 2001.

[6] Z. Chen, M. Cafarella, and H. Jagadish. Long-tail vocabulary dictionary extraction from the web. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, pages 625–634. ACM, 2016.

[7] K. Fisher and R. Gruber. Pads: A domain-specific language for processing ad hoc data. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 295–304, New York, NY, USA, 2005. ACM.

[8] K. Fisher, D. Walker, and K. Q. Zhu. Learnpads: automatic tool generation from ad hoc data. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1299–1302. ACM, 2008.

[9] K. Fisher, D. Walker, K. Q. Zhu, and P. White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 421–434, New York, NY, USA, 2008. ACM.

[10] Y. Gao, S. Huang, and A. G. Parameswaran. Navigating the data lake with datamaran: Automatically extracting structure from log datasets. In *SIGMOD*, 2018.

[11] S. Gardiner, A. Tomasic, J. Zimmerman, R. Aziz, and K. Rivard. Mixer: mixed-initiative data retrieval and integration by example. In *IFIP Conference on*

Human-Computer Interaction, pages 426–443. Springer, 2011.

[12] M. Gorinova, K. Prince, S. Meakins, A. Vuylsteke, M. Jones, and A. Blackwell. The end-user programming challenge of data wrangling. In *PPIG*, 2016.

[13] S. Gulwani. Dimensions in program synthesis. In *PPDP*, pages 13–24. ACM, 2010.

[14] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 317–330, New York, NY, USA, 2011. ACM.

[15] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communications of the ACM*, 55(8):97–105, 2012.

[16] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. volume 46, pages 62–73. ACM, 2011.

[17] W. R. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *ACM SIGPLAN Notices*, volume 46, pages 317–328. ACM, 2011.

[18] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. Tane: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal*, 42(2):100–111, 1999.

[19] D. Huynh and S. Mazzocchi. Openrefine, 2012.

[20] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224. ACM, 2010.

[21] Z. Jin, M. R. Anderson, M. Cafarella, and H. V. Jagadish. Foofah: A programming-by-example system for synthesizing data transformation programs. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1607–1610, New York, NY, USA, 2017. ACM.

[22] Z. Jin, M. R. Anderson, M. Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 683–698, New York, NY, USA, 2017. ACM.

[23] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 3363–3372, New York, NY, USA, 2011. ACM.

[24] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. Programming by demonstration using version space algebra. *Machine Learning*, 53(1):111–156, 2003.

[25] T. A. Lau, P. M. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

[26] F. Li and H. Jagadish. Constructing an interactive natural language interface for relational databases. volume 8, pages 73–84. VLDB Endowment, 2014.

[27] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Jagadish. Regular expression learning for information extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 21–30. Association for Computational Linguistics, 2008.

[28] S. Lopes, J.-M. Petit, and F. Toumani. Discovering interesting inclusion dependencies: application to logical database tuning. *Information Systems*, 27(1):1–19, 2002.

[29] T. M. Mitchell. Generalization as search. *Artificial intelligence*, 18(2):203–226, 1982.

[30] L. Qian, M. J. Cafarella, and H. Jagadish. Sample-driven schema mapping. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 73–84. ACM, 2012.

[31] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In *WebDB*, 2009.

[32] R. Singh. Blinkfill: Semi-supervised programming by example for syntactic string transformations. *Proc. VLDB Endow.*, 9(10):816–827, June 2016.

[33] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *Proceedings of the VLDB Endowment*, 5(8):740–751, 2012.

[34] R. Singh and S. Gulwani. Predicting a correct program in programming by example. In *International Conference on Computer Aided Verification*, pages 398–414. Springer, 2015.

[35] C. Wang, A. Cheung, and R. Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 452–466. ACM, 2017.

[36] B. Wu and C. A. Knoblock. An iterative approach to synthesize data transformation programs. In *IJCAI*, pages 1726–1732, 2015.

[37] H. Yao and H. J. Hamilton. Mining functional dependencies from data. *Data Mining and Knowledge Discovery*, 16(2):197–219, 2008.

[38] K. Zhu, K. Fisher, and D. Walker. Learnpads++: Incremental inference of ad hoc data formats. *Practical Aspects of Declarative Languages*, pages 168–182, 2012.

APPENDIX

A. QUESTIONS USED IN PROGRAM EXPLANATION EXPERIMENT

1. For task 1, if the input string is "Barack Obama", what is the output?
    A. Obama
    B. Barack, Obama
    C. Obama, Barack
    D. None of the above
2. For task 1, if the input string is "Barack Hussein Obama", what is the output?
    A. Obama, Barack Hussein
    B. Obama, Barack
    C. Obama, Hussein
    D. None of the above
3. For task 1, if the input string is "Obama, Barack Hussein", what is the output?
    A. Obama, Barack Hussein
    B. Obama, Barack

C. Obama, Hussein

D. None of the above

4. For task 2, if the input is "155 Main St, San Diego, CA 92173", what is the output
   A. San
   B. San Diego
   C. St, San
   D. None of the above
5. For task 2, if the input string is "14820 NE 36th Street, Redmond, WA 98052", what is the output?
   A. Redmond
   B. WA
   C. Street, Redmond
   D. None of the above
6. For task 2, if the input is "12 South Michigan Ave, Chicago", what is the output?
   A. South Michigan
   B. Chicago
   C. Ave, Chicago
   D. None of the above
7. For task 3, if the input string is "+1 (844) 332-282", what is the output?
   A. +1 (844) 282-332
   B. +1 (844) 332-282
   C. +1 (844)332-282
   D. None of the above
8. For task 3, if the input string is "844.332.282", what is the output?
   A. +844 (332)-282
   B. +844 (332) 332-282
   C. +1 (844) 332-282
   D. None of the above
9. For task 3, if the input string is "+1 (844) 332-282 ext57", what is the output?
   A. +1 (844) 322-282
   B. +1 (844) 322-282 ext57
   C. +1 (844) 282-282 ext57
   D. None of the above