Democratizing Self-Service Data Preparation through Example-Guided Program Synthesis

by

Zhongjun Jin

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2020

Zhongjun Jin
markjin@umich.edu
ORCID iD: 0000-0003-1833-8061

To my wonderful girlfriend Siyu and my supportive parents Mr. Shaohong Jin and Mrs. Ying Hu.

Acknowledgments

I would like to express my deepest appreciation to my two advisors, shifu, and friends, Michael Cafarella and H. V. Jagadish, for their relentless support and dedicated guidance during my PhD journey. Mike and Jag always put their students at the top of the list. Thanks to them, this adventure was ample of personal development, self-fulfillment, cheerfulness, and only a thin slice of bitterness and frustration that usually came along with paper acceptance notifications. They have been and will always be my role models that inspire and motivate me to become a humble and kind person as they are. Special thanks to Prof. Joe Hellerstein at UC Berkeley who generously helped me reshape my second project and gave constructive feedback. It is a heavenly bliss to have ever crossed paths with Joe. His seminal work in data wrangling also has a large impact on this dissertation. I also want to extend my sincere thanks to my committee members, Profs. Kevyn Collins-Thompson and Jenna Wiens, for their insightful feedback and generous help.

I am also grateful to all my previous collaborators. Mike Anderson is my first-ever collaborator in grad school and the best colleague I can dream of. Mike dedicated lots of time helping me get started in my first project Foofah and stayed supportive and a friend that I cannot spend more time with ever since. Chris Baik has been a long-time good buddy of mine and we spent remarkably long time everyday exchanging ideas, sharing new discoveries or simply chitchating about our PhD lives. We came up with many crazy project ideas and half of them miserably failed in the end which I usually took full responsibility for. Chris is a true Christian that is always there for me when I turn to him for help. Abolfazl Asudeh is another of my close but more senior collaborator, a mastermind that can develop solid mathematical foundation on top of every problem and solution, a genius that I always admire but never possess. Lastly, Haozhu Wang, a dedicated deep learning researcher, offered me tremendous help in developing a follow-up project that cannot go to this dissertation.

I also want to thank the rest of my database group folks and office mates at various points without whom my school life would be bland and insipid: Prof. Barzan Mozafari, Prof. Murali Mani, Prof. Prakash Ramanan, Yuval Moskovitch, Jie Song, Rui Liu, Junghwan Kim, Wenjia He, Irene Yin Lin, Yuze Lou, Mengjing Xu, Junjie Xing, Dong Young Yoon, Nikita Bhutani, Jiamin Huang, Yongjoo Park, Dolan Antenucci, Fei Li, Bernardo Gonçalves, Shirley Zhe Chen, Minghe Yu, Cheng Long, Aditi Sharma, Boyu Tian, Changfeng Liu, Ishan Patney, Jarrid Rector-Brooks, Giovanni Simonini, Vaspol Ruamviboonsuk, Nan Zheng, Li Sun, Jaeyoung Kim, Mahmood Barangi, Idongesit Ebong, Mahdi Aghadjani, Yalcin Yilmaz, Caleb Belth, Alican Buyukcakir, Di Jin, Puja Trivedi, Yujun Yan, Earlence Fernandes, Amir Rahmati and Kevin Eykholt.

Finally, I'd like to thank my family: my father—the first college graduate in my whole family—Shaohong Jin, my mother—a strong woman who, together with my father, built a successful business in environmental protection at my hometown, Liaoning, China—Ying Hu, who has been nothing but supportive of my decision studying abroad and pursuing a PhD degree, both emotionally and sometimes financially; my girlfriend and a bright and brave girl pursuing PhD in STEM, Siyu Zheng, who has been my listening ear and source of joy.

Table of Contents

Dedication **ii**

Acknowledgments **iii**

List of Figures **viii**

List of Tables **xi**

Abstract **xii**

Chapter 1. Introduction **1**

1.1 Challenges and Opportunities 4

1.1.1 Challenges for users 5

1.1.2 Technical challenges 7

1.2 Summary of Contributions 9

Chapter 2. Research Background **11**

2.1 Data Preparation Pipeline 11

2.1.1 Data Transformation 11

2.1.2 Data Pattern Profiling 14

2.1.3 Data Integration and Schema Mapping 16

2.2 Program Synthesis 17

Chapter 3. Synthesizing Data Transformation Programs using User Examples **19**

3.1 Introduction 19

3.2 Motivating Example 23

3.3 Problem Definition 26
3.3.1 Problem Definition 26
3.3.2 Data Transformation Programs 27
3.4 Program Synthesis 30
3.4.1 Program Synthesis Techniques 30
3.4.2 Search-based Program Synthesis 32
3.4.3 Pruning Techniques for Better Efficiency 44
3.4.4 Complexity Analysis 46
3.4.5 Synthesizing Perfect Programs 46
3.5 Experiments 48
3.5.1 Benchmarks 49
3.5.2 Performance Evaluation 50
3.5.3 Comparing Search Strategies 52
3.5.4 Effectiveness of Pruning Rules 54
3.5.5 Adaptiveness to New Operators 54
3.5.6 User Effort Study 55
3.5.7 Comparison with Other Systems 57
3.6 Related Work 60
3.7 Conclusion 62

Chapter 4. Synthesizing Data Format Standardization Programs using Pattern-based Examples 63
4.1 Introduction 63
4.2 Motivating Example 67
4.3 Overview 70
4.3.1 Patterns and Data Transformation Problem 70
4.3.2 Clx Data Transformation Paradigm 73
4.4 Clustering data on patterns 74
4.4.1 Initial Clustering Through Tokenization 75
4.4.2 Bottom-up Pattern Cluster Refinement 76
4.4.3 Limitations 79
4.5 Data Pattern Transformation Program 79
4.6 Program Synthesis 83
4.6.1 Identify Source Candidates 84
4.6.2 Token Alignment 87

4.6.3 Program Synthesis using Token Alignment Result . 92
4.6.4 Limitations and Program Repair 94
4.7 Experiments 96
4.7.1 Experimental Setup 96
4.7.2 User Study on Verification Effort 97
4.7.3 User Study on Explainability 101
4.7.4 Expressivity and Efficiency Tests 105
4.8 Related Work 109
4.9 Conclusion 112

Chapter 5. Synthesis of Complex Schema Mapping Queries using "Multiresolution" Examples 114
5.1 Introduction 114
5.2 Motivating Example 118
5.3 Problem Statement 120
5.4 Query Synthesis Algorithms 122
5.4.1 Preliminaries 123
5.4.2 Filter Scheduling 128
5.5 Risks and Limitations 137
5.6 Experiments 139
5.6.1 Experimental Setup 140
5.6.2 Overhead of Multiresolution Constraints 143
5.6.3 Filter Validation Efficiency 145
5.6.4 User Validation Effort 149
5.7 Related Work 152
5.8 Conclusion 154

Chapter 6. Conclusions and Future Work 156
6.1 Future Work 157

Bibliography 159

List of Figures

1.1 An example of business contact data 2
1.2 An example Python script transforming Figure 1.1 3
1.3 Figure 1.1 after transformation 3
1.4 An example pivot table of sales data and its relational form 5

3.1 A spreadsheet of business contact information 20
3.2 A relational form of Figure 3.1 20
3.3 Intermediate table state 23
3.4 Perform Unfold before Fill 23
3.5 Program created with WRANGLER 24
3.6 Program synthesized with FOOFAH 24
3.7 Pre-Split data 28
3.8 After Split on ':' 28
3.9 An example data transformation task 39
3.10 Cell-level edit operations composing the transformation from Table e_i to Table e_o in Figure 3.9 (circles are cells, bold arrows are Transforms, dashed arrows are Deletes) 39
3.11 (a) and (b) show number of records and synthesis time required by FOOFAH in the experiments of Section 3.5.1; (c) Percentage of successes for different search strategies in the experiments of Section 3.5.3. 50
3.12 (a) Percentage of tests synthesized in $\leq$ Y seconds using different search strategies; (b) Percentage of tests synthesized in $\leq$ Y seconds with different pruning rules settings; (c) Percentage of tests synthesized in $\leq$ Y seconds adding Wrap variants. 52

4.1 Phone numbers with diverse formats 68
4.2 Patterns after transformation 68
4.3 Pattern clusters of raw data 68
4.4 Suggested data transformation operations 68
4.5 "CLX" Model: Cluster–Label–Transform 72
4.6 Hierarchical clusters of data patterns 78
4.7 UniFi Language Definition 80
4.8 Preview Tab 84
4.9 Combine Extracts 84
4.10 Token alignment for the target pattern $\mathcal{T}$ 84
4.11 Scalability of the system usability as data volume and heterogeneity increases (shorter bars are better) 99
4.12 Verification time (shorter bars are better) 100
4.13 User comprehension test (taller bars are better) 100
4.14 Completion time (shorter bars are better) 100

5.1 Desired schema mapping (SQL-)query generating the target schema in Table 5.1 115
5.2 Examples at various resolutions to describe the target schema 117
5.3 Multiresolution constraint for a single cell 122
5.4 An example of a filter 127
5.5 Possible distribution of the filter query size 136
5.6 Number of candidate queries to be validated at the presence of multiresolution constraints ($p \in [2,5], r = 2, s = 50\%, w = 2$) 143
5.7 Number of filters to be validated at the presence of multiresolution constraints ($p \in [2,5], r = 2, s = 50\%, w = 2$) . 144
5.8 Compare BN-Greedy (BN), Naïve-Greedy (Naïve), and Shortest-First (SF) on the number of validations ($p \in [2,5], r = 2, s = 50\%, w = 2$) 146
5.9 Compare BN-Greedy (BN), Naïve-Greedy (Naïve), and Shortest-First (SF) on validation time ($p \in [2,5], r = 2, s = 50\%, w = 2$) 147

5.10 Number of satisfying schema mapping queries found when the size of user input increases ($r \in [2, 5], s = 50\%, p = 4, w = 2$) and when the amount of low-resolution constraints in the user input increases (absolute $s \in [0, 8], r = 4, p = 4, w = 2$) 150

List of Tables

3.1 Frequently used notation 26
3.2 Data transformation operators used by Foofah 27
3.3 Table Edit Operators 35
3.4 Geometric patterns 43
3.5 User study experiment results 55
3.6 Success rates for different techniques on both layout transformation and syntactic transformation benchmarks 57

4.1 Frequently used notations 70
4.2 Token classes and their descriptions 72
4.3 Normalizing messy medical billing codes 81
4.4 Normalizing messy employee names 82
4.5 Explainability test cases details 101
4.6 Benchmark test cases details 105
4.7 User effort simulation comparison. 108

5.1 Desired target schema 115
5.2 Dataset Details 140

Abstract

The majority of real-world data we can access today have one thing in common: they are not immediately usable in their original state. Trapped in a swamp of data usability issues like non-standard data formats and heterogeneous data sources, most data analysts and machine learning practitioners have to burden themselves with "data janitor" work, writing ad-hoc Python, PERL or SQL scripts, which is tedious and inefficient. It is estimated that data scientists or analysts typically spend 80% of their time in preparing data, a significant amount of human effort that can be redirected to better goals. In this dissertation, we accomplish this task by harnessing knowledge such as examples and other useful hints from the end user. We develop program synthesis techniques guided by heuristics and machine learning, which effectively make data preparation less painful and more efficient to perform by data users, particularly those with little to no programming experience.

Data transformation, also called data wrangling or data munging, is an important task in data preparation, seeking to convert data from one format to a different (often more structured) format. Our system FOOFAH shows that allowing end users to describe their desired transformation, through providing small input-output transformation examples, can significantly reduce the overall user effort. The underlying program synthesizer can often succeed in finding meaningful data transformation programs within a reasonably short amount of time. Our second system, CLX, demonstrates that

sometimes the user does not even need to provide complete input-output examples, but only label ones that are desirable if they exist in the original dataset. The system is still capable of suggesting reasonable and explainable transformation operations to fix the non-standard data format issue in a dataset full of heterogeneous data with varied formats. PRISM, our third system, targets a data preparation task of data integration, i.e., combining multiple relations to formulate a desired schema. PRISM allows the user to describe the target schema using not only high-resolution (precise) constraints of complete example data records in the target schema, but also (imprecise) constraints of varied resolutions, such as incomplete data record examples with missing values, value ranges, or multiple possible values in each element (cell), so as to require less familiarity of the database contents from the end user.

Chapter 1

Introduction

The majority of real-world data have one thing in common: they are not easily usable in their raw form. Ideally, data are expected to be clean and in a well-structured data frame or relational form, without missing values and peculiar data entries. People should be able to immediately perform regression analysis or train machine learning/deep learning models on such data. But the sad fact is, these data professionals are often trapped by a swamp of hairy data quality and usability issues like non-standard data formats, missing values, outliers, and heterogeneous data sources.

For example, Figure 1.1 is an example of data in an ad-hoc format. Despite being comprehensive, Figure 1.1 is still not ready for data analysis in the following ways.

- Column headers are in the second row rather than the first one.
- Some data values are missing in the first column.
- It uses empty spaces as record breakers.
- Data values in the second column contain secondary column header information.

Bureau of I.A.	
Regional Director	Numbers
Niles C.	Tel: (800)645-8397
	Fax: (907)586-7252
Jean H.	Tel: (918)781-4600
	Fax: (918)781-4604
Frank K.	Tel: (615)564-6500
	Fax: (615)564-6701

...

Figure 1.1: An example of business contact data

This data is in a non-standard form and difficult for downstream statistics and machine learning packages to consume in its original form.

Data preparation denotes a compilation of solutions and strategies that encompass a variety of data quality issues. The tasks within data preparation are usually classified into the following categories [39]:

- **Exploration**. Discovering and creating extra information from the given target data through data profiling and visualization. The additional information helps the user understand the target data.
- **Structuring**. Transforming the structure of data into a shape desired by the downstream analytics software. Pivot tables and relational tables are two common shapes that are desired.
- **Cleaning**. Detecting and fixing the data errors (e.g., inconsistency, outliers, and missing values) at the cell level.
- **Integration**. Creating an enriched dataset by combining data from multiple sources.

```python
from Operations import *
# We hide the details of loading raw data for presentation purpose
    , and assume t is the loaded raw table represented by 2d list
    in python.

# split Column 1 on ':'
t = split(t, 1, ':')
# delete rows where Column 2 is null
t = delete(t, 2)
# fill Column 0 with value above
t = fill(t, 0)
# Unfold on Column 1
t = unfold(t, 1)
```

Figure 1.2: An example Python script transforming Figure 1.1

Name	Tel	Fax
Niles C.	(800)645-8397	(907)586-7252
Jean H.	(918)781-4600	(918)781-4604
Frank K.	(615)564-6500	(615)564-6701

...

Figure 1.3: Figure 1.1 after transformation

Most data analysts often have to take the role of “data janitor” and resolve these data quality issues by themselves. To fix these data quality issues, data users often need to compose some Python, PERL or SQL scripts. For example, Figure 1.2 is a simple Python script to transform the ad-hoc dataset in Figure 1.1 into a more well-structured form in Figure 1.3.

Hand-writing these data preparation scripts is tedious and painful as it requires good programming skills and considerable patience from data users. Even worse, most of these scripts are customized to ad-hoc data and difficult to maintain or reused in a new data source.

Large business groups and organizations may afford to hire a group of IT professionals or BI engineers to take care of the data preparation chores to free up their data analysts. However, two groups of people—data analysts (know the data) and IT professionals (prepare the data)—often need to go through several interaction cycles before data analysts finally get the desired data and kick off their work. In a large organization, this could go back and forth for days to even weeks.

In either case, we see there is a huge waste of time and energy for these valued professionals: it is estimated that data scientists or analysts typically spend 80% of their time in preparing data [79]. They can be otherwise more productive in analyzing data if data preparation could require less programming skills and user effort.

In this dissertation, we argue that building a program synthesis system that can leverage the user knowledge in a less effort-consuming fashion and generate desirable programs for the data user is the key to addressing the above problem. We demonstrate this idea with several systems for varied tasks in the domain of data preparation. First, we detail the challenges in data preparation for human users and challenges we face as researchers in building such systems (Section 1.1). Next, we give a brief summary of our contributions in this dissertation (Section 1.2).

1.1 Challenges and Opportunities

As discussed previously, data preparation is a critical, but also painstaking, part of real-world data analysis workflow. People usually spend a large share of their time in preparing the required data either manually or through writing small, ad-hoc program scripts. We elaborate on the difficulties data users may face in data preparation. Also, we discuss why building user-friendly data preparation systems is non-trivial and some intuitive solutions.

1.1.1 Challenges for users

For data users, especially non-expert users, performing data preparation by themselves can be difficult in many ways.

Lack of programming skills — A survey of 719 participants conducted by KDNuggests in 2014 indicates R and SAS are the top two programming languages most frequently used for data analysis and data science [81]. Although, certain packages in R or SAS, like `stringr` and `lubridate`, offer functions to perform some data cleaning tasks like trimming whitespaces in a string or datetime format conversion, many real-world data preparation or cleaning tasks cannot be complete without writing scripts in other programming languages like Python, PERL or SQL. Extracting a relational table Table 1.3 from Table 1.1 is one such example. Yet, many data analysts and statisticians may have little to no experience in these languages, which means they either take baby steps to learn new programming languages or ask more experienced programmers for help. Program synthesis seems to be a promising trajectory we should follow to address this issue.

Region	2015	2016
East	2300	2453
West	9866	8822
Midwest	2541	2575

Region	Year	Sale
East	2015	2300
East	2016	2453
West	2015	9866
West	2016	8822
Midwest	2015	2541
Midwest	2016	2575

Figure 1.4: An example pivot table of sales data and its relational form

Lack of domain knowledge — Knowing how to program is fundamental but not sufficient. To clean data, the user should also be familiar with transformation operations. Once the data user figures out the desired form of the data, she must be able to quickly come up with a transformation plan. Transforming between a pivot table and a relational table can serve as a good example to illustrate the complexity of the logic behind such a transformation. The table on the left of Figure 1.4 is a textbook pivot table of sales data from [87]. We can see that the second column and the third column both represent the same type of data. To "unpivot" this table, i.e., convert it into a relational form, one needs to 1) identify which columns contain sales data, i.e., Years, 2) create a new column using the cross product of Year values and Region values, and 3) map the sale data values to the corresponding Region and Year value pairs. In the new relational table (on the right of Figure 1.4), each record is a unique combination of region, year and sale data. For those data users unfamiliar with the "unpivot" operation, even writing a Python script to support this transformation can require some deep thinking. Instead of asking the correct operations to use, a data preparation system should be able to use hints of other forms, such as examples, from the end users who are not familiar with data preparation when synthesizing programs.

Lack of data familiarity — Suppose a data user is both a skilled programmer and an expert in data cleaning. Data preparation can still be challenging as the user may need to have deep knowledge of the target data she wants to clean. For example, in a relational dataset, the user needs to first identify all existing quality issues, such as typos, missing values and incorrect string patterns, before writing any program to fix them. When the data is large-scale or messy, such problems may be buried deep in the dataset, and manually scanning through the dataset to identify these problems can be demanding. Intuitively, automatically discovering and presenting useful metadata information about the target data should be helpful for users to be familiar with the data. For example, suppose the task is to convert a set of dates into the format of "`MM/DD/YYYY`", it is useful to first inform the user what the existing date formats in the dataset are.

1.1.2 Technical challenges

Facing above issues, data users both experienced and inexperienced are unlikely to prepare data in a simple and efficient manner. Recently, techniques including "Programming by Example" and "predictive interaction" have been adopted to lower the bar of data preparation for data users. However, previous solutions usually identified and addressed one of the above issues leaving the rest untouched. In this dissertation, we target developing human-in-the-loop (interactive) data preparation systems with all following desiderata: 1) capable of automatically generating programs, 2) able to harness user knowledge and hints in varied forms, 3) does not require the user to be familiar with the target data to provide hints.

Building such a system poses several key technical challenges on the system side detailed below.

Usability — Similar to other interactive systems, a critical quality of our system is usability—how easy it is for target users to interact with the system. One critical aspect of usability in a system is efficiency; short average waiting time in between interaction cycles is key to the system usability. Designing efficient data preparation program synthesis algorithms is clearly a major challenge for us here. On the other hand, as the target user can be a non-expert in data preparation, another usability barrier can be associated with user input. The challenge then is to explore novel forms of user input leveraging the user knowledge because the end user is assumed to be a non-expert and thereby unlikely to provide precise information that is immediately useful for the system to identify the correct (transformation-) operation. Unlike the former challenge, the usability of the input side is less straightforward to evaluate in practice. In the following sections, it will be measured by several methods including mouse/keyboard clicks, number of examples, timespan, etc.

Coverage — Besides being interactive, a successful system in our problem setting should also be able to cover a wide spectrum of particular tasks within a sub-domain in data preparation so that it can be practical in solving real-world problems. Further, it is favorable to researchers if the system can quickly adapt when input data or workflow changes (with minimum extra engineering effort). Otherwise, researchers will have to go through the hassle of redesigning the critical components of the system again.

Scalability — A typical concern for a system is that when the amount or quality of user hints changes, how well the system can scale, i.e., maintain its efficiency. Most program synthesis problems can be seen as a search problem in a large program space. The changes in user input may drastically increase the difficulty of the search problem which affects the system efficiency. Even worse, not only the user input but also the complexity, like the length, of the desired output program may similarly affect the system efficiency. The concern of scalability arises not only at the system level but also at the user level. For example, if the target data to clean becomes larger and more heterogeneous, the amount of user input may increase accordingly, which hurts the system usability in return.

1.2 Summary of Contributions

In this section, we introduce three systems we have constructed over the years as our main contributions in this dissertation to 1) alleviating the difficulties end users face when performing data preparation, and 2) addressing aforementioned issues in human-in-the-loop data preparation.

The first work we present is Foofah [45, 46]—a system that synthesizes straight-line data transformation programs. To describe the desired transformation, the user can provide a small input-output example made from sampled data records. Our experiments show that Foofah costs 60% less interaction time than the state-of-the-art interactive data transformation system, TrifactaWrangler [52], that takes in procedural hints from the user. Also, the combinatorial-search-based algorithm guided by a novel distance metric, TED, and a combination of pruning rules we propose can efficiently discover desired transformation programs in a large search space. Besides, Foofah can be extensible since evidence suggests that it can support a richer set of transformation operations with minimum tuning effort.

The second system, Clx [49], takes on another important data preparation task—data pattern (format) standardization. In this project, we propose a pattern-based Programming-by-Example data pattern transformation interaction model. Unlike traditional PBE systems for string syntactic transformation, such as FlashFill [29], where end users provide examples at the instance level, our users could simply select a data pattern Clx derives from the input data as the desired pattern. The system will then synthesize a complete set of possible programs converting all non-standard data patterns to the selected pattern. We show sufficient evidence that interaction time is required when end users interact at the pattern level rather than the instance level. With the pattern information, end users are able to quickly identify the data in non-standard formats and alter the suggested transformations. Lastly, any inferred transformation program in our proposed DSL can be easily translated to a set of `RegEx_Replace` operations, which offers an alternative approach to verifying the inferred transformation programs for end users besides checking the program result.

The last system PRISM [47, 48] targets discovering schema mapping queries in data integration in a relational database. Previous PBE schema mapping systems, such as MWEAVER [84] and FILTER [93], ask the end user to provide high-resolution constraints, i.e., complete data records in the target schema. This requirement creates challenges for end users who are not highly familiar with the source dataset. In comparison, PRISM is able to leverage imprecise user knowledge of the source database in schema mapping query synthesis: it supports a richer set of example constraints of varied resolutions, such as incomplete data records with missing values, multiple possible values or a value range in each element (cell). In the algorithm design, we propose to apply Bayesian models in scheduling expensive candidate query validations to speed up the entire schema mapping query search. Our experiment results show that the use of Bayesian models achieves a verification workload reduction of up to $\sim$ 69% and a runtime reduction of up to $\sim$ 30% compared to the baseline strategy used by FILTER.

Chapter 2

Research Background

In this chapter, we give a brief overview of some background information and research work that are related to our work. The discussion is organized around two broad domains: *data preparation* and *program synthesis*.

2.1 Data Preparation Pipeline

The concept of data preparation represents a combination of solutions to a variety of data usability/quality issues. Some key tasks in data preparation include transformation/wrangling, cleaning, integration, profiling and visualization/reporting [39, 51]. In this dissertation, we mainly focus on the first four task domains.

2.1.1 Data Transformation

Data transformation, also called data wrangling or data munging, is to convert data stored in different formats into a uniform and desirable format for ease of access and decision making [86]. There are three particular classes of tasks: syntactic transformations, layout (structure) transformations, and semantic transformations [30].

Syntactic transformations manipulate data strings at the cell level often through regular expressions. The goal is typically to transform cell data from one pattern into another (e.g., transforming phone number format from "123.456.7890" into "(123) 456-7890"). Previous work on syntactic transformation include IPBE [107], FlashFill [29] and BlinkFill [94]. FlashFill [29] (now a feature in Excel) is an influential work for syntactic transformation. 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. Similarly, similar to FlashFill, our second system, Clx, also targets synthesizing syntactic transformation programs. **Layout transformations**, or **structure transformations**, mainly relocate the cells in a tabular form. The example in Figure 1.4 from Chapter 1 converting a pivot table to a relational table is an illustration of such transformations. Previous research targeting structure transformation includes ProgFromEx [35] and Wrangler [52]. Our first work, Foofah, covers a wider range of transformation tasks than FlashFill, BlinkFill, IPBE and ProgFromEx. FlashFill, BlinkFill and IPBE use the same domain specific language which manipulates strings within cells (strings) in a single column, whereas ProgFromEx only rearranges the cell locations within a spreadsheet (tables) without changing the cell values. In comparison, Foofah is more expressive as it includes both layout and syntactic transformations, same as Wrangler. **Semantic transformations** also transform data at the cell level. Differing from syntactic transformations, semantic transformations manipulate a string not as a sequence of characters but based on its semantic meaning within a semantic type. An example of semantic transformation is transforming “March” to “03”. Although none of our projects in this dissertation attempts to tackle semantic transformations, it is still a critical transformation useful in scenarios like data integration [111] and offered by many commercial data curation tools like Tamr and Trifacta Wrangler. Unlike certain semantic transformations, such as converting `mile` to `km`, that can be performed by using pre-defined formulas, a large set of transformations can be quite ad-hoc or large to be hard-coded. Previous research including DataXFormer [2, 3] and work by Singh et al. [95] have attempted to automatically learn such transformations from web tables or the dataset itself, and suggest them in real time.

A large portion of previous work [7, 29, 30, 35, 57, 94, 107] in data transformation have used Programming-by-Example (PBE) as their interaction interface. In classic PBE, users typically describe their intents through positive examples, and optionally negative examples. Compared to previous work like FlashFill, Clx allows the user to provide positive examples at the pattern level (i.e., regular expressions) rather than cell level. Our user study shows that Clx on average requires 66.3% less user interaction time than FlashFill. When the data size grew by a factor of 30, the user verification time required by Clx grew by 1.3× whereas that required by FlashFill grew by 11.4×, which is evidence that the usability of PBE at the pattern level scales better than PBE at the cell level. Same as ProgFromEx [35] and FlashRelate [7], our first system, Foofah, uses the classic PBE model in transforming spreadsheet data. However, as mentioned, Foofah is more expressive than the other two systems.

Another thread of seminal research including [86], Wrangler [52] and Trifacta created by Hellerstein et al. follow a different interaction paradigm called “predictive interaction”. They proposed an inference-enhanced visual platform supporting various data wrangling and profiling tasks. Based on the user selection of columns, rows or text, the system intelligently suggests possible data transformation operations, such as Split, Fold, or pattern-based extraction operations. Our first project, FOOFAH, offers the same set of transformations as WRANGLER but with a PBE interaction model. Based on our user study result, FOOFAH requires $\sim$ 60% less interaction time than WRANGLER, which suggests it can conserve user effort. A more recent PBE project, TDE [37], also targets data transformation. It can practically solve many real-world data transformation tasks because it supports a wide range of data transformation operators crawled online and can express both syntactic and semantic transformations.

Another domain of work, data extraction, is similar to data transformation. Data extraction seeks to extract data from unstructured or semi-structured data. Various data extraction tools and synthesizers have been created to automate this process: TextRunner [6] and WebTables [16] extract relational data from web pages; Senbazuru [17, 18] and FlashRelate [7] extract relations from spreadsheets; FlashExtract [57] extracts data from a broader range of documents including text files, web pages, and spreadsheets, based on examples provided by the user.

2.1.2 Data Pattern Profiling

Data profiling is the discovery of the metadata of an unknown dataset or database [1]. Some of the metadata that are commonly profiled in a dataset include number of missing values, distinct values, and redundancies. Researchers also have made progress in profiling foreign keys [91], functional dependencies [41, 110] and inclusion dependencies [8, 66] in a relational database. In our second work, CLX, we focus on clustering ad hoc string data based on structures and derive the structure information. The LEARN-

PADS [25] project is somewhat related. It presents 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 [24]) is hard for a human to read, whereas our pattern cluster hierarchy is simpler to understand. Most recently, Datamaran[26] has proposed methodologies for discovering structure information in a data set whose record boundaries are unknown, but for the same reasons as LearnPADS, Datamaran is not suitable for our problem in the second work. FlashProfile [76] is a more recent pattern profiling system that is more align with our proposal in Clx. Same as Clx, it targets a messy dataset with varied formats. Also, the returned patterns are a set of simple data patterns, which is more readable than on one single gigantic and complex pattern like ones offered by PADS. FlashProfile proposes a two-phase approach – "clustering" and "profiling" – to return a fixed number (specified by the user) of patterns representing all data entries in the dataset. However, this approach is not quite applicable in our case. First, FlashProfile requires the number of returned patterns/clusters k be pre-determined by the end user, whereas we believe asking the user to know k for an unfamiliar dataset can be non-trivial in our case. Second, the patterns discovered by FlashProfile are mostly for understanding purposes. Since all patterns returned by FlashProfile are fixed, it is not as clear how to use these patterns in synthesizing transformation programs afterwards as if they were subject to change.

2.1.3 Data Integration and Schema Mapping

Data integration seeks to combine data from different sources, and present a unified view of these data to the user [58]. Schema mapping is to discover a query (or a set of queries) that transform the source data into the target schema [70] and is fundamental to the data integration problem [10].

In the context of DBMS, schema mapping usually involves creating SQL queries with joins. Composing SQL queries for schema mapping is known to be non-trivial and burdensome for both professionals and naïve end users. Researchers from both academia and industry have made attempts to facilitate this process for end users in the past two decades. These developed techniques can be generally categorized into two classes based on the interaction model: *schema-driven* and *sample-driven*. IBM Clio [83], Microsoft BizTalk [104] are notable schema mapping systems that support the schema-driven model, which requires hints of possible matching relations and columns from end users. Another thread of research projects [12, 50, 72, 84, 93, 105] focused on supporting a sample-driven model in human-in-the-loop schema mappings. Instead of soliciting matching hints, these systems only ask the user for a few data records in the target schema. Although knowing a few data records can be relatively easier than knowing the database schemas if the user is not familiar with the source database, the sample-driven schema mapping systems can be still impractical if the user's knowledge of the data samples is imprecise and coarse. Our third project, PRISM, targets at leveraging coarse or inaccurate user knowledge of target data in schema mapping.

2.2 Program Synthesis

Program synthesis is the task to discover programs matching the user intent. It has garnered 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.

Program synthesis techniques have been applied to a variety of problem domains: parsers [59], regular expressions [11], bit-manipulation programs [31, 44], data structures [97]; code snippets and suggestions in IDEs [67, 88], and SQL query based on natural language queries [60] and data handling logic [20], schema mappings [4]. There are also several projects that synthesize data transformation and extraction programs, discussed in more detail next.

Four main categories of algorithmic approaches have been proposed for program synthesis: logic-solver-based (constraint-based) approach, sketching, version space algebra, and search-based approach. Gulwani et al. proposed a *logic-solver-based* program synthesis technique to synthesize loop-free bit-manipulation programs [32, 44] using logic solvers, like the SMT solver. Solar-Lezama's work with sketching [98] attempts to formulate certain types of program automatically through clever formulation of SAT solving methods. This approach focuses on programs that are "difficult and important" for humans to write by hand, such for thread locking or decoding compressed data streams. Version space algebra requires a complete search space of programs between two states, which make it more suitable for a Programming By Demonstration problem where the user explicitly provides intermediate states and the search space between these states is small [63] or for PBE problems that can be easily divided into independent sub-problems [29].

For most of our work, the first three approaches do not fit. Logic-solver-based program synthesis does not fit because existing logic solvers could not scale to solve a large number of constraints quadratic in the input size. Sketching [98] is computationally infeasible for interactive data transformation. Version space algebra [29, 55] is usually applied in Programming-by-Demonstration systems.

In this dissertation, we formulate most of our problems as a search problem in the state space graph and solve it using a combinatorial-search-based algorithm. Other program synthesis projects using the search-based approach include [59, 68, 80, 88].

Chapter 3

Synthesizing Data Transformation Programs using User Examples

3.1 Introduction

The many domains that depend on data for decision making have at least one thing in common: raw data is often in a non-relational or poorly structured form, possibly with extraneous information, and cannot be directly used by a downstream information system, like a database or visualization system. Figure 3.1 from [33] is a good example of such raw data. In modern data analytics, data transformation (or data wrangling) is usually a crucial first step that reorganizes raw data into a more desirable format that can be easily consumed by other systems. Figure 3.2 showcases a relational form obtained by transforming Figure 3.1.

Traditionally, domain experts handwrite task specific scripts to transform unstructured data—a task that is often labor-intensive and tedious. The

Bureau of I.A.	
Regional Director	Numbers
Niles C.	Tel: (800)645-8397
	Fax: (907)586-7252
Jean H.	Tel: (918)781-4600
	Fax: (918)781-4604
Frank K.	Tel: (615)564-6500
	Fax: (615)564-6701

...

Figure 3.1: A spreadsheet of business contact information

	Tel	Fax
Niles C.	(800)645-8397	(907)586-7252
Jean H.	(918)781-4600	(918)781-4604
Frank K.	(615)564-6500	(615)564-6701

...

Figure 3.2: A relational form of Figure 3.1

requirement for programming hamstrings data users that are capable analysts but have limited coding skills. Even worse, these scripts are tailored to particular data sources and cannot adapt when new sources are acquired. People normally spend more time preparing data than analyzing it; up to 80% of a data scientist's time can be spent on transforming data into a usable state [65].

Recent research into automated and assisted data transformation systems have tried to reduce the need of a programming background for users, with some success [42, 52, 99]. These tools help users generate reusable data transformation programs, but they still require users to know which data transformation operations are needed and in what order they should be applied. Current tools still require some level of imperative programming, placing a significant burden on data users. Take Wrangler [52], for example, where a user must select the correct operators and parameters to complete a data transformation task. This is often challenging if the user has no experience in data transformation or programming.

Existing data transformation tools are difficult to use mainly due to two usability issues based on our observation in the user study:

- *High Skill*: Users must be familiar with the often complicated transformation operations and then decide which operations to use and in what order.
- *High Effort*: The amount of user effort, including interaction time and user input, increases as the data transformation program gets lengthy.

To resolve the above usability issues, we envision a data transformation program synthesizer that can be successfully used by people without a programming background and that requires minimal user effort. Unlike WRANGLER, which asks the user for procedural hints, this system should allow the user to specify a desired transformation simply by providing an input-output example: the user only needs to know how to *describe the transformed data*, as opposed to knowing any particular transformation operation that must be performed.

Our Approach — In this project, we solve the data transformation program synthesis problem using a Programming By Example (PBE) approach. Our proposed technique aims to help an unsophisticated user easily generate a quality data transformation program using purely input-output examples. The synthesized program is designed to be easy-to-understand (it is a straight-line program composed of simple primitives), so an unsophisticated user can understand the semantics of the program and validate it. Because it is often infeasible to examine and approve a very large transformed dataset synthesizing a readable transformation program is preferred over performing an opaque transformation.

We model program synthesis as a search problem in a state space graph and use a heuristic search approach based on the classic A* algorithm to synthesize the program. A major challenge in applying A* to program synthesis is to create a heuristic function estimating the cost of any proposed partial solution. Unlike robotic path planning, where a metric like Euclidean distance naturally serves as a good heuristic function, there is no straightforward heuristic for data transformation. In this work, we define an effective A* heuristic for data transformation, as well as lossless pruning rules that significantly reduce the size of the search space. We have implemented our methods in a prototype data transformation program synthesizer called FOOFAH.

Organization — After motivating our problem with an example in Section 3.2 and formally defining the problem in Section 3.3, we discuss the following contributions:

- We present a PBE data transformation program synthesis technique backed by an efficient heuristic-search-based algorithm inspired by the A* algorithm. It has a novel, operator-independent heuristic, Table Edit Distance Batch, along with pruning rules designed specifically for data transformation (Section 3.4).
- We prototype our method in a system, FOOFAH, and evaluate it with a comprehensive set of benchmark test scenarios that show it is both effective and efficient in synthesizing data transformation programs. We also present a user study that shows FOOFAH requires about 60% less user effort than WRANGLER(Section 3.5).

We finish with a discussion of future work in Section 3.7

Niles C.	Tel	(800)645-8397
	Fax	(907)586-7252
Jean H.	Tel	(918)781-4600
	Fax	(918)781-4604
Frank K.	Tel	(615)564-6500
	Fax	(615)564-6701

Figure 3.3: Intermediate table state

	Tel	Fax
Niles C.	(800)645-8397	
		(615)564-6701
Jean H.	(918)781-4600	
Frank K.	(615)564-6500	

Figure 3.4: Perform Unfold before Fill

3.2 Motivating Example

Data transformation can be a tedious task involving the application of complex operations that may be difficult for a naïve user to understand, as illustrated by the following simple but realistic scenario:

Example 3.1. *Bob wants to load a spreadsheet of business contact information (Figure 3.1) into a database system. Unfortunately, the raw data cannot be loaded in its original format, so Bob hopes to transform it into a relational format (Figure 3.2). Manually transforming the data record-by-record would be tedious and error-prone, so he uses the interactive data cleaning tool* Wrangler *[52].*

Bob first removes the rows of irrelevant data (rows 1 and 2) and empty rows (rows 5, 8, and more). He then splits the cells containing phone numbers on ":", extracting the phone numbers into a new column. Now that almost all the cells from the desired table exist in the intermediate table (Figure 3.3), Bob intends to perform a cross-tabulation operation that tabulates phone numbers of each category against the human names. He looks through Wrangler*'s provided operations and finally decides that* Unfold *should be used. But* Unfold *does not transform the intermediate table correctly, since there are missing values in the column of names, resulting in "null" being the unique identifier for all rows without a human name*

```
1 Delete row 1
2 Delete row 2
3 Delete rows where column 2 is null
4 Split column 2 on ':'
5 Fill split with values from above
6 Unfold column 2 on column 3
```

Figure 3.5: Program created with Wrangler

```
t = split(t, 1, ':')
t = delete(t, 2)
t = fill(t, 0)
t = unfold(t, 1)
```

Figure 3.6: Program synthesized with Foofah

(Figure 3.4). Bob backtracks and performs a Fill *operation to fill in the empty cells with the appropriate names before finally performing the* Unfold *operation. The final data transformation program is shown in Figure 3.5.*

The usability issues described in Section 3.1 have occurred in this example. Lines 1–3 in Figure 3.5 are repetitive as it consists of three Delete operations (*High Effort*). Lines 5–6 require a good understanding of the Unfold operation, which can be difficult for naïve users (*High Skill*). Note that Deletes in Lines 1–2 are different from the Delete in Line 3 in that the latter could apply to the entire file. Non-savvy users may find such conditional usage of Delete difficult to discover, further illustrating the *High Skill* issue.

Consider another scenario where the same task becomes much easier for Bob, our data analyst:

Example 3.2. *Bob decides to use an alternative data transformation system,* FOOFAH. *To use* FOOFAH, *Bob simply needs to choose a small sample of the raw data (Figure 3.1) and describe what this sample should be after being transformed (Figure 3.2).* FOOFAH *automatically infers the data transformation program in Figure 3.6 (which is semantically the same as Figure 3.5, and even more succinct). Bob takes this inferred program and executes it on the entire raw dataset and finds that raw data are transformed exactly as desired.*

The motivating example above gives an idea of the real-world data transformation tasks our proposed technique is designed to address. In general, we aim to transform a poorly-structured grid of values (e.g., a spreadsheet table) to a relational table with coherent rows and columns. Such a transformation can be a combination of the following chores:

1. changing the structure of the table
2. removing unnecessary data fields
3. filling in missing values
4. extracting values from cells
5. creating new cell values out of several cell values

We assume that the input data should be transformed without any extra semantic information, so, for example, transforming "NY" to "New York" is not possible (previous projects [3, 19, 95] have addressed such semantic transformations). Transformations should not add new information that is not in the input table, such as adding a column header.

Notation	Description
$\mathcal{P} = \{p_1, \ldots, p_n\}$	Data transformation program
$p_i = (op_i, par_1, \ldots)$	Transformation operation with operator op_i and parameters par_1, par_2, etc.
$\mathcal{R}$	Raw dataset to be transformed
$e_i \in \mathcal{R}$	Example input sampled from $\mathcal{R}$ by user
$e_o = \mathcal{P}(e_i)$	Example output provided by user, transformed from e_i
$\mathcal{E} = (e_i, e_o)$	Input-output example table pair, provided as input to the system by user

Table 3.1: Frequently used notation

3.3 Problem Definition

To help the user synthesize a correct data transformation program, we take a Programming By Example (PBE) approach: the user provides an input-output example pair set, made out of a subset of the input data, and the system generates a program satisfying the example pairs and hopefully can transform the entire input data correctly.

3.3.1 Problem Definition

With all notations summarized in Table 3.1, we define this problem formally:

Problem *Given a user's set of input-output examples $\mathcal{E} = (e_i, e_o)$, where e_i is drawn from raw dataset $\mathcal{R}$ and e_o is the desired transformed form of e_i, synthesize a data transformation program $\mathcal{P}$, parameterized with a library of data transformation operators, that will transform e_i to e_o.*

Like previous work in data transformation [35, 52], we assume the raw data $\mathcal{R}$ is a grid of values. $\mathcal{R}$ might not be relational but must have some regular structure (and thus may have been programmatically generated). Further, $\mathcal{R}$ may contain schematic information (e.g., column or row headers)

Operator	Description
Drop	Deletes a column in the table
Move	Relocates a column from one position to another in the table
Copy	Duplicates a column and append the copied column to the end of the table
Merge	Concatenates two columns and append the merged column to the end of the table
Split	Separates a column into two or more halves at the occurrences of the delimiter
Fold	Collapses all columns after a specific column into one column in the output table
Unfold	"Unflatten" tables and move information from data values to column names
Fill	Fill empty cells with the value from above
Divide	Divide is used to divide one column into two columns based on some predicate
Delete	Delete rows or columns that match a given predicate
Extract	Extract first match of a given regular expression each cell of a designated column
Transpose	Transpose the rows and columns of the table
Wrap (added)	Concatenate multiple rows conditionally

Table 3.2: Data transformation operators used by Foofah

as table values, and even some extraneous information (e.g., "Bureau of I.A." in Figure 3.1).

Once the raw data and the desired transformation meet the above criteria, the user must choose the input sample and specify the corresponding output example. More issues with creating quality input-output examples will be discussed in detail in Section 3.4.5.

3.3.2 Data Transformation Programs

Transforming tabular data into a relational table usually require two types of transformations: *syntactic transformations* and *layout transformations* [30].

Numbers
Tel:(800)645-8397
Fax:(907)586-7252

Figure 3.7: Pre-Split data

Numbers	
Tel	(800)645-8397
Fax	(907)586-7252

Figure 3.8: After Split on ':'

Syntactic transformations reformat cell contents (e.g., split a cell of "mm/dd/yyyy" into three cells containing month, day, year). Layout transformations do not modify cell contents, but instead change how the cells are arranged in the table (e.g., relocating cells containing month information to be column headers).

We find that the data transformation operators shown in Table 3.2 (defined in Potter's Wheel project [85, 86] and used by state-of-art data transformation tool WRANGLER [52]) are expressive enough to describe these two types of transformations. We use these operations in FOOFAH: operators like Split and Merge are syntactic transformations and operators like Fold, and Unfold are layout transformations. To illustrate the type of operations in our library, consider Split. When applying Split parameterized by ':' to the data in Figure 3.7, we get Figure 3.8 as the output. Detailed definitions for each operator are shown in [46].

Our proposed technique is not limited to supporting Potter's Wheel operations; users are able to add new operators as needed to improve the expressiveness of the program synthesis system. We assume that new operators will match our system's focus on syntactic and layout transformations (as described in Section 3.2); if an operator attempts a semantic transformation, our system may not correctly synthesize programs that use it. As we describe below, the synthesized programs do not contain loops, so novel operators must be useful outside a loop's body.

We have tuned the system to work especially effectively when operators make "conventional" transformations that apply to an entire row or column at a time. If operators were to do otherwise — such as an operator for "Removing the cell values at odd numbered rows in a certain column", or for "Splitting the cell values on Space in cells whose values start with 'Math' " — the system will run more slowly. Experimental results in Section 3.5.5 show evidence that adding operators can enhance the expressiveness of our synthesis technique without hurting efficiency.

Program Structure — All data transformation operators we use take in a whole table and output a new table. A reasonable model for most data transformation tasks is to sequentially transform the original input into a state closer to the output example until we finally reach that goal state. This linear process of data transformation results in a loop-free or straight-line program, a control structure that can express a wide range of computations and applied by many previous data transformation projects [35, 52, 55, 106]. We use the operators mentioned above as base component and loop-free programs as the program control structure. Although the synthesized programs will be without loops, they are still challenging to generate given the sheer exponential search space.

Still, the loop-free program structure could restrict us from synthesizing programs that require an undetermined number of iterations of a data transformation operation, or could lead to verbose programs with "unrolled loops". For example, if the user wants to "Drop column 1 to column $\lfloor k/2 \rfloor$ where k is the number of columns in the table" our system will be unable to synthesize a loop-based implementation and instead will simply repeat Drop many times.

Motivated by the above considerations, we formally define the data transformation to be synthesized as follows:

Definition 3.1 (Data transformation program $\mathcal{P}$)**.** *$\mathcal{P}$ is a loop-free series of operations $(p_1, p_2, ..., p_k)$ such that:* ***1****. Each operation $p_i = (op_i, par_1, ...) : t_{in} \rightarrow t_{out}$. p_i includes operator op_i with corresponding parameter(s) and transforms an input data table t_{in} to an output data table t_{out}.* ***2****. The output of operation p_i is the input of p_{i+1}.*

3.4 Program Synthesis

We formulate data transformation program synthesis as a search problem. Other program synthesis approaches are not efficient enough given the huge search space in our problem setting (Section 3.4.1). We thus propose an efficient heuristic search method, inspired by the classic A* algorithm. In Section 3.4.2, we introduce a straw man heuristic and then present our novel operator-independent heuristic, *Table Edit Distance Batch* (TED Batch), based on a a novel metric, *Table Edit Distance* (TED), which measures the dissimilarity between tables. In addition, we propose a set of pruning rules for data transformation problems to boost search speed (Section 3.4.3). We compare the time complexity of our technique with other previous projects (Section 3.4.4). Finally, we discuss issues about creating examples and validation (Section 3.4.5).

3.4.1 Program Synthesis Techniques

In Section 3.3.2, we described the structure of our desired data transformation program to be component-based and loop-free. Gulwani et al. proposed a *constraint-based* program synthesis technique to synthesize loop-free bit-manipulation programs [32, 44] using logic solvers, like the SMT solver. However, the constraint-based technique is impractical for our interactive

PBE system because the number of constraints dramatically increases as the size of data increases, scaling the problem beyond the capabilities of modern logic solvers.

Other methods for synthesizing component programs include *sketching* and *version space algebra*. Solar-Lezama's work with sketching [98] attempts to formulate certain types of program automatically through clever formulation of SAT solving methods. This approach focuses on programs that are "difficult and important" for humans to write by hand, such for thread locking or decoding compressed data streams, so it is acceptable for the solver to run for long periods. In contrast, our aims to improve productivity on tasks that are "easy but boring" for humans. To preserve interactivity for the user, our system must find a solution quickly.

Version space algebra requires a complete search space of programs between two states, which make it more suitable for a Programming By Demonstration problem where the user explicitly provides intermediate states and the search space between these states is small [63] or for PBE problems that can be easily divided into independent sub-problems [29]. In our problem, the search space of the synthesized programs is exponential, and thus version space algebra is not practical.

Search-based techniques are another common approach used by previous program synthesis projects [59, 68, 80, 88]. For our problem, we formulate program synthesis as a search problem in a *state space graph* defined as follows:

Definition 3.2 (Program synthesis as a search problem)**.** *Given input-output examples $\mathcal{E} = (e_i, e_o)$, we construct a state space graph $G(V, A)$ where arcs A represent candidate data transformation operations, vertices V represent intermediate states of the data as transformed by the operation on previously traversed arcs, e_i is the initial state v_0, and e_o is the goal state*

v_n. Synthesizing a data transformation program is finding a path that is a sequence of operations leading from v_0 to v_n in G.

Graph Construction — To build a state space graph G, we first expand the graph from v_0 by adding out-going edges corresponding to data transformation operators (e.g., Drop, Fold) with all possible parameterizations (parameters and their domains for each operator are defined both in [86] and [46]). The resulting intermediate tables become the vertices in G. Since the domain for all parameters of our operator set is restricted, the number of arcs is still tractable. More importantly, in practice, the pruning rules introduced in Section 3.4.3 trim away many obviously incorrect operations and states, making the actual number of arcs added for each state reasonably small (e.g., the initial state e_i in Figure 3.10 has 15 child states, after 161 are pruned).

If no child of v_0 happens to be the goal state v_n, we recursively expand the most promising child state (evaluated using the method introduced in Section 3.4.2) until we finally reach v_n. When the search terminates, the path from v_0 to v_n is the sequence of operations that comprise the synthesized data transformation program.

3.4.2 Search-based Program Synthesis

Given our problem formulation, what is needed is a search-based program synthesis algorithm, i.e., an algorithm finding a program that is "consistent with all the user-provided examples and fits the syntactic template of the native language" [5]. As the search space is exponential in the number of the operations in the program, searching for a program in a space of this size is non-trivial. *Brute-force* search quickly becomes intractable. As a PBE solution needs to be responsive to preserve interactivity, we are exposed to a challenging search problem with a tight time constraint. Therefore, the major consideration in designing the search algorithm is efficiency (besides correctness). Another less critical consideration is the complexity of the synthesized programs. In our problem, we prefer shorter programs over longer ones, because shorter programs are generally easier to understand than longer ones.

Given the above concerns, we develop a heuristic search algorithm for synthesizing data transformation programs inspired by the classic pathfinding algorithm, the A* algorithm [36] and some of the recent work in program synthesis [59, 88]. To find a path in the graph from the initial state to the goal state, the A* algorithm continually expands the state with the minimum cost $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach state n from the initial state and heuristic function $h(n)$ is the approximate cost of the cheapest path from state n to the goal state. The definition of cost depends on the performance measure of the search task. In robotic pathfinding, the cost is typically distance traveled, an *admissible* heuristic that guarantees the search is the most efficient and the discovered path is also the shortest.

As we discussed, we are concerned more with search efficiency and also the synthesized program complexity in our problem, and hence take the number of operations as the cost. Formally, we define cost as follows:

Definition 3.3 (Data transformation cost)**.** *Given any two states* (v_i, v_j) *in graph* G*, cost is the minimum number of data transformation operations needed to transform* v_i *to* v_j*.*

For simplicity, we treat all operators equally. Although, some operators like Fold might be conceptually more complex for users to understand, we observe that such operators rarely occur more than once in our benchmarks. Assigning different weights for different operators is possible in the current architecture but is not a focus in this project.

Ideally, we want an *admissible* heuristic like that used by robotic pathfinding to guarantee the search efficiency and program readability. However, in our problem setting, designing such a heuristic is almost impossible. Hence, we relax the need for admissibility, accepting less-than-perfect search efficiency and a synthesized program that is slightly longer than the program with the minimal length.

Naïve Heuristic — Possibly the most straightforward heuristic is a rule-based one. The intuition is that we create some rules, based on our domain knowledge, to estimate whether a certain Potter's Wheel operator is needed given $\mathcal{E}$, and use the total count as the final heuristic score in the end. An example heuristic rule for the Split operator is "number of cells from $T_i[k]$ (i.e., the *row* k in T_i) with strings that do not appear fully in $T_o[k]$, but do have substrings that appear in $T_o[k]$." (This is a reasonable rule because the Split operator splits a cell value in the input table into two or more pieces in the output table, as in Figures 3.7 and 3.8.) The details about this naïve heuristic are presented in [46].

Although this naïve heuristic might appear to be effective for our problem, it is weak for two reasons. First, the estimation is likely to be inaccurate when the best program entails layout transformations. Second, the heuristic is defined in terms of the existing operators and will not easily adapt to new operators in the future. We expect different operators to be helpful in different application scenarios and our framework is designed to be operator independent.

To overcome these shortcomings, we have designed a novel heuristic function explicitly for tabular data transformation.

Operator	Description
Add	Add a cell to table
Delete	Remove a cell from table
Move	Move a cell from location (x_1, y_1) to (x_2, y_2)
Transform	Syntactically transform a cell into a new cell

Table 3.3: Table Edit Operators

Table Edit Distance

The purpose of the heuristic function in A* is guiding the search process towards a more promising direction. Inspired by previous research [88], which used edit distance as the heuristic function, we define *Table Edit Distance* (TED), which measures the table dissimilarity:

$$TED(T_1, T_2) = \min_{(p_1,\ldots,p_k)\in P(T_1,T_2)} \sum_{i=i}^{k} cost(p_i) \tag{3.1}$$

TED is the minimum total cost of *table edit operations* needed to transform T_1 to T_2, where $P(T_1, T_2)$ denotes the set of edit paths transforming T_1 to T_2 and $cost(p_i)$ is the cost of each table edit operation p_i. The table edit operations include Add, Delete, Move, Transform (see Table 3.3 for definition).

Inspired by the *graph edit distance* algorithm [74], we designed an algorithm to calculate the exact TED [46]. Unfortunately, computing TED in real time is not practical: it is equivalent to computing graph edit distance, which is NP-complete [27].

We therefore designed an efficient greedy algorithm to approximate TED, shown in Algorithm 1. The idea behind Algorithm 1 is to greedily add the cheapest operations among the candidate operations to formulate each cell

Algorithm 1: Approximate TED Algorithm

```
Data: Intermediate Table $e_x = \{u_1, u_2, ..., u_{|e_x|}\}$, where $u_i$ represents a cell from $e_x$;
      Example Output Table $e_o = \{v_1, v_2, ..., v_{|e_o|}\}$, where $v_i$ represents a cell from $e_o$
Result: cost, edit path
1  $p_{final} \leftarrow \emptyset$;
2  $p_{temp} \leftarrow \emptyset$;
3  for $w$ in $e_x$ do
4      add $AddCandTransform(w, v_1)$ to $p_{temp}$;
5  add Add($v_1$) to $p_{temp}$;
6  $p_{final} \leftarrow argmin_{\forall p \in p_{temp}} cost(p)$ ;
7  Let $\{u_1, ..., u_j\}$ & $\{v_1, ..., v_k\}$ be processed cells;
8  while $j <| e_x |$ and $k <| e_o |$ do
9      $p_{temp} \leftarrow \emptyset$;
10     for $w \in \{u_{j+1}, ..., u_{|e_x|}\}$ do
11         add $AddCandTransform(w, v_{k+1})$ to $p_{temp}$;
12     add Add($v_{k+1}$) to $p_{temp}$;
13     if $cost(argmin_{\forall p \in p_{temp}} cost(p)) \geq \infty$ then
14         $p_{temp} \leftarrow \emptyset$;
15         for $w \in \{u_1, ..., u_{|e_x|}\}$ do
16             add $AddCandTransform(w, v_{k+1})$ to $p_{temp}$;
17         add Add ($v_{k+1}$) to $p_{temp}$;
18     $p_{final} \leftarrow p_{final} \cup argmin_{\forall p \in p_{temp}} cost(p)$ ;
19     Let $\{u_1, ..., u_j\}$ & $\{v_1, ..., v_k\}$ be processed cells;
20 if $j <| e_x |$ then
21     for $w \in \{u_{j+1}, ..., u_{|e_x|}\}$ do
22         add Delete ($w$) to $p_{final}$
23 if $k <| e_o |$ then
24     for $q \in \{v_{k+1}, ..., v_{|e_o|}\}$ do
25         $p_{temp} \leftarrow \emptyset$;
26         for $w \in \{u_1, ..., u_{|e_x|}\}$ do
27             add $AddCandTransform(w, q)$ to $p_{temp}$;
28         add Add ($q$) to $p_{temp}$;
29         $p_{final} \leftarrow p_{final} \cup argmin_{\forall p \in p_{temp}} cost(p)$ ;
30 Return cost($p_{final}$), $p_{final}$
```

in the output table e_o, building up a sequence of edits until we obtain a *complete edit path*. The edit path formulates the entire output table. The final heuristic score is the total cost of this path.

Algorithm 1 consists of three core steps. We use Figure 3.10, which describes the edit path found to transform input table e_i to e_o in Figure 3.9, as an example to explain each step.

Step 3.1. *(lines 3–19) For each unprocessed cell in the output table (picked in row-major order), we choose the cheapest cell-specific operation sequence (a tie is broken by row-major order of the cell from the input table), from one of:*

1. *Transformation from an unprocessed cell in* e_x *into a cell in* e_o*. Transformation sequences for a pair of cells are generated by the function "AddCandTransform" and can include a* Move *operator (if the cell coordinates differ), a* Transform *operator (if the cell contents differ), or both operators (if both conditions apply).*
2. Add *a new cell to* e_o*.*

After picking an operation sequence, we hypothesize an edit path (p_{temp}) *for each cell that consists of all edits made so far, plus the chosen operation. We measure the cost of each edit path using the cost function. By default, all table edit operations have equal cost; however, we assign a cost of infinity to: (1)* Transform *operations between cells with no string containment relationship and (2)* Add *operations for non-empty cells. (These fall into the category of tasks beyond the scope of our technique described in Section 3.2.)*

For example, for $\mathcal{O}_1$ in Figure 3.10[1], Algorithm 1 finds that transforming from $\mathcal{I}_2$ to $\mathcal{O}_1$ to is the best, because the costs of transforming from $\mathcal{I}_1$, $\mathcal{I}_3$,

[1]$\mathcal{O}_n$ means cell n from e_o. $\mathcal{I}_m$ means cell m from e_i.

and $\mathcal{I}_5$ are all infinite (no string containment), and although transforming from $\mathcal{I}_4$ or $\mathcal{I}_6$ costs the same as transforming from $\mathcal{I}_2$, $\mathcal{I}_2$ has a higher row-major order in the input table. For $\mathcal{O}_2$, we find that transforming from any unprocessed cell in the input example (i.e., $\mathcal{I}_{1,3,4,5,6}$) to $\mathcal{O}_2$ yields an infinite cost, so using only the unprocessed cells would not result in a reasonable edit path. We fix this problem by adding transformations from the processed cells in lines 13–18; this helps find the cheapest edit operation to formulate $\mathcal{O}_2$: transforming from $\mathcal{I}_2$ to $\mathcal{O}_2$.

Step 3.2. *(line 20–22) Delete all unprocessed cells from e_x.*

In our running example, after we have discovered edit operations for all cells in the output example, we find that cells 1, 3, and 5 from the input example remain unprocessed. We simply add Delete operations to remove them.

Step 3.3. *(line 23–29) When we have unprocessed cells in e_o, but no remaining unprocessed cells in e_x, our only options are to: (1) Transform from a processed cell in e_x, (we process every input cell at least one time before processing any cell for a second time) OR (2) Add a new cell.*

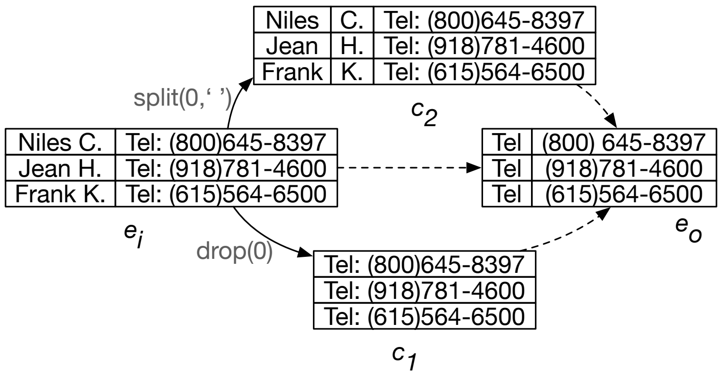


Figure 3.9: An example data transformation task

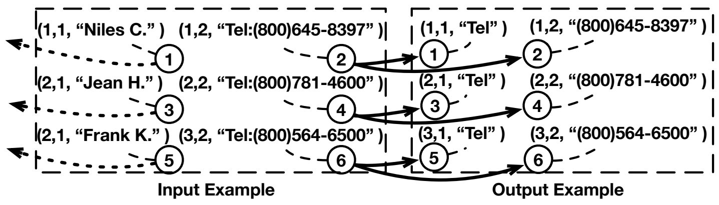


Figure 3.10: Cell-level edit operations composing the transformation from Table e_i to Table e_o in Figure 3.9 (circles are cells, bold arrows are Transforms, dashed arrows are Deletes)

The edit path discovered in Figure 3.10 is as follows:[2]

$$P_0 = \left\{ \begin{array}{cc} \text{Transform((1,2),(1,1))}, & \text{Move((1,2),(1,1))} \\ \text{Transform((1,2),(1,2))}, & \text{Transform((2,2),(2,1))} \\ \text{Move((2,2),(2,1))}, & \text{Transform((2,2),(2,2))} \\ \text{Transform((3,2),(3,1))}, & \text{Move((3,2),(3,1))} \\ \text{Transform((3,2),(3,2))}, & \text{Delete((1,1))} \\ \text{Delete((2,1))}, & \text{Delete((3,1))} \end{array} \right\}$$

[2]Transform((a_1,a_2),(b_1,b_2)) means Transform the cell at (a_1,a_2) in e_i to the cell at (b_1,b_2) in e_o.
Move((a_1,a_2),(b_1,b_2)) means Move the cell from (a_1,a_2) in e_i to (b_1,b_2) in e_o
Delete((c_1,c_2)) means Delete the cell at (c_1,c_2) in e_i.

Figure 3.9 shows a data transformation task, where e_i is the input example and e_o is the output example. c_1 and c_2 are two child states of e_i representing outcomes of two possible candidate operations applied to e_i. Below we define P_1 and P_2, the edit paths discovered by Algorithm 1 for c_1 and c_2:

$$P_1 = \left\{ \begin{array}{cc} \mathsf{Transform((1,1),(1,1))}, & \mathsf{Transform((1,1),(1,2))} \\ \mathsf{Move((1,1),(1,2))}\ , & \mathsf{Transform((2,1),(2,1))} \\ \mathsf{Transform((2,1),(2,2))}, & \mathsf{Move((2,1),(2,2))} \\ \mathsf{Transform((3,1),(3,1))}, & \mathsf{Transform((3,1),(3,2))} \\ \mathsf{Move((3,1),(3,2))} & \end{array} \right\}$$

$$P_2 = \left\{ \begin{array}{cc} \mathsf{Transform((1,3),(1,1))}, & \mathsf{Move((1,3),(1,1))} \\ \mathsf{Transform((1,3),(1,2))}, & \mathsf{Move((1,3),(1,2))} \\ \mathsf{Transform((2,3),(2,1))}, & \mathsf{Move((2,3),(2,1))} \\ \mathsf{Transform((2,3),(2,2))}, & \mathsf{Move((2,3),(2,2))} \\ \mathsf{Transform((3,3),(3,1))}, & \mathsf{Move((3,3),(3,1))} \\ \mathsf{Transform((3,3),(3,2))}, & \mathsf{Move((3,3),(3,2))} \\ \mathsf{Delete((1,1))}, & \mathsf{Delete((2,1))} \\ \mathsf{Delete((3,1))}, & \mathsf{Delete((1,2))}, \\ \mathsf{Delete((2,2))}, & \mathsf{Delete((3,2))} \end{array} \right\}$$

. The actual cost of edit paths P_0, P_1, and P_2 are 12, 9, and 18, respectively. These costs suggest that the child state c_1, as an intermediate state, is closer to the goal than both its “parent” e_i and its “sibling” c_2. Those costs are consistent with the fact that Drop(0) is a more promising operation than Split(0,‘ ’) from the initial state. (Only one operation—Split(1,‘:’)—is needed to get from c_1 to e_o, whereas three operations are needed to transform c_2 to e_o). This example shows that our proposed heuristic is effective in prioritizing the good operations over the bad ones.

Table Edit Distance Batch

Although TED seems to be a good metric for table dissimilarity, it is not yet a good heuristic function in our problem because (1) it is an estimate of the cost of *table edit path* at a cell level which is on a scale different from our data transformation operations *cost* defined in Definition 3.3 and (2) the TED score depends on the number of cells in the example tables The scaling problem in our setting cannot be fixed by simply multiplying the cost by a constant like has been done in other domains [43], because different Potter's Wheel operators affect different number of cells.

We have developed a novel method called **Table Edit Distance Batch (TED Batch)** (Algorithm 2) that approximates the number of Potter's Wheel operators by grouping table edit operations belonging to certain *geometric patterns* into batches and compacting the cost of each batch. The intuition behind this methodology is based on the observation that data transformation operators usually transform, remove or add cells within the same column or same row or that are geometrically adjacent to each other. Consider Split, for example: it transforms one column in the input table into two or more columns, so instead of counting the individual table edit operations for each affected cell, the operations are batched into groups representing the affected columns.

The definitions of the geometric patterns and related data transformation operators are presented in Table 3.4. For example, "Vertical to Vertical" captures the edit operations that are from vertically adjacent cells (in the same column) in the input table to vertically adjacent cells in the output table. In Figure 3.10, Deletes of $\mathcal{I}_{1,3,5}$ are a batch of edit operations that follow "Vertical to Vertical" pattern.

Algorithm 2: Table Edit Distance Batch

```
   Data: $p_{final} = \{u_{i1} \to v_1, \dots, u_{i|T_2|} \to v_{|T_2|}\}$, $patterns$ from Table 3.4
   Result: cost
1  $batch_{temp} \leftarrow \emptyset$;
2  $batch_{final} \leftarrow \emptyset$;
3  $G_{type} \leftarrow$ Group $p_{final}$ by table edit operators type ;
4  for $g \in G_{type}$ do
5      for $p \in patterns$ do
6          $batch_{temp} \leftarrow batch_{temp} \cup Group\ g\ by\ p$;
7  while $\bigcup batch_{final}$ is not a complete edit path do
8      $batch_{max} \leftarrow argmax_{b \in batch_{temp}} size(b)$ ;
9      if $batch_{max} \cap \bigcup batch_{final} = \emptyset$ then
10         add $batch_{max}$ to $batch_{final}$;
11     $batch_{temp} \leftarrow batch_{temp} \backslash batch_{max}$;
12 $cost \leftarrow 0$;
13 for $group \in batch_{final}$ do
14     $sum \leftarrow 0$;
15     for $editOp \in group$ do
16         $sum \leftarrow sum + cost(editOp)$;
17     $cost \leftarrow cost + sum/size(group)$;
18 Return $cost$;
```

To recalculate the heuristic score using this idea, we propose Algorithm 2, which consists of the following three steps. We use Figure 3.10 and P_0 to demonstrate each step.

Step 3.1. *(lines 3 –6) Find all sets of edit operations (from the edit path obtained by Algorithm 1) following each geometric pattern. Each set is a candidate batch. Each edit operator could only be batched with operators of the same type (e.g.,* Move *should not be in the same batch as* Drop*); line 3 first groups operations by types.*

Pattern	Formulation (X is a table edit operator)	Related Operators
Horizontal to Horizontal	$\{X((x_i, y_i), (x_j, y_j)), X((x_i, y_i + 1), (x_j, y_j + 1)), ...\}$	Delete(Possibly)
Horizontal to Vertical	$\{X((x_i, y_i), (x_j, y_j)), X((x_i, y_i + 1), (x_j + 1, y_j)), ...\}$	Fold, Transpose
Vertical to Horizontal	$\{X((x_i, y_i), (x_j, y_j)), X((x_i + 1, y_i), (x_j, y_j + 1)), ...\}$	Unfold,Transpose
Vertical to Vertical	$\{X((x_i, y_i), (x_j, y_j)), X((x_i + 1, y_i), (x_j + 1, y_j)), ...\}$	Move, Copy, Merge, Split, Extract, Drop
One to Horizontal	$\{X((x_i, y_i), (x_j, y_j)), X((x_i, y_i), (x_j, y_j + 1)), ...\}$	Fold(Possibly), Fill(Possibly)
One to Vertical	$\{X((x_i, y_i), (x_j, y_j)), X((x_i, y_i), (x_j + 1, y_j)), ...\}$	Fold, Fill
Remove Horizontal	$\{X((x_i, y_i)), X((x_i, y_i + 1)), ...\}$	Delete
Remove Vertical	$\{X((x_i, y_i)), X((x_i + 1, y_i)), ...\}$	Drop, Unfold

Table 3.4: Geometric patterns

In P_0, Transform((1,2),(1,1)) ($\mathcal{I}_2$ to $\mathcal{O}_1$ in Figure 3.10) should be grouped by pattern "Vertical to Vertical" with Transform((2,2),(2,1)) ($\mathcal{I}_4$ to $\mathcal{O}_3$) and Transform((3,2),(3,1)) ($\mathcal{I}_6$ to $\mathcal{O}_5$). Meanwhile, it could also be grouped by pattern "One to Horizontal" with Transform((2,2),(2,2)) ($\mathcal{I}_2$ to $\mathcal{O}_2$).

Step 3.2. *(lines 7 – 11) One edit operation might be included in multiple batches in Step 3.1. To finalize the grouping, Algorithm 2 repeatedly chooses the batch with the maximum number of edit operations, and none of the operations in this batch should be already included batch$_{final}$. The finalization terminates when batch$_{final}$ covers a complete edit path.*

In the example in Step 3.1, Transform((1,2),(1,1)) will be assigned to the "Vertical to Vertical" group because it has more members than the "One to Horizontal" group.

Step 3.3. *(lines 13 – 17) The final heuristic score is the sum of the mean cost of edit operations within each chosen batch.*

In this case, the cost of the batch with Transform((1,2),(1,1)), Transform((2,2),(2,1)) and Transform((3,2),(3,1)) will be 1, not 3. Finally, the batched form of P_0 is $\{p_1, p_2, p_3, p_4\}$, where

$$\begin{aligned}
p_1 = \{&\mathsf{Transform((1{,}2),(1{,}1))}, \mathsf{Transform((2{,}2),(2{,}1))},\\
&\mathsf{Transform((3{,}2),(3{,}1))}\},\\
p_2 = \{&\mathsf{Transform((1{,}2),(1{,}2))}, \mathsf{Transform((2{,}2),(2{,}2))},\\
&\mathsf{Transform((3{,}2),(3{,}2))}\},\\
p_3 = \{&\mathsf{Move((1{,}2),(1{,}1))}, \mathsf{Move((2{,}2),(2{,}1))}, \mathsf{Move((3{,}2),(3{,}1))}\},\\
p_4 = \{&\mathsf{Delete((1{,}1))}, \mathsf{Delete((2{,}1))}, \mathsf{Delete((3{,}1))}\}.
\end{aligned}$$

The estimated cost of P_0 is reduced to 4 which is closer to the actual Potter's Wheel cost and less related to the number of cells than using TED alone. Likewise, cost of P_1 is now 3 and cost of P_2 is now 6. In general, this shows that the TED Batch algorithm effectively "scales down" the TED heuristic and reduces the heuristic's correlation to the table size.

3.4.3 Pruning Techniques for Better Efficiency

If we indiscriminately tried all possible operations during graph expansion, the search would quickly become intractable. However, not all the potential operations are valid or reasonable. To reduce the size of the graph and improve the runtime of the search, we created three *global pruning rules* (which apply to all operators) and two *property-specific pruning rules* (which apply to any operators with certain specified properties). The following pruning rules are designed to boost efficiency; our proposed data transformation program synthesis technique is still complete without them.

Global Pruning Rules — These pruning rules apply to all operations in the library.

- *Missing Alphanumerics* — Prune the operation if any letter (a–z, A–Z) or digit (0–9) in e_o does not appear in the resulting child state. We assume transformations will not introduce new information, thus if an operation completely eliminates a character present in e_o from current state, no valid path to the goal state exists.
- *No Effect* — Prune the operation that generates a child state identical to the parent state. In this case, this operation is meaningless and should be removed.
- *Introducing Novel Symbols* — Prune the operation if it introduces a printable non-alphanumeric symbol that is not present in e_o. If an operator were to add such a symbol, it would inevitably require an additional operation later to remove the unneeded symbol.

Property-specific Pruning Rules — The properties of certain operators allow us to define further pruning rules.

- *Generating Empty Columns* — Prune the operation if it adds an empty column in the resulting state when it should not. This applies to Split, Divide, Extract, and Fold. For example, Split adds an empty column to a table when parameterized by a delimiter not present in the input column; this Split is useless and can be pruned.
- *Null In Column* — Prune the operation if a column in the parent state or resulting child state has null value that would cause an error. This applies to Unfold, Fold and Divide. For example, Unfold takes in one column as header and one column as data values: if the header column has null values, it means the operation is invalid, since column headers should not be null values.

3.4.4 Complexity Analysis

The worst-case time complexity for our proposed program synthesis technique is $O((kmn)^d)$, where m is the number of cells in input example e_i, n is the number of cells in the output example e_o, k is the number of candidate data transformation operations for each intermediate table, and d is the number of components in the final synthesized program. In comparison, two of the previous works related to our project, ProgFromEx and FlashRelate, have worst-case time complexities that are exponential in the size of the example the user provides. ProgFromEx's worst-case time complexity is $O(m^n)$, where m is the number of cells in the input example and n is the number of cells in the output example. FlashRelate's worst-case complexity is $O(t^{t-2})$, where t is the number of columns in the output table.

In practice, we believe the complexity exponential in input size will not cause a severe performance issue because none of the three PBE techniques require a large amount of user input. However, if a new usage model arises in the future that allows the user to provide a large example easily, ProgFromEx might become impractical.

3.4.5 Synthesizing Perfect Programs

Since the input-output example $\mathcal{E}$ is the only clue about the desired transformation provided by the user, the effectiveness of our technique could be greatly impacted by the quality of $\mathcal{E}$. We can consider its *fidelity* and *representativeness.*

Fidelity of $\mathcal{E}$ — The success of synthesizing a program is premised on the *fidelity* of the user-specified example $\mathcal{E}$: the end user must not make any mistake while specifying $\mathcal{E}$. Some common mistakes a user might make are: *typos*, *copy-paste-mistakes*, and *loss of information*. This last mistake occurs when the user forgets to include important information, such as column headers, when specifying $\mathcal{E}$. When such mistakes occur, our proposed technique is almost certain to fail. However, the required user input is small, and, as we show in Section 3.5.6, our system usually fails quickly. This could be a simple sign for the end user that their input may be errotic and needs to be fixed. In Section 3.7, we describe future work that allows tolerance for user error.

Representativeness of $\mathcal{E}$ — Once a program $\mathcal{P}$ is generated given the user input, the synthesized program is guaranteed to be *correct*: $\mathcal{P}$ must transform the input example e_i to the output example e_o. However, we do not promise that $\mathcal{P}$ is *perfect*, or guarantees to transform the entire raw data $\mathcal{R}$ as the user may expect. How well a synthesized program generalizes to $\mathcal{R}$ relies heavily on the *representativeness* of $\mathcal{E}$, or how accurately $\mathcal{E}$ reflects the desired transformation. Our proposed synthesis technique requires the user to carefully choose a representative sample from $\mathcal{R}$ as the input example to formulate $\mathcal{E}$. With a small sample from $\mathcal{R}$, there is a risk of synthesizing a $\mathcal{P}$ that will not generalize to $\mathcal{R}$ (similar to overfitting when building a machine learning model with too few training examples). Experimentally, however, we see that a small number (e.g., 2 or 3) of raw data records usually suffices to formulate $\mathcal{E}$ (Section 3.5).

Validation — In Section 3.1, we mentioned that one way the user can validate the synthesized program is by understanding the semantics of the program. Alternatively, the user could follow the sampling-based *lazy* approach of Gulwani et al. [35] To the best of our knowledge, no existing work in the PBE area provides guarantees about the reliability of this approach or how many samples it may require. Of course, not only PBE systems, but work in machine learning and the program test literature must wrestle with the same sampling challenges. Our system neither exacerbates nor ameliorates the situation, so we do not address these issues here.

3.5 Experiments

In this section, we evaluate the effectiveness and efficiency of our PBE data transformation synthesis technique and how much user effort it requires. We implemented our technique in a system called Foofah. Foofah is written in Python and C++ and runs on a 16-core (2.53GHz) Intel Xeon E5630 server with 120 GB RAM.

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

- How generalizable are the synthesized programs output by Foofah? (Section 3.5.2)
- How efficient is Foofah at synthesizing data transformation programs? (Section 3.5.2)
- How is the chosen search method using the TED Batch heuristic better than other search strategies, including BFS and a naïve rule-based heuristic? (Section 3.5.3)
- How effectively do our pruning rules boost the search speed? (Section 3.5.4)
- What happens to Foofah if we add new operators to the operator library? (Section 3.5.5)
- How much effort does Foofah save the end users compared to the baseline system Wrangler? (Section 3.5.6)

- How does Foofah compare to other PBE data transformation systems? (Section 3.5.7)

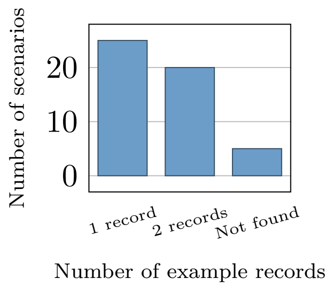
Overall, when supplied with an input-output example comprising two records, Foofah can synthesize perfect data transformation programs for over 85% of test scenarios within five seconds. We also show Foofah requires 60% less user effort than a state-of-art data transformation tool, Wrangler.

3.5.1 Benchmarks

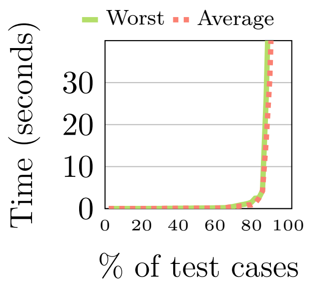
To empirically evaluate Foofah, we constructed a test set of data transformation tasks. Initially, we found 61 test scenarios used in related work including ProgFromEx [35], Wrangler [52], Potter's Wheel (PW) [85, 86] and Proactive Wrangler (Proactive) [33] that were candidate benchmark tests. However, not all test scenarios discovered were appropriate for evaluating Foofah. One of our design assumptions is that the output/-target table must be relational; we eliminated 11 scenarios which violated this assumption. In the end, we created a set of benchmarks with 50 test scenarios[3], among which 37 are real-world data transformation tasks collected in Microsoft Excel forums (from ProgFromEx [35]) and the rest are synthetic tasks used by other related work.

For test scenarios with very little data, we asked a Computer Science student not involved with this project to synthesize more data for each of them following a format similar to the existing raw data of the scenario. This provided sufficient data records to evaluate each test scenario in Section 3.5.2.

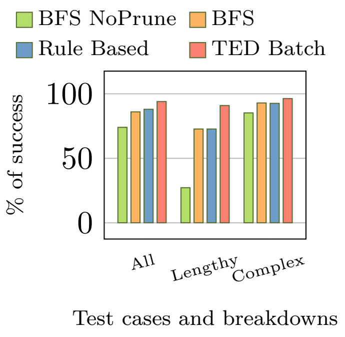
[3]https://github.com/markjin1990/foofah_benchmarks



(a) Number of records required in test scenarios to infer *perfect* programs



(b) Worst and average synthesis time in each interaction



(c) Percentage of *success* breakdowns

Figure 3.11: (a) and (b) show number of records and synthesis time required by Foofah in the experiments of Section 3.5.1; (c) Percentage of successes for different search strategies in the experiments of Section 3.5.3.

3.5.2 Performance Evaluation

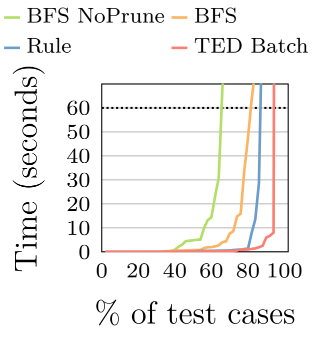
In this section, we experimentally evaluate the response time of Foofah and the *perfectness* of the synthesized programs on all test scenarios. Our experiments were designed in a way similar to that used by an influential work in spreadsheet data transformation, ProgFromEx [35], as well as other related work in data transformation [7, 57].

Overview — For each test scenario, we initially created an input-output example pair (made out of the first record in the output data) and sent this pair to Foofah to synthesize a data transformation program. We executed this program on the entire raw data of the test scenario to check if the raw data was completely transformed as expected. If the inferred program did transform the raw data correctly, Foofah synthesized what we term a *perfect* program. If the inferred program did not transform the raw data correctly, we created a new input-output example (made out of the second record in the output data), making the example more descriptive. We gave the new example to FOOFAH and again checked if the synthesized program correctly transformed the raw data. We repeated this process until FOOFAH found a perfect program, giving each round a time limit of 60 seconds.

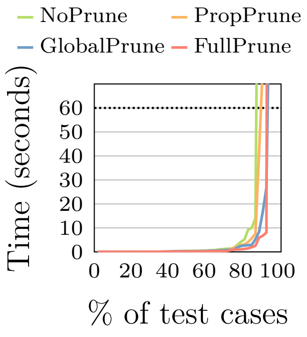
Results — Figure 3.11a shows numbers of data records required to synthesize a perfect program. FOOFAH was able to synthesize perfect programs for 90% of the test scenarios (45 of 50) using input-output examples comprising only 1 or 2 records from the raw data. FOOFAH did not find perfect programs for 5 of the 50 test scenarios. The five failed test scenarios were real-world tasks from PROGFROMEX, but overall FOOFAH still found perfect programs for more than 85% of the real-world test scenarios (32 of 37).

Among the five failed test scenarios, four required unique data transformations that cannot be expressed using our current library of operators; FOOFAH could not possibly synthesize a program that would successfully perform the desired transformation. The remaining failed test scenario required a program that *can* be expressed with FOOFAH's current operations. This program has five steps, which contain two Divide operations. FOOFAH likely failed in this case because Divide separates a column of cells in two columns conditionally, which requires moves of cells following no geometric patterns we defined for TED Batch. The TED Batch heuristic overestimates the cost of paths that include Divide. FOOFAH required more computing time to find the correct path, causing it to reach the 60 second timeout.

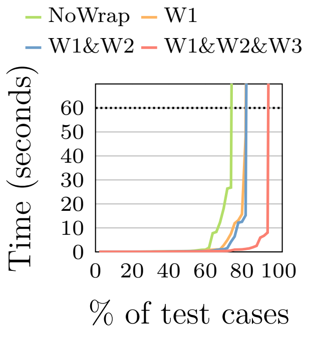
Figure 3.11b shows the average and worst synthesis time of each interaction in all test scenarios. The y-axis indicates the synthesis time in seconds taken by FOOFAH; the x-axis indicates the percentage of test scenarios that completed within this time. The worst synthesis time in each interaction



(a) Compare search strategies



(b) Effectiveness of pruning rules



(c) Adding new operators

Figure 3.12: (a) Percentage of tests synthesized in ≤ Y seconds using different search strategies; (b) Percentage of tests synthesized in ≤ Y seconds with different pruning rules settings; (c) Percentage of tests synthesized in ≤ Y seconds adding Wrap variants.

is less than 1 second for over 74% of the test scenarios (37 of 50) and is less than 5 seconds for nearly 86% of the test scenarios (43 of 50), and the average synthesis time is 1.4 seconds for successfully synthesized perfect programs.

Overall, these experiments suggest that Foofah, aided by our novel TED Batch heuristic search strategy, can efficiently and effectively synthesize data transformation programs. In general, Foofah can usually find a perfect program within interactive response times when supplied with an input-output example made up of *two data records from the raw data*.

3.5.3 Comparing Search Strategies

In this section, we evaluate several search strategies to justify our choice of TED Batch.

Overview — We considered Breadth First Search (BFS) and A* search with a rule-based heuristic (Rule), both mentioned in Section 3.4, and a baseline, Breadth First Search without pruning rules (BFS NoPrune). Based on the conclusion from Section 3.5.1, we created a set of test cases of input-output pairs comprising two records for all test scenarios. In this experiment, each search strategy was evaluated on the entire test set and the synthesis times were measured. The *perfectness* of the synthesized programs was not considered. A time limit of 300 seconds was set for all tests. When a program was synthesized within 300 seconds, we say FOOFAH was successful for the given test case.

Results — Figure 3.11c shows that TED Batch achieves the most successes among all four search strategies and significantly more than the baseline "BFS NoPrune" over the full test suite. To understand the performance of the search strategies in different type of data transformation tasks, we examined the data for two specific categories of test cases.

We first checked the test cases requiring lengthy data transformation programs, since program length is a key factor affecting the efficiency of the search in the state space graph. We considered the program to be *lengthy* if it required four or more operations. Figure 3.11c shows the success rate for all four search strategies in lengthy test cases. TED Batch achieves the highest success rate of any of the strategies, with a margin larger than that for over *all* test cases. This indicates that our proposed strategy, TED Batch, is effective at speeding up the synthesis of lengthy programs.

Since end users often feel frustrated when handling complex data transformations, we wished to know how TED Batch fared compared to other search strategies on complex tasks. We considered test cases that required the operators Fold, Unfold, Divide, Extract to be *complex*. Figure 3.11c shows the success rate for those complex test cases. TED Batch outperforms the other three strategies.

Figure 3.12a shows the time required to synthesize the programs for our set of tests for each search strategy. The TED Batch search strategy is significantly the fastest, with over 90% of the tests completing in under 10 seconds.

3.5.4 Effectiveness of Pruning Rules

One contribution of our work is the creation of a set of pruning rules for data transformation. We examine the efficiency of FOOFAH with and without these pruning rules to show how effectively these pruning rules boost the search speed, using the benchmarks from Section 3.5.1.

Figure 3.12b presents the response times of FOOFAH with pruning rules removed. The pruning rules do improve the efficiency of the program synthesis. However, the difference between the response time of FOOFAH with and without pruning rules is not very significant ($<$ 10s in 86% of the test cases). This is because the search strategy we use—TED Batch—is itself also very effective in "pruning" bad states, by giving them low priority in search. In comparison, if we look at "BFS NoPrune" and "BFS" in Figure 3.12a, the difference between their response time is more significant ($<$ 10s in only 56% of the test cases), showing that the pruning rules are indeed quite helpful at reducing the size of the search space.

3.5.5 Adaptiveness to New Operators

A property of our program synthesis technique is its operator-independence, as we discussed in Section 3.4. To demonstrate this, we compared the efficiency of our prototype, FOOFAH, with and without a newly added operator: Wrap. Wrap has three variants: Wrap on column x (W1), Wrap every n

			WRANGLER				FOOFAH			
Test	Complex	$\geq$ 4 Ops	Time t_W	SE$_{\bar{t}_W}$	Mouse	Key	Time t_F vs t_W	SE$_{\bar{t}_F}$	Mouse	Key
PW1	No	No	**104.2**	16.4	17.8	11.6	**49.4 $\searrow$52.6%**	10.1	20.8	22.6
PW3 (modified)	No	No	**96.4**	17.0	28.8	26.6	**38.6 $\searrow$60.0%**	6.2	14.2	23.6
ProgFromEx13	Yes	No	**263.6**	91.7	59.0	16.2	**145.8 $\searrow$44.7%**	16.8	43.6	78.4
PW5	Yes	No	**242.0**	65.6	52.0	15.2	**58.8 $\searrow$75.7%**	7.4	31.4	32.4
ProgFromEx17	No	Yes	**72.4**	14.9	18.8	11.6	**48.6 $\searrow$32.9%**	10.8	18.2	15.2
PW7	No	Yes	**141.0**	12.9	41.8	12.2	**44.4 $\searrow$68.5%**	1.8	19.6	35.8
Proactive1	Yes	Yes	**324.2**	80.3	60.0	13.8	**104.2 $\searrow$67.9%**	14.6	41.4	57.0
Wrangler3	Yes	Yes	**590.6**	9.4	133.2	29.6	**137.0 $\searrow$76.8%**	13.2	58.6	99.8

Table 3.5: User study experiment results

rows (W2) and Wrap into one row (W3). We examined the responsiveness of FOOFAH on all test cases as we sequentially added the three variants of Wrap.

Figure 3.12c shows the response time of FOOFAH as we add new variants of Wrap, using the same set of test cases as in Section 3.5.3. The addition of the Wrap operations allowed more test scenarios to be successfully completed, while the synthesis time of overall test cases did not increase. This is evidence that the system can be improved through the addition of new operators, which can be easily incorporated without rewriting the core algorithm.

3.5.6 User Effort Study

FOOFAH provides a Programming By Example interaction model in hopes of saving user effort. In this experiment, we asked participants to work on both WRANGLER and FOOFAH and compared the user effort required by both systems.

Overview — We invited 10 graduate students in Computer Science Department who claimed to have no experience in data transformation to participate in our user study. From our benchmark test suite, we chose eight user study tasks of varied length and complexity, shown in Table 3.5.

Column "Complex" indicates if a task requires a complex operator: Fold, Unfold, Divide, and Extract. Column "$\geq$ 4 Ops" indicates if a task requires a data transformation program with 4 or more operations.

Before the experiment, we educated participants on how to use both Wrangler and Foofah with documentation and a complex running example. During the experiment, each participant was given four randomly selected tasks, covering complex, easy, lengthy, and short tasks, to complete on both systems. Each task had a 10 minute time limit.

Evaluation Metrics — To quantify the amount of user effort on both systems, we measured the time a user spends to finish each user study task. In addition to time, we also measured the number of user mouse clicks and keystrokes.

Results — Table 3.5 presents the measurement of the average user efforts on both Wrangler and Foofah over our 8 user study tasks. The percentages of time saving in each test is presented to the right of the time statistics of Foofah. The timing results show that Foofah required 60% less interaction time in every test on average. Foofah also saved more time on complex tasks. On these tasks, Foofah took one third as much interaction time as Wrangler. In the lengthy and complex "Wrangler3" case, 4 of 5 test takers could not find a solution within 10 minutes using Wrangler, but all found a solution within 3 minutes using Foofah.

In Table 3.5 we see that Foofah required an equal or smaller number of mouse clicks than Wrangler. However, Table 3.5 also shows that Foofah required more typing than Wrangler, mainly due to Foofah's interaction model. Typing can be unavoidable when specifying examples, while Wrangler often only requires mouse clicks.

	Layout Trans.	Syntactic Trans.
Foofah	88.4%	100%
ProgFromEx	97.7%	0%
FlashRelate	74.4%	0%

Table 3.6: Success rates for different techniques on both layout transformation and syntactic transformation benchmarks

3.5.7 Comparison with Other Systems

Foofah is not the first PBE data transformation system. There are two other closely related pieces of previous work: ProgFromEx [35] and FlashRelate [7]. In general, both ProgFromEx and FlashRelate are less expressive than Foofah; they are limited to *layout transformations* and cannot handle *syntactic transformations*. Further, in practice, both systems are likely to require more user effort and to be less efficient than Foofah on complex tasks.

Source code and full implementation details for these systems are not available. However, their published experimental benchmarks overlap with our own, allowing us to use their published results in some cases and hand-simulate their results in other cases. As a result, we can compare our system's success rate to that of ProgFromEx and FlashRelate on at least some tasks, as seen in Table 3.6. Note that syntactic transformation tasks may also entail layout transformation steps, but the reverse is not true.

ProgFromEx

The ProgFromEx project employs the same usage model as Foofah: the user gives an "input" grid of values, plus a desired "output" grid, and the system formulates a program to transform the input into the output. A ProgFromEx program consists of a set of *component programs*. Each component program takes in the input table and yields a map, a set of *input-output cell coordinate pairs* that copies cells from the input table to some location in the output table.

A component program can be either a *filter program* or an *associative program*. A filter program consists of a mapping condition (in the form of a conjunction of cell predicates) plus a *sequencer* (a geometric summary of where to place data in the output table). To execute a filter program, ProgFromEx tests each cell in the input table, finds all cells that match the mapping condition, and lets the sequencer decide the coordinates in the output table to which the matching cells are mapped. An associative program takes a component program and applies an additional transformation function to the output cell coordinates, allowing the user to produce output tables using copy patterns that are not strictly one-to-one (e.g., a single cell from the input might be copied to multiple distinct locations in the output).

Expressiveness — The biggest limitation of ProgFromEx is that it cannot describe *syntactic transformations*. It is designed to move values from an input grid cell to an output grid cell; there is no way to perform operations like Split or Merge to modify existing values. Moreover, it is not clear how to integrate such operators into their cell mapping framework. In contrast, our system successfully synthesizes programs for 100% of our benchmark syntactic transformation tasks, as well as 90% of the layout transformation tasks (see Table 3.6). (Other systems can handle these critical syntactic transformation tasks [33, 52, 86], but Foofah is the first PBE system to do so that we know of). ProgFromEx handles slightly more layout transformations in the benchmark suite than our current Foofah prototype, but ProgFromEx's performance comes at a price: the system administrator or the user must pre-define a good set of cell mapping conditions. If the user were willing to do a similar amount of work on Foofah by adding operators, we could obtain a comparable result.

User Effort and Efficiency — For the subset of our benchmark suite that both systems handle successfully (i.e., cases without any syntactic transformations), ProgFromEx and Foofah require roughly equal amounts of user effort. As we describe in Section 3.5.1, 37 of our 50 benchmark test scenarios are borrowed from the benchmarks of ProgFromEx. For each of these 37 benchmarks, both ProgFromEx and Foofah can construct a successful program with three or fewer user-provided examples. Both systems yielded wait times under 10 seconds for most cases.

FlashRelate

FlashRelate is a more recent PBE data transformation project that, unlike ProgFromEx and Foofah, only requires the user to provide output examples, not input examples. However, the core FlashRelate algorithm is similar to that of ProgFromEx: it conditionally maps cells from a spreadsheet to a relational output table. FlashRelate's cell condition tests are more sophisticated than those in ProgFromEx (e.g., they can match on regular expressions and geometric constraints).

Expressiveness — Like ProgFromEx, FlashRelate cannot express syntactic transformations, because FlashRelate requires exact matches between regular expressions and cell contents for cell mapping. Moreover, certain cell-level constraints require accurate schematic information, such as column headers, in the input table. FlashRelate achieves a lower success rate than Foofah in Table 3.6. In principle, FlashRelate should be able to handle some tasks that ProgFromEx cannot, but we do not observe any of these in our benchmark suite.

User Effort and Efficiency — FlashRelate only requires the user to provide output examples, suggesting that it might require less overall user effort than Foofah or ProgFromEx. However, on more than half of the benchmark cases processed by both FlashRelate and Foofah, FlashRelate required five or more user examples to synthesize a correct transformation program, indicating that the effort using FlashRelate may be no less than either ProgFromEx or Foofah. Published results show that more than 80% of tasks complete within 10 seconds, suggesting that FlashRelate's runtime efficiency is comparable to that of Foofah and ProgFromEx.

3.6 Related Work

Both program synthesis and data transformation have been the focus of much research, which we discuss in depth below.

Program Synthesis — Several common techniques to synthesize programs have been discussed in Section 3.4.1: constraint-base program synthesis [32, 44] does not fit our problem because existing logic solvers could not scale to solve a large number of constraints quadratic in the input size; sketching [98] is computationally infeasible for interactive data transformation; version space algebra [29, 55] is usually applied in PBD systems. Therefore, we formulate our problem as a search problem in the state space graph and solve it using a search-based technology with a novel heuristic—TED Batch—as well as some pruning rules.

Researchers have applied program synthesis techniques to a variety of problem domains: parsers [59], regular expressions [11], bit-manipulation programs [31, 44], data structures [97]; code snippets and suggestions in

IDEs [67, 88], and SQL query based on natural language queries [60] and data handling logic [20], schema mappings [4]. There are also several projects that synthesize data transformation and extraction programs, discussed in more detail next.

Data Transformation — Data extraction seeks to extract data from unstructured or semi-structured data. Various data extraction tools and synthesizers have been created to automate this process: TextRunner [6] and WebTables [16] extract relational data from web pages; Senbazuru [17, 18] and FlashRelate [7] extract relations from spreadsheets; FlashExtract [57] extracts data from a broader range of documents including text files, web pages, and spreadsheets, based on examples provided by the user.

Data transformation (or data wrangling) is usually a follow-up step after data extraction, in which the extracted content is manipulated into a form suitable for input into analytics systems or databases. Work by Wu et al. [107, 108, 109], as well as FlashFill [29, 96] and BlinkFill [94] are built for syntactic transformation. DataXFormer [3] and work by Singh and Gulwani [95] are built for semantic transformation. ProgFromEx [35] is built for layout transformation, and Wrangler [52] provides an interactive user interface for data cleaning, manipulation and transformation.

Some existing data transformation program synthesizers follow Programming By Example paradigm similar to FOOFAH [7, 29, 30, 35, 57, 94, 107, 108, 109]. ProgFromEx [35] and FlashRelate [7] are two important projects in PBE data transformation which have been compared with our proposed technique in Section 3.5.7. In general, their lack of expressiveness for syntactic transformations prevent them from addressing many real-world data transformation tasks.

3.7 Conclusion

In this project, we have presented a Programming By Example data transformation program synthesis technique that reduces the user effort for naïve end users. It takes descriptive hints in form of input-output examples from the user and generates a data transformation program that transforms the input example to the output example. The synthesis problem is formulated as a search problem, and solved by a heuristic search strategy guided by a novel operator-independent heuristic function, TED Batch, with a set of pruning rules. The experiments show that our proposed PBE data transformation program synthesis technique is effective and efficient in generating perfect programs. The user study shows that the user effort is 60% less using our PBE paradigm compared to Wrangler [52].

In the future, we would like to extend our system with an interface allowing the user to easily add new data transformation operators and to explore advanced methods of generating the geometric patterns for batching. Additionally, we would like to generate useful programs even when the user's examples may contain errors. We could do so by alerting the user when the system observes unusual example pairs that may be mistakes, or by synthesizing programs that yield outputs *very similar* to the user's specified example.

Chapter 4

Synthesizing Data Format Standardization Programs using Pattern-based Examples

4.1 Introduction

Data transformation, or data wrangling, is a critical pre-processing step essential to effective data analytics on real-world data and is widely known to be human-intensive as it usually requires professionals to write ad-hoc scripts that are difficult to understand and maintain. A *human-in-the-loop* Programming By Example (PBE) approach has been shown to reduce the burden for the end user: in projects such as FlashFill [29], BlinkFill [94], and Foofah [46], the system synthesizes data transformation programs using simple examples the user provides.

Problems — Most of existing research in PBE data transformation tools has focused on the "system" part — improving the efficiency and expressivity of the program synthesis techniques. Although these systems have demonstrated some success in efficiently generating high-quality data transformation programs for real-world data sets, **verification**, as an indispensable interaction procedure in PBE, remains a major bottleneck within existing PBE data transformation system designs.

Any reasonable user who needs to perform data transformation should certainly care about the "correctness" of the inferred transformation logic. In fact, a user will typically go through rounds of "verify-and-specify" cycles when using a PBE system. In each interaction, a user has to verify the correctness of the current inferred transformation logic by validating the transformed data instance by instance until she identifies a data instance mistakenly transformed; then she has to provide a new example for correction. **Given a potentially large and varied input data set, such a verification process is like "finding a needle in a haystack" which can be extremely time-consuming and tedious, and may deter the user from confidently using these tools.**

A naïve way to simplify the cumbersome verification process is to add some visual aids to the transformed data so that the user does not have to read them in their raw form. For example, if we can somehow know the desired data pattern, we can write a checking function to automatically check if the post-transformed data satisfies the desired pattern, and highlight data entries that are not correctly transformed.

However, a visual aid alone can not solve the entire verification issue; the discovered transformation logic is undisclosed and mystifying to the end user. Users can at best verify that existing data are converted into the right form, but **the logic is not guaranteed to be correct and may function unexpectedly on new input** (see Section 4.2 for an example). Without good insight into the transformation logic, PBE system users cannot tell if the inferred transformation logic is correct, or when there are errors in the logic, they may not be able to debug it. **If the user of a traditional PBE system lacks a good understanding of the synthesize program's logic, she can only verify it by spending large amounts of time testing the synthesized program on ever-larger datasets**.

Naïvely, previous PBE systems can support *program explanation* by presenting the inferred programs to end users. However, these data transformation systems usually design their own Domain Specific Languages (DSLs), which are usually sophisticated. The steep learning curve makes it unrealistic for most users to quickly understand the actual logic behind the inferred programs. Thus, besides more explainable data, a desirable PBE system should be able to present the transformation logic in a way that most people are already familiar with.

Insight — Regular expressions (regexp) have been known to most programmers of various expertise and `regexp replace` operations have been commonly applied in data transformations. The influential data transformation system, Wrangler (later as Trifacta), proposes simplified natural-language-like regular expressions which can be understood and used even by non-technical data analysts. This makes `regexp replace` operations a promising candidate as an *explainable transformation language* for non-professionals. The challenge then is how to automatically synthesize `regexp replace` operations as the desired transformation logic in a PBE system.

A `regexp replace` operation takes in two parameters: an *input pattern* and a *replacement function*. Suppose an input data set is given, and the desired data pattern can be known, the challenge is to determine a suitable input pattern and the replacement function to convert all input data into the desired pattern. Moreover, if the input data set is heterogeneous with many formats, we need to find out an unknown set of such input-pattern-

and-replace-function pairs.

Pattern profiling, or syntactic profiling, is another useful technique to address the problem. It is to discover the structural patterns that summarize the data [76]. For example, given a set of messy phone numbers, multiple formats could be discovered, such as “`\d{3}-\d{3}-\d{4}`”, “`\d{7}`”, and so on. Some of the important work in automatic pattern profiling include PADS [25], Datamaran [26], and FlashProfile [76]. The patterns discovered by a pattern profiler can be naturally used to generate `RegExp Replace` operations. Moreover, it can also serve as a data explanation approach helping the user quickly understand the pre- and post-transformation data which reduces the verification challenge users face in PBE systems.

Proposed Solution — In this project, we propose a new data transformation paradigm, CLX, to address the two specific problems within our claimed verification issue. The CLX paradigm has three components: two algorithmic components—*clustering* and *transformation*—with an intervening component of *labeling*. Here, we present an instantiation of the CLX paradigm. We present (1) an efficient pattern clustering algorithm that groups data with similar structures into small clusters, (2) a DSL for data transformation, that can be interpreted as a set of regular expression replace operations, (3) a program synthesis algorithm to infer desirable transformation logic in the proposed DSL.

Through the above means, we are able to greatly ameliorate the usability issue in verification within PBE data transformation systems. Our experimental results show improvements over the state of the art in saving user verification effort, along with increasing users’ comprehension of the inferred transformations. Increasing comprehension is highly relevant to reducing the verification effort. In one user study on a large data set, when the data size grew by a factor of 30, the CLX prototype cost 1.3× more verification time whereas FlashFill cost 11.4× more verification time. In a separate user study accessing the users' understanding of the transformation logic, Clx users achieved a success rate about twice that of FlashFill users. Other experiments also suggest that the expressive power of the Clx prototype and its efficiency on small data are comparable to those of FlashFill.

Organization — After motivating our problem with an example in Section 4.2, we discuss the following contributions:

- We define the data transformation problem and present the PBE-like CLX framework solving this problem. (Section 4.3)
- We present a data pattern profiling algorithm to hierarchically cluster the raw data based on patterns. (Section 4.4)
- We present a new DSL for data pattern transformation in the Clx paradigm. (Section 4.5)
- We develop algorithms synthesizing data transformation programs, which can transform any given input pattern to the desired standard pattern. (Section 4.6)
- We experimentally evaluate the Clx prototype and other baseline systems through user studies and simulations. (Section 4.7)

We finish with a discussion of future work in Section 4.9.

4.2 Motivating Example

Bob is a technical support employee at the customer service department. He wanted to have a set of 10,000 phone numbers in varied formats (as in Figure 4.1) in a unified format of "(xxx) xxx-xxxx". Given the volume and

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

Figure 4.1: Phone numbers with diverse formats

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

Figure 4.2: Patterns after transformation

`\({digit}3\){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 4.3: Pattern clusters of raw data

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

Figure 4.4: Suggested data transformation operations

the heterogeneity of the data, neither manually fixing them or hard-coding a transformation script was convenient for Bob. He decided to see if there was an automated solution to this problem.

Bob found that Excel 2013 had a new feature named FlashFill that could transform data patterns. He loaded the data set into Excel and performed FlashFill on them.

Example 4.1. *Initially, Bob thought using* FlashFill *would be straightforward: he would simply need to provide an example of the transformed form of each ill-formatted data entry in the input and copy the exact value of each data entry already in the correct format. However, in practice, it turned out not to be so easy. First, Bob needed to carefully check each phone* *number entry deciding whether it is ill-formatted or not. After obtaining a new input-output example pair,* FlashFill *would update the transformation results for the entire input data, and Bob had to carefully examine again if any of the transformation results were incorrect. This was tedious given the large volume of heterogeneous data* ***(verification at string level is challenging)****. After rounds of repairing and verifying, Bob was finally sure that* FlashFill *successfully transformed all existing phone numbers in the data set, and he thought the transformation inferred by* FlashFill *was impeccable. Yet, when he used it to transform another data set, a phone number "+1 724-285-5210" was mistakenly transformed as "(1) 724-285", which suggested that the transformation logic may fail anytime* ***(unexplainable transformation logic functions unexpectedly)****. Customer phone numbers were critical information for Bob's company and it was important not to damage them during the transformation. With little insight from* FlashFill *regarding the transformation program generated, Bob was not sure if the transformation was reliable and had to do more testing* ***(lack of understanding increases verification effort)****.*

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

Example 4.2. *He loaded his data into* Clx *and it immediately presented a list of distinct, natural language-like string patterns for phone numbers in the input data (Figure 4.3), which helped Bob quickly tell which part of the data were ill-formatted. After Bob selected the desired pattern,* Clx *immediately transformed all the data and showed a new list of string patterns as Figure 4.2.* ***So far, verifying the transformation result was straightforward.*** *The inferred program is presented as a set of* Replace *operations on raw patterns in Figure 4.3, each with a picture visualizing the transformation effect (like Figure 4.8). Enhanced by visualizations and nat-*

Notation	Description
$\mathcal{S} = \{s_1, s_2, ...\}$	A set of ad hoc strings $s_1, s_2, ...$ to be transformed.
$\mathcal{P} = \{p_1, p_2, ...\}$	A set of string patterns derived from $\mathcal{S}$.
$p_i = \{t_1, t_2, ...\}$	Pattern made from a sequence of tokens t_i
$\mathcal{T}$	The desired target pattern that all strings in $\mathcal{S}$ needed to be transformed into.
$\mathcal{L} = \{(p_1, f_1), (p_2, f_2), ...\}$	Program synthesized in CLX transforming data the patterns of $\mathcal{P}$ 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 this section.
$\mathcal{Q}(\tilde{\mathfrak{t}}, p)$	Frequency of token $\tilde{\mathfrak{t}}$ in pattern p
$\mathcal{G}$	Potential expressions represented in Directed Acyclic Graph.

Table 4.1: Frequently used notations

ural language-like pattern representations, these Replace *operations seemed not difficult to understand and verify. Like many users in our User Study (Section 4.7.3),* ***Bob had a deeper understanding of the inferred transformation logic with* Clx *than with* FlashFill*, and hence, he knew well when and how the program may fail, which saved him from the effort of more blind testing.***

4.3 Overview

4.3.1 Patterns and Data Transformation Problem

A data pattern, or string pattern, is a "high-level" description of the attribute value's string. A natural way to describe a pattern could be a regular expression over the characters that constitute the string. In data transformation, 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 4.2 presents all *token classes* we currently support in our instantiation of CLX, including their class names, regular expressions, and notation. In addition, we also support tokens of constant values (e.g., ",", ":"). In the rest of the section, we represent and handle these tokens of constant values differently from the 5 token classes defined in Table 4.2. For convenience of presentation, we denote such tokens with constant values as ***literal tokens*** and tokens of 5 token classes defined in Table 4.2 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. A quantifier is either a single natural number or "+", indicating that the token appears at least once. In the rest of this section, to be succinct, a token will be denoted as "$\langle\tilde{\mathfrak{t}}\rangle\mathfrak{q}$" if $\mathfrak{q}$ is a number (e.g., $\langle D\rangle 3$) or "$\langle\tilde{\mathfrak{t}}\rangle+$" otherwise (e.g., $\langle D\rangle+$). If $\tilde{\mathfrak{t}}$ is a literal token, it will be surrounded by a single quotation mark, like ':'. When a pattern is shown to the end user, it is presented as a *natural language-like regular expression* proposed by WRANGLER [52] (see regexps in Fig 4.4).

With the above definition of data patterns, we hereby formally define the problem we tackle using the CLX framework—data transformation. Data transformation or wrangling is a broad concept. Our focus in this project is to apply the CLX paradigm to transform a data set of heterogeneous patterns into a desired pattern. A formal definition of the problem is as follows:

Definition 4.1 (Data (Pattern) Transformation)**.** *Given a set of strings*

Token Class	Regular Expression	Example	Notation
digit	`[0-9]`	"12"	$\langle D \rangle$
lower	`[a-z]`	"car"	$\langle L \rangle$
upper	`[A-Z]`	"IBM"	$\langle U \rangle$
alpha	`[a-zA-Z]`	"Excel"	$\langle A \rangle$
alpha-numeric	`[a-zA-Z0-9_-]`	"Excel2013"	$\langle AN \rangle$

Table 4.2: Token classes and their descriptions

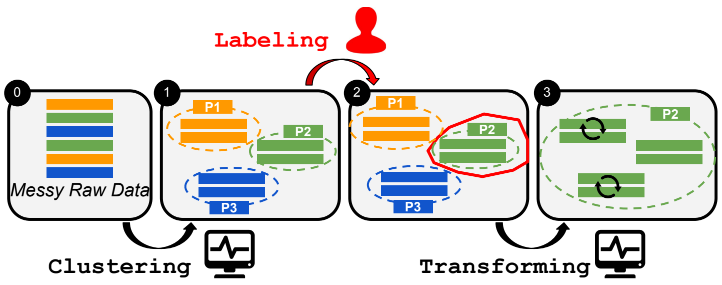


Figure 4.5: "CLX" Model: Cluster–Label–Transform

$\mathcal{S} = \{s_1, \ldots, s_n\}$, *generate a program $\mathcal{L}$ that transforms each string in $\mathcal{S}$ to an equivalent string matching the user-specified desired target pattern $\mathcal{T}$.*

$\mathcal{L} = \{(p_1, f_1), (p_2, f_2), \ldots\}$ is the program we synthesize in the transforming phase of CLX. It is represented as a set regexp replace operations, Replace(p, f)[4], that many people are familiar with (e.g., Fig 4.4).

With above definitions of patterns and data transformations, we present the CLX framework for data transformation.

[4] p is the regular expression, and f is the replacement string indicating the operation on the string matching the pattern p.

4.3.2 Clx Data Transformation Paradigm

We propose a data transformation paradigm called Cluster-Label-Transform (Clx, pronounced "clicks"). Figure 4.5 visualizes the interaction model in this framework.

Clustering — The clustering component groups the raw input data into clusters based on their data patterns/formats. Compared to raw strings, data patterns is a more abstract representation. The number of patterns is fewer than raw strings, and hence, it can make the user understand the data and verify the transformation more quickly. Patterns discovered during clustering is also useful information for the downstream program synthesis algorithm to determine the number of regexp replace operations, as well as the desirable input patterns and transformation functions.

Labeling — Labeling is to specify the desired data pattern that every data instance is supposed to be transformed into. Presumably, labeling can be achieved by having the user choose among the set of patterns we derive in the clustering process assuming some of the raw data already exist in the desired format. If no input data matches the target pattern, the user could alternatively choose to manually specify the target data form.

Transforming — After the desired data pattern is labeled, the system automatically synthesizes data transformation logic that transforms all undesired data into the desired form and also proactively helps the user understand the transformation logic.

In this section, we present an instantiation of the Clx paradigm for *data pattern transformation*. Details about the clustering component and the transformation component are discussed in Section 4.4 and 4.6. In Section 4.5, we show the domain-specific-language (DSL) we use to represent the program $\mathcal{L}$ as the outcome of program synthesis, which can be then presented as the regexp replace operations. The paradigm has been designed to allow new algorithms and DSLs for transformation problems other than data pattern transformation; we will pursue other instantiations in future work.

4.4 Clustering data on patterns

In Clx, we first cluster data into meaningful groups based on their pattern structure and obtain the pattern information, which helps the user quickly understand the data. To minimize user effort, this clustering process should ideally not require user intervention.

PADS [25] is a seminal project that also targets string pattern discovery. However, PADS is orthogonal to our effort in that their goal is mainly to find a comprehensive and unified description for the entire data set whereas we seek to partition the data into clusters, each cluster with a single data pattern. Also, the PADS language [24] itself is known to be hard for a non-expert to read [112]. Our interest is to derive simple patterns that are comprehensible. Besides the explainability, efficiency is another important aspect of the clustering algorithm we must consider, because input data can be huge and the real-time clustering must be interactive.

A recent work FlashProfile [76] targets the same problem of pattern profiling for a dataset. Compared to previous work, FlashProfile is more in line with our expectation for a pattern profiler in that 1) it is able to profile patterns for a messy dataset with varied formats, 2) it returns multiple simple patterns instead of one giant and complex pattern as the final output, which are more readable, 3) it allows the user-defined atoms (atomic patterns).

However, the approach proposed by FlashProfile is not quite applicable in our case. First, FLASHPROFILE requires the number of returned patterns/clusters k be pre-determined by the end user, whereas we believe asking the user to know k for an unfamiliar dataset can be non-trivial in our case. Second, the patterns discovered by FLASHPROFILE are mostly for understanding purposes. Since all patterns returned by FLASHPROFILE are final, it is not as clear how to use these patterns in synthesizing transformation programs afterwards as if they were subject to change.

In this project, we design an automated means to hierarchically cluster data based on data patterns given a set of strings. The data is clustered through a two-phase profiling: (1) tokenization: tokenize the given set of strings of ad hoc data and cluster based on these initial patterns, (2) bottom-up refinement: recursively merge pattern clusters to formulate a ***pattern cluster hierarchy***. In this case, the patterns for each data entry from the raw dataset can be in varied generalities, which 1) allows the end user to view/understand the pattern structure information in a simpler and more systematic way, and also 2) facilitates the subsequent transformation program synthesis.

4.4.1 Initial Clustering Through 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 about 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 the tokenization phase.

Example 4.3. *Suppose the string “Bob123@gmail.com” is to be tokenized. The result of tokenization becomes [*$\langle U \rangle$*,* $\langle L \rangle 2$*,* $\langle D \rangle 3$*, ‘@’,* $\langle L \rangle 5$*, ‘.’,* $\langle L \rangle 3$*].*

After tokenization, each string corresponds to a data pattern composed of tokens. We create the initial set of pattern clusters by clustering the strings sharing the same patterns. Each cluster uses its pattern as a label which will later be used for refinement, transformation, and user understanding.

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.’,‘\ ’, ‘$\langle U \rangle$’, ‘$\langle L \rangle$+’] than [‘$\langle U \rangle$’, ‘$\langle L \rangle$’, ‘.’, ‘\ ’, ‘$\langle U \rangle$’, ‘$\langle L \rangle$+’]. Similar to [25], we find tokens with constant values using the statistics over tokenized strings in the data set.

4.4.2 Bottom-up Pattern Cluster Refinement

In the initial clustering step, we distinguish different patterns by token classes, token positions, and quantifiers, the actual number of pattern clusters discovered in the ad hoc data in the tokenization phase could be huge. User comprehension is inversely related to the number of patterns. It is not

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

very helpful to present too many very specific pattern clusters all at once to the user. Plus, it can be unacceptably expensive to develop data pattern transformation programs separately for each pattern.

To mitigate the problem, we build *pattern cluster hierarchy*, i.e., a hierarchical pattern cluster representation with the leaf nodes being the patterns discovered through tokenization, and every internal node being a *parent pattern*. 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. Plus, we do not lose any pattern discovered previously.

From bottom-up, we recursively cluster the patterns at each level to obtain *parent patterns*, i.e., more generic patterns, formulating the new layer in the hierarchy. To build a new layer, Algorithm 3 takes in different generalization strategy $\tilde{g}$ and the child pattern set $\mathcal{P}$ from the last layer. Line 3-5 clusters the current set of pattern clusters to get parent pattern clusters us-

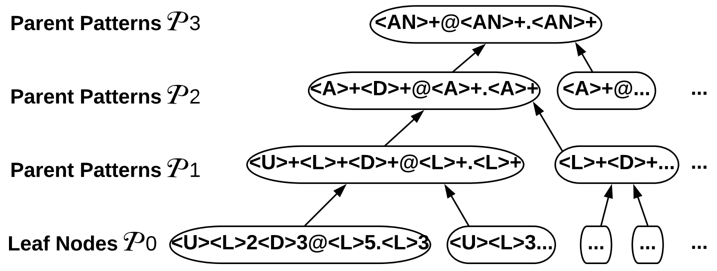


Figure 4.6: Hierarchical clusters of data patterns

ing the generalization strategy $\tilde{g}$. The generated set of parent patterns may be identical to others or might have overlapping expressive power. Keeping all these parent patterns in the same layer of the cluster hierarchy is unnecessary and increases the complexity of the hierarchy 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). Overall, the complexity is $\mathcal{O}(n \log n)$, where n is the number of patterns in $\mathcal{P}$, and hence, the algorithm itself can quickly converge.

In this project, we perform three rounds of refinement to construct the new layer in the hierarchy, each with a particular generalization strategy:

1. natural number quantifier to '+'
2. $\langle L \rangle$, $\langle U \rangle$ tokens to $\langle A \rangle$
3. $\langle A \rangle$, $\langle N \rangle$, '-', '_' tokens to $\langle AN \rangle$

Example 4.4. *Given the pattern we obtained in Example 4.3, we successively apply Algorithm 3 with Strategy 1, 2 and 3 to generalize parent* *patterns $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$ and construct the pattern cluster hierarchy as in Figure 4.6.*

4.4.3 Limitations

The pattern hierarchy constructed can succinctly profile the pattern information for many data. However, the technique itself may be weak in two situations. First, as the scope of this project is limited to addressing the syntactic transformation problem (Section 4.5), the pattern discovery process we propose only considers syntactic features, and no semantic features. Ignoring semantic features may introduce the issue of "misclustering". For example, a date of format "MM/DD/YYYY" and a date of format "DD/MM/YYYY" may be grouped into the same cluster of "$\langle N\rangle 2/\langle N\rangle 2/\langle N\rangle 4$", and hence, transforming from the former format into the latter format is impossible in our case. Addressing this problem requires the support for semantic information discovery and transformation, which will be in our future work. Another possible weakness of "fail to cluster" is also mainly affected by the semantics issue: we may fail to cluster semantically-same but very messy data. E.g., we may not cluster the local-part (everything before '@') of a very weird email address "Mike'John.Smith@gmail.com" (token $\langle AN\rangle$ cannot capture ''' or '.'). Yet, this issue can be easily resolved by adding additional regexp-based token classes (e.g., emails). Adding more token classes is beyond the interest of our work.

4.5 Data Pattern Transformation Program

As motivated in Section 4.1 and Section 4.3, our proposed data transformation framework is to synthesize a set of regexp replace operations that people

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

Figure 4.7: UniFi Language Definition

are familiar with as the desired transformation logic. However, representing the logic as regexp strings will make the program synthesis difficult. Instead, to simplify the program synthesis, we propose a new language, UniFi, as a representation of the transformation logic internal to Clx. The grammar of UniFi is shown in Figure 4.7. We then discuss how to explain an inferred UniFi program as regexp replace operations.

The top-level of any UniFi program 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 discovered previously. Once a string matches this pattern, it will be processed by an *atomic transformation plan* (expression $\mathcal{E}$ in UniFi) defined below.

Definition 4.2 (Atomic Transformation Plan)**.** *Atomic transformation plan is a sequence of parameterized string operators that converts a given source pattern into the target pattern.*

The available string operators include ConstStr and Extract. ConstStr($\tilde{s}$) denotes a constant string $\tilde{s}$. Extract($\tilde{\mathfrak{t}}_i$,$\tilde{\mathfrak{t}}_j$) extracts from the i^{th} token to the j^{th} token in a pattern. In the rest of the section, we denote an Extract operation as Extract(i,j), or Extract(i) if $i = j$. A token $\mathfrak{t}$ is represented as $(\tilde{\mathfrak{t}}, \mathfrak{r}, \mathfrak{q}, i)$: $\tilde{\mathfrak{t}}$ is the token class in Table 4.2; $\mathfrak{r}$ represents the corresponding regular expression of this token; $\mathfrak{q}$ is the quantifier of the token expression; i denotes the index (one-based) of this token in the source pattern.

As with FlashFill [29] and BlinkFill [94], we only focus on syntactic transformation, where strings are manipulated as a sequence of characters and no external knowledge is accessible, in this instantiation design. *Semantic transformation* (e.g., converting "March" to "03") is a subject for future work. Further–again like BlinkFill–our proposed data pattern transformation language UniFi does not support loops. Without the support for loops, UniFi may not be able to describe transformations on an unknown number of occurrences of a given pattern structure.

We use the following two examples used by FlashFill and BlinkFill to briefly demonstrate the expressive power of UniFi, and the more detailed expressive power of UniFi would be examined in the experiments in Section 4.7.4. For simplicity, $\mathsf{Match}(s,p)$ is shortened as $\mathsf{Match}(p)$ as the input string s is fixed for a given task.

Example 4.5. *This problem is modified from test case "**Example 3**" in* BlinkFill. *The goal is to transform all messy values in the medical billing codes into the correct form "[CPT-XXXX]" as in Table 4.3.*

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

Table 4.3: Normalizing messy medical billing codes

The UniFi *program for this standardization task is*

```
Switch((Match("\[<U>+\-<D>+"),
    (Concat(Extract(1,4),ConstStr(']')))),
  (Match("<U>+\-<D>+"),
    (Concat(ConstStr('['),Extract(1,3),
      ConstStr(']'))))
  (Match("<U>+<D>+"),
    (Concat(ConstStr('['),Extract(1),
    ConstStr('-'),Extract(2),ConstStr(']')))))
```

Example 4.6. *This problem is borrowed from "**Example 9**" in* FLASH-FILL. *The goal is to transform all names into a unified format as in Table 4.4.*

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

Table 4.4: Normalizing messy employee names

A UNIFI *program for this task is*

```
Switch((Match("<U><L>+\.\ <U><L>+\ <U><L>+"),
    Concat(Extract(8,9),ConstStr(','),
        ConstStr(' '),Extract(5))),
  (Match("<U><L>+\ <U><L>+\,\ <U><L>+\."),
    Concat(Extract(4,5),ConstStr(','),
        ConstStr(' '),Extract(1))),
  (Match("<U><L>+\ <U>+\ <U><L>+"),
    Concat(Extract(6,7),ConstStr(','),
        ConstStr(' '),Extract(1))))
```

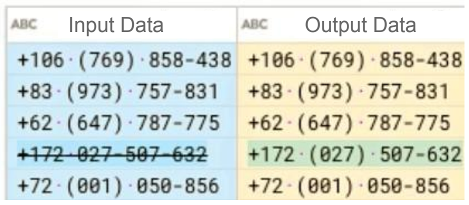
Program Explanation — Given a UNIFI program L, we want to present it as a set of regexp replace operations, Replace, parameterized by *natural-language-like regexps* used by Wrangler [52] (e.g., Figure 4.4), which are straightforward to even non-expert users. Each component of $(b, \mathcal{E})$, within the Switch statement of L, will be explained as a Replace operation. The replacement string f in the Replace operation is created from p and the transformation plan $\mathcal{E}$ for the condition b. In f, a ConstStr($\tilde{s}$) operation will remain as $\tilde{s}$, whereas a Extract($\tilde{t_i}, \tilde{t_j}$) operation will be interpreted as $\$\tilde{t_i} \dots \$\tilde{t_j}$. The pattern p in the predicate $b = \mathsf{Match}(s, p)$ in UniFi naturally becomes the regular expression p in Replace with each tokens to be extracted surrounded by a pair of parentheses indicating that it can be extracted. Note that if multiple consecutive tokens are extracted in p, we merge them as one component to be extracted in p and change the f accordingly for convenience of presentation. Figure 4.4 is an example of the transformation logic finally shown to the user.

In fact, these Replace operations can be further explained using visualization techniques. For example, we could add a *Preview Table* (e.g., Figure 4.8) to visualize the transformation effect in our prototype in a sample of the input data. The user study in Section 4.7.3 demonstrates that our effort of outputting an explainable transformation program helps the user understand the transformation logic generated by the system.

4.6 Program Synthesis

We now discuss how to find the desired transformation logic as a UniFi program using the pattern cluster hierarchy obtained. Algorithm 4 shows our synthesis framework.

Given a pattern hierarchy, we do not need to create an *atomic transformation plan* (Definition 4.2) for every pattern cluster in the hierarchy. We traverse the pattern cluster hierarchy top-down to find valid *candidate source patterns* (line 6, see Section 4.6.1). Once a source candidate is identified, we discover all *token matches* between this source pattern in Q_{solved}



Input Data	Output Data
+106 (769) 858-438	+106 (769) 858-438
+83 (973) 757-831	+83 (973) 757-831
+62 (647) 787-775	+62 (647) 787-775
~~+172-027-507-632~~	+172 (027) 507-632
+72 (001) 050-856	+72 (001) 050-856

Figure 4.8: Preview Tab

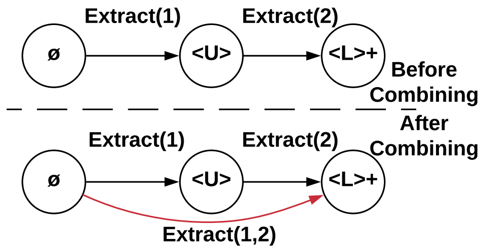


Figure 4.9: Combine Extracts

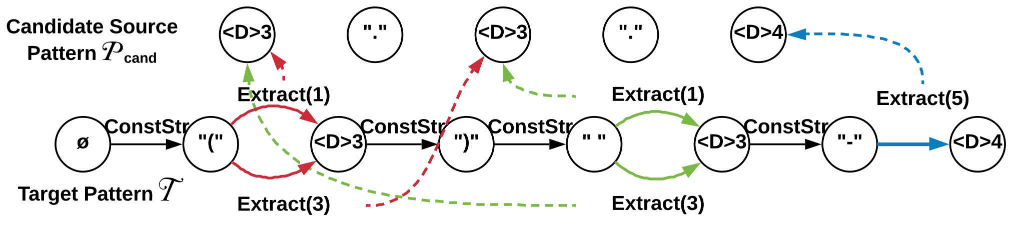


Figure 4.10: Token alignment for the target pattern $\mathcal{T}$

and the target pattern (line 7, see Section 4.6.2). With the generated token match information, we synthesize the data pattern normalization program including an atomic transformation plan for every source pattern (line 11, see Section 4.6.3).

4.6.1 Identify Source Candidates

Before synthesizing a transformation for a source pattern, we want to quickly check whether it can be a *candidate source pattern* (or source candidate), i.e., it is possible to find a transformation from this pattern into the target pattern, through `validate`. **If we can immediately disqualify some patterns, we do not need to go through a more expensive data transformation synthesis process for them.** There are a few reasons why some pattern in the hierarchy may not be qualified as a candidate source pattern:

1. The input data set may be ad hoc and a pattern in this data set can be a description of noise values. For example, a data set of phone numbers may contain "N/A" as a data record because the customer refused to reveal this information. In this case, it is meaningless to generate transformations.

2. We may be fundamentally not able to support some transformations (e.g., semantic transformations are not supported as in our case). Hence, we should filter out certain patterns which we think semantic transformation is unavoidable, because it is impossible to transform them into the desired pattern without the help from the user.

3. Some patterns are too general; it can be hard to determine how to transform these patterns into the target pattern. We can ignore them and create transformation plans for their children. For instance, if a pattern is "$\langle AN\rangle+,\langle AN\rangle+$", it is hard to tell if or how it could be transformed into the desired pattern of "$\langle U\rangle\langle L\rangle+ : \langle D\rangle+$". By comparison, its child pattern "$\langle U\rangle\langle L\rangle+, \langle D\rangle+$" seems to be a better fit as the candidate source.

Any input data matching no candidate source pattern is left unchanged and flagged for additional review, which could involve replacing values with NULL or default values or manually overriding values.

Since the goal here is simply to quickly prune those patterns that are not good source patterns, the checking process should be able to find unqualified source patterns with *high precision* but not necessarily *high recall*. Here, we use a simple heuristic of *frequency count* that can effectively reject unqualified source patterns with high confidence: examining if there

Algorithm 4: Synthesize UniFi Program

```
Data: Pattern cluster hierarchy root $\mathcal{P}_R$, target pattern $\mathcal{T}$
Result: Synthesized program $\mathcal{L}$
1  $\mathcal{Q}_{unsolved}, \mathcal{Q}_{solved} \leftarrow [\ ]$ ;
2  $\mathcal{L} \leftarrow \emptyset$;
3  push $\mathcal{P}_R$ to $\mathcal{Q}_{unsolved}$;
4  while $\mathcal{Q}_{unsolved} \neq \emptyset$ do
5      $p \leftarrow$ pop $\mathcal{Q}_{unsolved}$;
6      if validate$(p, \mathcal{T}) = \top$ then
7          $\mathcal{G} \leftarrow$ findTokenAlignment$(p, \mathcal{T})$;
8          push $\{p, \mathcal{G}\}$ to $\mathcal{Q}_{solved}$;
9      else
10         push $p$.children to $\mathcal{Q}_{unsolved}$;
11 $\mathcal{L} \leftarrow$ createProgs$(\mathcal{Q}_{solved})$;
12 Return $\mathcal{L}$
```

are sufficient base tokens of each class in the source pattern matching the base tokens in the target tokens. The intuition is that any source pattern with fewer base tokens than the target is unlikely to be transformable into the target pattern without external knowledge; base tokens usually carry semantic meanings and hence are likely to be hard to invent *de novo*.

To apply frequency count on the source pattern p_1 and the target pattern p_2, `validate` (denoted as $\mathcal{V}$) compares the *token frequency* for every class of base tokens in p_1 and p_2. The token frequency $\mathcal{Q}$ of a token class $\langle\tilde{\mathtt{t}}\rangle$ in p is defined as

$$\mathcal{Q}(\langle\tilde{\mathtt{t}}\rangle, p) = \sum_{i=1}^{n} \{t_i.\mathtt{q} | t.name = \langle\tilde{\mathtt{t}}\rangle\}, p = \{t_1, \ldots, t_n\} \tag{4.1}$$

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

Suppose $\mathfrak{T}$ is the set of all token classes (in our case, $\mathfrak{T} = [\langle D\rangle, \langle L\rangle, \langle U\rangle, \langle A\rangle, \langle AN\rangle]$), $\mathcal{V}$ is then defined as

$$\mathcal{V}(p_1, p_2) = \begin{cases} \text{true} & \text{if } \mathcal{Q}(\langle\tilde{\mathfrak{t}}\rangle, p_1) \geq \mathcal{Q}(\langle\tilde{\mathfrak{t}}\rangle, p_2), \forall\langle\tilde{\mathfrak{t}}\rangle \in \mathfrak{T} \\ \text{false} & \text{otherwise} \end{cases} \tag{4.2}$$

Example 4.7. *Suppose the target pattern $\mathcal{T}$ in Example 4.5 is [‘[’, $\langle U\rangle+$, ‘-’, $\langle D\rangle+$, ‘]’], we know*

$$\mathcal{Q}(\langle D\rangle, \mathcal{T}) = \mathcal{Q}(\langle U\rangle, \mathcal{T}) = 1$$

A pattern [‘[’, $\langle U\rangle 3$, ‘-’, $\langle D\rangle 5$] derived from data record “[CPT-00350” will be identified as a source candidate by ***validate****, because*

$$\mathcal{Q}(\langle D\rangle, p) = 5 > \mathcal{Q}(\langle D\rangle, \mathcal{T}) \wedge$$
$$\mathcal{Q}(\langle U\rangle, p) = 3 > \mathcal{Q}(\langle U\rangle, \mathcal{T})$$

Another pattern [‘[’, $\langle U\rangle 3$, ‘-’] derived from data record “[CPT-” will be rejected because

$$\mathcal{Q}(\langle D\rangle, p) = 0 < \mathcal{Q}(\langle D\rangle, \mathcal{T})$$

4.6.2 Token Alignment

Once a source pattern is identified as a source candidate in Section 4.6.1, 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 matches for each token in the target pattern: discover all possible operations that yield a token. This process is called *token alignment*.

```
Algorithm 5: Token Alignment Algorithm
Data: Target pattern $\mathcal{T} = \{t_1, \dots, t_m\}$, candidate source pattern $\mathcal{P}_{cand} =$
      $\{t'_1, \dots, t'_n\}$, where $t_i$ and $t'_i$ denote base tokens
Result: Directed acyclic graph $\mathcal{G}$
1  $\tilde{\eta} \leftarrow \{0, \dots, n\}$; $\eta^s \leftarrow 0$; $\eta^t \leftarrow n$; $\xi \leftarrow \{\}$;
2  for $t_i \in \mathcal{T}$ do
3      for $t'_j \in \mathcal{P}_{cand}$ do
4          if $SyntacticallySimilar(t_i, t'_j) = \top$ then
5              $e \leftarrow \mathsf{Extract}(t'_j)$;
6              add $e$ to $\xi_{(i-1,i)}$;
7      if $t_i.type =$ 'literal' then
8          $e \leftarrow \mathsf{ConstStr}(t_i.name)$;
9          add $e$ to $\xi_{(i-1,i)}$;
10 for $i \in \{1, \dots, n-1\}$ do
11     $\xi_{in} \leftarrow \{\forall e_p \in \xi_{(i-1,i)}, e_p$ is an $\mathsf{Extract}$ operation$\}$;
12     $\xi_{out} \leftarrow \{\forall e_q \in \xi_{(i,i+1)}, e_q$ is an $\mathsf{Extract}$ operation$\}$;
13     for $e_p \in \xi_{in}$ do
14         for $e_q \in \xi_{out}$ do
15             if $e_p.srcIdx + 1 = e_q.srcIdx$ then
16                 $e \leftarrow \mathsf{Extract}(e_p.t_i, e_q.t_j)$;
17                 add $e$ to $\xi_{(i-1,i+1)}$;
18 $\mathcal{G} \leftarrow Dag(\tilde{\eta}, \eta^s, \eta^t, \xi)$;
19 Return $\mathcal{G}$
```

For each token in the target pattern, there might be multiple different token matches. Inspired by [29], we store the results of the token alignment in Directed Acyclic Graph (DAG) represented as a $DAG(\tilde{\eta}, \eta^s, \eta^t, \xi)$. $\tilde{\eta}$ 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 $\tilde{\eta}$ 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 5.

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 4.3) to the target token t_i in $\mathcal{T}$, we create a token match between t'_j and t_i with an Extract operation on an edge from t_{i-1} to t_i (line 2-9).

Definition 4.3 (Syntactically Similar)**.** *Two tokens t_i and t_j are syntactically similar if: 1) they have the same class, 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 7-9), instead of extracting it from the source pattern. This does not violate our previous assumption of not introducing any external knowledge during the transformation.

Example 4.8. *Let the candidate source pattern be [$\langle D\rangle 3$, '.', $\langle D\rangle 3$, '.', $\langle D\rangle 4$] and the target pattern be ['(', $\langle D\rangle 3$, ')', ' ', $\langle D\rangle 3$, '-', $\langle D\rangle 4$]. Token alignment result for the source pattern $\mathcal{P}_{cand}$ and the target pattern $\mathcal{T}$, generated by Algorithm 5 is shown in Figure 4.10. In Figure 4.10, a dashed line is a token match, indicating the token(s) in the source pattern that can formulate a token in the target pattern. A solid line embeds the actual operation in* UniFi *rendering this token match.*

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 4-9 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 10-17). The Extracts are then added back to ξ. Figure 4.9 visualizes combining two sequential Extracts. The first half of the figure (titled "Before Combining") shows a transformation plan that generates a target pattern pattern $\langle U\rangle\langle D\rangle+$ with two operations— Extract(1) and Extract(2). The second half of the figure (titled "After Combining") showcases merging the incoming edge and the outgoing edge (representing the previous two operations) and formulate a new operation (red arrow), Extract(1,2), as a combined operation of the two.

A benefit of discovering sequential extracts is it helps yield a "simple" program, as described in Section 4.6.3.

Correctness — Algorithm 5 is *sound* and *complete*, which is proved below.

Theorem 4.1 (Soundness). *If the token alignment algorithm (Algorithm 5) successfully discovers a token correspondence, it can be transformed into a* UniFi *program.*

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 5 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 can be possibly admitted into an atomic transformation plan, which will end up becoming part of a UniFi program. The soundness is true. □

Theorem 4.2 (Completeness)**.** *If there exists a* UniFi *program, the token alignment algorithm (Algorithm 5) will for sure discover the corresponding token correspondence matching the 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 5, line 4-6 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 7-9 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 token class, and should be merged as one token whose quantifier is the sum of all these tokens. Similarly, the reverse is also true. What remains to prove is whether Algorithm 5 is guaranteed to generate an Extract of multiple tokens, i.e., $\mathsf{Extract}(p, q)(p < q)$, in the source pattern that produces multiple tokens in the target pattern. In Algorithm 5, line 4-6 is guaranteed to discover $\mathsf{Extract}(p)$, $\mathsf{Extract}(p + 1)$, ... , $\mathsf{Extract}(q)$. With these Extracts, when performing line 11-17 when $i = p + 1$ in Algorithm 5, it will discover the incoming edge representing $\mathsf{Extract}(p)$ and the output edge representing $\mathsf{Extract}(p + 1)$ and combine them, generating $\mathsf{Extract}(p, p + 1)$. When $i = p + 2$, it will discover the incoming edge representing $\mathsf{Extract}(p, p + 1)$ and the outgoing edge representing $\mathsf{Extract}(p + 2)$ and combine them, generating $\mathsf{Extract}(p, p + 2)$. If we repeat this process, we will definitely find

$\mathsf{Extract}(p, q)$ in the end. Therefore, the solution is complete. □

4.6.3 Program Synthesis using Token Alignment Result

As we represent all token matches for a source pattern as a DAG (Algorithm 5), finding a transformation plan is to find a path from the initial state 0 to the final state l, where l is the length of the target pattern $\mathcal{T}$.

The Breadth First Traversal algorithm can find all possible atomic transformation plans for this DAG. However, not all of these plans are equally likely to be correct and desired by the end user. The hope is to prioritize the correct plan. The Occam's razor principle suggests that the simplest explanation is usually correct. Here, we apply **Minimum Description Length** (MDL) [90], a formalization of Occam's razor principle, to gauge the *simplicity* of each candidate program, and rank them.

Suppose $\mathcal{M}$ is the set of models. In this case, it is the set of atomic transformation plans found given the source pattern $\mathcal{P}_{cand}$ and the target pattern $\mathcal{T}$. $\mathcal{E} = f_1 f_2 \dots f_n \in \mathcal{M}$ is an atomic transformation plan, where f is a string expression. Inspired by [86], we define *Description length* (DL) as follows:

$$L(\mathcal{E}, \mathcal{T}) = L(\mathcal{E}) + L(\mathcal{T}|\mathcal{E}) \tag{4.3}$$

$L(\mathcal{E})$ is the *model description length*, which is the length required to encode the model, and in this case, $\mathcal{E}$. Hence,

$$L(\mathcal{E}) = |\mathcal{E}| \log m \tag{4.4}$$

where m is the number of distinct types of operations.

$L(\mathcal{T}|\mathcal{E})$ is the *data description length*, which is the sum of the length required to encode $\mathcal{T}$ using the atomic transformation plan $\mathcal{E}$. Thus,

$$L(\mathcal{T}|\mathcal{E}) = \sum_{f_i \in \mathcal{E}} \log L(f_i) \tag{4.5}$$

where $L(f_i)$ the length to encode the parameters for a single expression. For an $\mathsf{Extract(i)}$ or $\mathsf{Extract(i,j)}$ operation, $L(f) = \log |\mathcal{P}_{cand}|^2$ (recall $\mathsf{Extract(i)}$ is short for $\mathsf{Extract(i,i)}$). For a $\mathsf{ConstStr}(\tilde{s})$, $L(f) = \log c^{|\tilde{s}|}$, where c is the size of printable character set ($c = 95$).

With the concept of description length described, we define the minimum description length as

$$L_{min}(\mathcal{T}, \mathcal{M}) = \min_{\mathcal{E} \in \mathcal{M}} \Big[L(\mathcal{E}) + L(\mathcal{T}|\mathcal{E}) \Big] \tag{4.6}$$

In the end, we present the atomic transformation plan $\mathcal{E}$ with the minimum description length as the default transformation plan for the source pattern. Also, we list the other k transformation plans with lowest description lengths.

Example 4.9. *Suppose the source pattern is "$\langle D\rangle 2/\langle D\rangle 2/\langle D\rangle 4$", the target pattern $\mathcal{T}$ is "$\langle D\rangle 2/\langle D\rangle 2$". The description length of a transformation plan $\mathcal{E}_1$ = Concat(Extract(1,3)) is $L(\mathcal{E}_1, \mathcal{T}) = 1\log 1 + 2\log 3$. In comparison, the description length of another transformation plan $\mathcal{E}_2$ = Concat(Extract(1), ConstStr('/'),Extract(3)) is $L(\mathcal{E}_2, \mathcal{T}) = 3\log 2 + \log 3^2 + \log 95 + \log 3^2 > L(\mathcal{E}_1, \mathcal{T})$. Hence, we prefer $\mathcal{E}_1$, a simpler plan than $\mathcal{E}_2$, and more likely to be correct from our perspective.*

4.6.4 Limitations and Program Repair

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. Also, we currently do not support "semantic transformation". We may face the issue of "semantic ambiguity"—mismatching syntactically similar tokens with different semantic meanings. For example, if the goal is to transform a date of pattern "DD/MM/YYYY" into the pattern "MM-DD-YYYY" (our clustering algorithm works in this case). Our token alignment algorithm may create a match from "DD" in the first pattern to "MM" in the second pattern because they have the same pattern of $\langle D \rangle 2$. The atomic transformation plan we *initially* select for each source pattern can be a transformation that mistakenly converts "DD/MM/YYYY" into "DD-MM-YYYY".

Fortunately, as our token alignment algorithm is complete and the program synthesis algorithm can discover all possible transformations and rank them in a smart way, the user can quickly find the correct transformation through *program repair*: replace the initial atomic transformation plan with another atomic transformation plans among the ones Section 4.6.3 suggests for a given source pattern.

To make the repair even simpler for the user, we deduplicate equivalent atomic transformation plans defined below before the repair phase.

Definition 4.4 (Equivalent Plans)**.** *Two Transformation Plans are* equivalent *if, given the same source pattern, they always yield the same transformation result for any matching string.*

For instance, suppose the source pattern is $[\langle D \rangle 2$, '/', $\langle D \rangle 2]$. Two transformation plans $\mathcal{E}_1 = [\mathsf{Extract}(3), \mathsf{Const}(\text{'/'}), \mathsf{Extract}(1)]$ and $\mathcal{E}_2 = [\mathsf{Extract}(3), \mathsf{Extract}(2), \mathsf{Extract}(1)]$ will yield exactly the same output because the first and third operations are identical and the second operation will always generate a '/' in both plans. If two plans are *equivalent*, presenting both rather than one of them will only increase the user effort. Hence, we only pick the simplest plan in the same equivalence class and prune the rest. Checking whether a candidate transformation plan P_1 is equivalent to another candidate transformation plan P_2 is performed through the following procedures:

1. Split each $\mathsf{Extract}(m, n)$ operation in both plans into $\mathsf{Extract}(m)$,$\mathsf{Extract}(m+1)$, ... , $\mathsf{Extract}(n)$.
2. Assuming $P_1 = \{op_1^1, op_2^1, \dots, op_n^1\}$ and $P_2 = \{op_1^2, op_2^2, \dots, op_m^2\}$. If $m \neq n$, we stop checking and return **False**. Otherwise, from left to right, we compare operations of two plans one by one. For example, we first compare op_1^1 with op_1^2, then op_2^1 with op_2^2, and so on. The check continues when

   a) op_k^1 is exactly the same as op_k^2, or

   b) op_k^1 is not same as op_k^2. However, one of them is an Extract operation and the other is a ConstStr operation, and the first operation extracts a constant string whose content is exactly the same as the content of the second operation.

   .
3. We stop and return **True** if we reach the end of both plans.

The computational complexity of above pairwise comparison is clearly linear to the length of the plan, and is therefore inexpensive.

Overall, the repair process does not significantly increase the user effort.

4.7 Experiments

We make three broad sets of experimental claims. First, we show that as the input data becomes larger and messier, Clx tends to be less work to use than FlashFill because verification is less challenging (Section 4.7.2). Second, we show that Clx programs are easier for users to understand than FlashFill programs (Section 4.7.3). Third, we show that Clx's expressive power is similar to that of baseline systems, as is the required effort for non-verification portions of the PBE process (Section 4.7.4).

4.7.1 Experimental Setup

We implemented a prototype of Clx and compared it against the state-of-the-art PBE system FlashFill. For ease of explanation, in this section, we refer this prototype as "Clx". Additionally, to make the experimental study more complete, we had a third baseline approach, a non-PBE feature offered by TrifactaWrangler[5] allowing the user to perform string transformation through manually creating Replace operations with simple natural-language-like regexps (referred as RegexReplace). All experiments were performed on a 4-core Intel Core i7 2.8G CPU with 16GB RAM. Other related PBE systems, Foofah [46] and TDE [37], target different workloads and also share the same verification problem we claim for PBE systems, and hence, are not considered as baselines.

[5]TrifactaWrangler is a commercial product of Wrangler launched by Trifacta Inc. The version we used is 3.2.1

4.7.2 User Study on Verification Effort

In this section, we conduct a user study on a real-world data set to show that (1) verification is a laborious and time-consuming step for users when using the classic PBE data transformation tool (e.g., FlashFill) particularly on a large messy data set, (2) asking end users to hand-write regexp-based data transformation programs is challenging and inefficient, and (3) the Clx model we propose effectively saves the user effort in verification during data transformation and hence its interaction time does not grow fast as the size and the heterogeneity of the data increase.

Test Data Set — Finding public data sets with messy formats suitable for our experiments is very challenging. The first experiment uses a column of 331 messy phone numbers from the "Times Square Food & Beverage Locations" data set [75].

Overview — The task was to transform all phone numbers into the form "$\langle D\rangle$3-$\langle D\rangle$3-$\langle D\rangle$4". We created three test cases by randomly sampling the data set with the following data sizes and heterogeneity: "10(2)" has 10 data records and 2 patterns; "100(4)" has 100 data records and 4 patterns; "300(6)" has 300 data records and 6 patterns.

We invited 9 students in Computer Science with a basic understanding of regular expressions and not involved in our project. Before the study, we educated all participants on how to use the system. Then, each participant was asked to work on one test case on a system and we recorded their performance.

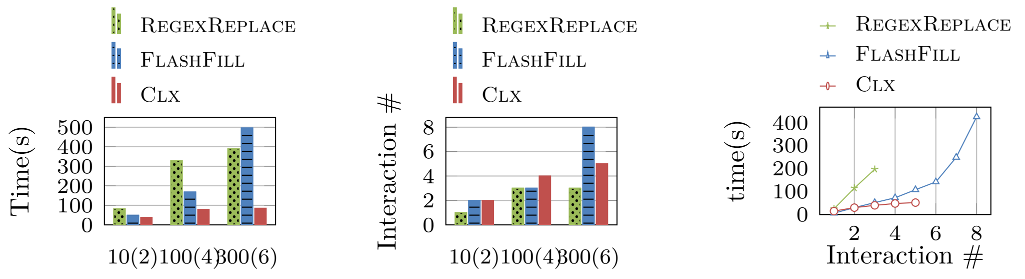
We looked into the user performances on three systems from various perspectives: *overall completion time*, *number of interactions*, and *verification time*. The *overall completion time* gave us a quick idea of how much the cost of user effort was affected when the input data was increasingly large and heterogeneous in this data transformation task. The other two metrics allowed us to check the user effort in verification. While measuring completion time is straightforward, the other two metrics need to be clarified.

Number of interactions. For FlashFill, the number of interactions is essentially the number of examples the user provides. For Clx we define the number of interactions as the number of times the user verifies (and repairs, if necessary) the inferred atomic transformation plans. We also add one for the initial labeling interaction. For RegexReplace, the number of interactions is the number of Replace operations the user creates.

Verification Time. All three systems follow different interaction paradigms. However, we can roughly divide the interaction process into two parts, *verification* and *specification*: the user is either busy inputting (typing keyboards, selecting, etc.) or paused to verify the correctness of the transformed data or synthesized/hand-written regular expressions.

Measuring verification time is meaningful because we hypothesize that PBE data transformation systems become harder to use when data is large and messy not because the user has to provide a lot more input, but it becomes harder to verify the transformed data at the instance level.

Results — As shown in Figure 4.11a, “100(4)” cost 1.1× more time than “10(2)” on Clx, and “300(6)” cost 1.2× more time than “10(2)” on Clx. As for FlashFill, “100(4)” cost 2.4× more time than “10(2)”, and “300(6)” cost 9.1× more time than “10(2)”. Thus, in this user study, the user effort required by Clx grew slower than that of FlashFill. Also, RegexReplace cost significantly more user effort than Clx but its cost grew not as quickly as FlashFill. This shows good evidence that (1) manually writing data transformation script is cumbersome, (2) the user interaction time grows very fast in FlashFill when data size and heterogeneity increase,



(a) Overall completion time **(b) Rounds of interactions** **(c) Interaction timestamps for 300(6)**

Figure 4.11: Scalability of the system usability as data volume and heterogeneity increases (shorter bars are better)

and (3) the user interaction time in Clx also grows, but not as fast.

Now, we dive deeper into understanding the causes for observation (2) and (3). Figure 4.11b shows the number of interactions in all test cases on all systems. We see that all three systems required a similar number of interactions in the first two test cases. Although FlashFill required 3 more interactions than Clx in case "300(6)", this could hardly be the main reason why FlashFill cost almost 5x more time than Clx.

We take a close look at the three systems' interactions in the case of "300(6)" and plot the timestamps of each interaction in Figure 4.11c. The result shows that, in FlashFill, as the user was getting close to achieving a perfect transformation, it took the user an increasingly longer amount of time to make an interaction with the system, whereas the interaction time intervals were relatively stable in Clx and RegexReplace. Obviously, the user spent a longer time in each interaction NOT because an example became harder to type in (phone numbers have relatively similar lengths). We observed that, without any help from FlashFill, the user had to eyeball the entire data set to identify the data records that were

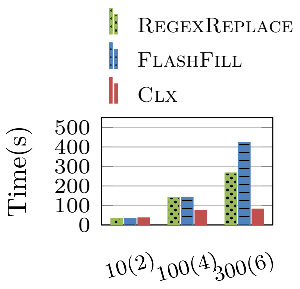


Figure 4.12: Verification time (shorter bars are better)

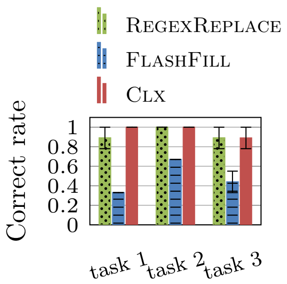


Figure 4.13: User comprehension test (taller bars are better)

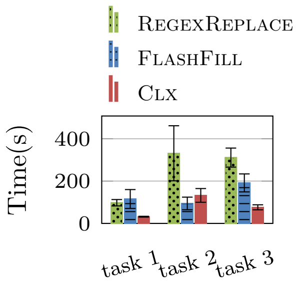


Figure 4.14: Completion time (shorter bars are better)

still not correctly transformed, and it became harder and harder to do so simply because there were fewer of them. Figure 4.12 presents the average verification time on all systems in each test case. "100(4)" cost 1.0× more verification time than "10(2)" on Clx, and "300(6)" cost 1.3× more verification time than "10(2)" on Clx. As for FlashFill, "100(4)" cost 3.4× more verification time than "10(2)", and "300(6)" cost 11.4× more verification time than "10(2)". The fact that the verification time on FlashFill also grew significantly as the data became larger and messier supports our analysis and claim.

To summarize, this user study presents evidence that FlashFill becomes much harder to use as the data becomes larger and messier mainly because verification is more challenging. In contrast, Clx users generally are not affected by this issue.

Task ID	Size	AvgLen	MaxLen	DataType
Task1	10	11.8	14	Human name
Task2	10	20.3	38	Address
Task3	100	16.6	18	Phone number

Table 4.5: Explainability test cases details

4.7.3 User Study on Explainability

Through a new user study with the same 9 participants on three tasks, we demonstrate that (1) FLASHFILL users lack understanding about the inferred transformation logic, and hence, have inadequate insights on how the logic will work, and show that (2) the simple program generated by CLX improves the user's understanding of the inferred transformation logic.

Additionally, we also compared the overall completion time of three systems.

Test Set — Since it was impractical to give a user too many data pattern transformation tasks to solve, we had to limit this user study to just a few tasks. To make a fair user study, we chose tasks with various data types that cost relatively the same user effort on all three systems. From the benchmark test set we will introduce in Section 4.7.4, we randomly chose 3 test cases that each is supposed to require same user effort on both CLX and FLASHFILL: Example 11 from FlashFill (task 1), Example 3 from PredProg (task 2) and "phone-10-long" from SyGus (task 3). Statistics (number of rows, average/max/min string length of the raw data) about the three data sets are shown in Table 4.5.

Overview — We designed 3 multiple choice questions for every task examining how well the user understood the transformation regardless of the system he/she interacted with. The complete set of questions are shown as follows:

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

During the user study, we asked every participant to participate all three tasks, each on a different system (completion time was measured). Upon completion, each participant was asked to answer all questions based on the transformation results or the synthetic programs generated by the system.

Explainability Results — The correct rates for all 3 tasks using all systems are presented in Figure 4.13. The result shows that the participants were able to answer these questions almost perfectly using CLX, but struggled to get even half correct using FLASHFILL. REGEXREPLACE also achieved a success rate similar to CLX, but required higher user effort and expertise.

The result suggests that FLASHFILL users have insufficient understanding about the inferred transformation logic and CLX improves the users' understanding in all tasks, which provides evidence that verification in CLX can be easier.

Overall Completion Time — The average completion time for each task using all three systems is presented in Figure 4.14. Compared to FLASHFILL, the participants using CLX spent 30% less time on average: $\sim 70\%$

Sources	# tests	AvgSize	AvgLen	MaxLen	DataType
SyGus [100]	27	63.3	11.8	63	car model ids, human name, phone number, university name and address
FlashFill [29]	10	10.3	15.8	57	log entry, phone number, human name, date, name and position, file directory, url, product name
BlinkFill [94]	4	10.8	14.9	37	city name and country, human name, product id, address
PredProg [96]	3	10.0	12.7	38	human name, address
Prose [89]	3	39.3	10.2	44	country and number, email, human name and affiliation
Overall	47	43.6	13.0	63	

Table 4.6: Benchmark test cases details

less time on task 1 and $\sim 60\%$ less time on task 3, but $\sim 40\%$ more time on task 2. Task 1 and task 3 have similar heterogeneity but task 3 (100 records) is bigger than task 1 (10 records). 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 provides more evidence that Clx saves the verification effort. Task 2 is small (10 data records) but heterogeneous. Both FlashFill and Clx made imperfect transformation logic synthesis, and the participants had to make several corrections or repairs. We believe Clx lost in this case simply because the data set is too small, and as a result, Clx was not able to exploit its advantage in saving user effort on large-scale data sets. The study also gives evidence that Clx is sometimes effective in saving user verification effort in small-scale data transformation tasks.

4.7.4 Expressivity and Efficiency Tests

In a simulation test using a large benchmark test set, we demonstrate that (1) the expressive power of Clx is comparable to the other two baseline systems FlashFill and RegexReplace, and (2) Clx is also pretty efficient in costing user interaction effort.

Test Set — We created a benchmark of 47 data pattern transformation test cases using a mixture of public string transformation test sets and example tasks from related research publications. The information about the number of test cases from each source, average raw input data size (number of rows), average/max data instance length, and data types of these test cases are shown in Table 4.6.

Among the 47 test cases we collected, 27 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 normalization benchmarks of 27 tasks. We collected 10 tasks from FlashFill [29]. 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 [94], 3 tasks from PredProg [96], 3 tasks from Microsoft PROSE SDK [89].

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. Also, the current Clx 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.

Overview — We evaluated Clx against 47 benchmark tests. As conducting an actual user study on all 47 benchmarks is not feasible, we simulated a user following the "lazy approach" used by Gulwani et al. [35]: a simulated user selected a target pattern or multiple target patterns and then repaired the atomic transformation plan for each source pattern if the system proposed answer was imperfect.

Also, we tested the other two systems against the same benchmark test suite. As with Clx, we simulated a user on FlashFill; this user provided the first positive example on the first data record in a non-standard pattern, and then iteratively provided positive examples for the data record on which the synthetic string transformation program failed. On RegexReplace, the simulated user specified a Replace operation with two regular expressions indicating the matching string pattern and the transformed pattern, and iteratively specified new parameterized Replace operations for the next ill-formatted data record until all data were in the correct format.

Evaluation Metrics — In experiments, we measured how much user effort all three systems required. Because systems follow different interaction models, a direct comparison of the user effort is impossible. We quantify the user effort by *Step*, which is defined differently as follows

- For Clx, the total Steps is the sum of the number of correct patterns the user chooses (Selection) and the number of repairs for the source patterns whose default atomic transformation plans are incorrect (Repair). In the end, we also check if the system has synthesized a "perfect" program: a program that successfully transforms all data.
- For FlashFill, 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.

Baselines	CLX Wins	Tie	CLX Loses
vs. FLASHFILL	17 (36%)	17 (36%)	13 (28%)
vs. REGEXREPLACE	33 (70%)	12 (26%)	2 (4%)

Table 4.7: User effort simulation comparison.

- For REGEXREPLACE, 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.

In each test, for any system, if not all data records were correctly transformed, we added the number of data records that the system fails to transform correctly to its total *Step* value as a punishment. In this way, we had a coarse estimation of the user effort in all three systems on the 47 benchmarks.

Expressivity Results — CLX could synthesize right transformations for 42/47 ($\sim$ 90%) test cases, whereas FLASHFILL reached 45/47 ($\sim$ 96%). This suggests that the expressive power of CLX is comparable to that of FLASHFILL.

There were five test cases where CLX failed to yield a perfect transformation. Only one of the failures was due to the expressiveness of the language itself, the others could be fixed if there were more representative examples in the raw data. "Example 13" in FlashFill requires the inference of advanced conditionals (Contains keyword "picture") that UNIFI cannot currently express, but adding support for these conditionals in UNIFI is straightforward. 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 transformation task, where there is a last name

"McMillan" to extract. However, all data in the target pattern contained last names comprising 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.

RegexReplace allows the user to specify any regular expression replace operations, hence it was able to correctly transform all the input data existed in the test set, because the user could directly write operations replacing the exact string of an individual data record into its desired form. However, similar to UniFi, RegexReplace is also limited by the expressive power of regular expressions and cannot support advanced conditionals. As such, it covered 46/47 ($\sim$ 98%) test cases.

User Effort Results — As the *Step* metric is a potentially noisy measure of user effort, it is more reasonable to check whether Clx costs more or less effort than other baselines, rather than to compare absolute *Step* numbers. The aggregated result is shown in Table 4.7. It suggests Clx often requires less or at least equal user effort than both PBE systems. Compared to RegexReplace, Clx almost always costs less or equal user effort.

4.8 Related Work

Data Transformation — FlashFill (now a feature in Excel) is an influential work for syntactic transformation by Gulwani [29]. 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. A more recent PBE project, TDE [37], also targets string transformation.

Similar to FlashFill, TDE requires the user to verify at the instance level and the generated program is unexplainable to the user. Other related PBE data cleaning projects include [46, 94].

Another thread of seminal research including [86], Wrangler [52] and Trifacta created by Hellerstein et al. follow a different interaction paradigm called "predictive interaction". They proposed an inference-enhanced visual platform supporting many different data wrangling and profiling tasks. Based on the user selection of columns, rows or text, the system intelligently suggests possible data transformation operations, such as Split, Fold, or pattern-based extraction operations.

Pattern Profiling — In our project, we focus on clustering ad hoc string data based on structures and derive the structure information. The LearnPADS [25] project is somewhat related. It presents 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 [24]) is hard for a human to read, whereas our pattern cluster hierarchy is simpler to understand. Most recently, Datamaran[26] has proposed methodologies for discovering structure information in a data set whose record boundaries are unknown, but for the same reasons as LearnPADS, Datamaran is not suitable for our problem. FlashProfile [76] is a more recent pattern profiling system that is closer to our focus in this project. Most importantly, it targets a messy dataset with varied formats. Second, it allows the user-defined atoms (atomic patterns), which makes it more expressive. Third, the returned patterns are more readable. FlashProfile proposes a two-phase approach –

"clustering" and "profiling" – to return a fixed number of patterns representing all data entries in the dataset. However, this approach is not quite applicable in our case. First, FlashProfile requires the number of returned patterns/clusters k be pre-determined by the end user, whereas we believe asking the user to know k for an unfamiliar dataset can be non-trivial in our case. Second, the patterns discovered by FlashProfile are mostly for understanding purposes. Since all patterns returned by FlashProfile are fixed, it is not as clear how to use these patterns in synthesizing transformation programs afterwards as if they were subject to change.

Program Synthesis — Program synthesis has garnered 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 [60, 84, 105], regular expressions [11, 62], data cleaning programs [29, 107], and more.

Researchers have proposed various techniques for program synthesis. [32, 44] 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 [29, 55, 56, 71]. [22] recently focuses on using deep learning for program synthesis. Most of these projects rely on user inputs to reduce the search space until a quality program can be discovered; they share the hope that there is one simple solution matching most, if not all, user-provided example pairs. In our case, transformation plans for different heterogeneous patterns can be quite distinct. Thus, applying the version space algebra technique is difficult.

4.9 Conclusion

Data transformation is a difficult human-intensive task. PBE is a leading approach of using computational inference to reduce human burden in data transformation. However, we observe that standard PBE for data transformation is still difficult to use due to its laborious and unreliable verification process.

We proposed a new data transformation paradigm Clx to alleviate the above issue. In Clx, data patterns are initially constructed given a column of string data to be transformed. They are to help the user quickly identify both well-formatted and ill-formatted data which immediately saves the verification time. Clx also infers regexp replace operations as the desired transformation, which many users are familiar with and boosts their confidence in verification.

We presented an instantiation of Clx with a focus on data pattern transformation including (1) a pattern profiling algorithm that hierarchically clusters both the raw input data and the transformed data based on data patterns, (2) a DSL, UniFi, that can express many data pattern transformation tasks and can be interpreted as a set of simple regular expression replace operations, (3) algorithms inferring a correct UniFi program.

We presented two user studies. In a user study on data sets of various sizes, when the data size grew by a factor of 30, the user verification time required by Clx grew by 1.3× whereas that required by FlashFill grew by 11.4×. The comprehensibility user study shows the Clx users achieved a success rate about twice that of the FlashFill users. The results provide good evidence that Clx greatly alleviates the verification issue.

Although building a highly-expressive data pattern transformation tool is not the central goal of this project, we are happy to see that the expressive power and user effort efficiency of our initial design of Clx is comparable to those of FlashFill in a simulation study on a large test set in another test.

Clx is a data transformation paradigm that can be used not only for data pattern transformation but other data transformation or transformation tasks too. For example, given a set of heterogeneous spreadsheet tables storing the same information from different organizations, Clx can be used to synthesize programs converting all tables into the same standard format. Building such an instantiation of Clx will be our future work.

Chapter 5

Synthesis of Complex Schema Mapping Queries using "Multiresolution" Examples

5.1 Introduction

Most data analysts and machine learning practitioners expect their data to be prepared in a well-structured, stand-alone data frame or table. However, in reality, input data is often spread across multiple different relations or schemas in a single database, or even worse, scattered in a *data lake*, a repository that stores both structured and unstructured data at various scales collected from heterogeneous data sources. To prepare the data, schema mapping is often a critical first step, converting data from source databases with different schemas to a *target schema*, or *mediated schema*, i.e. a desirable schema that provides an integrated view of the data sources of interest.

Composing a schema mapping query for a real-world complex database

State	Area$^{km^2}$ (State)	Lake Name	Area$^{km^2}$ (Lake)
California	411,047	Lake Tahoe	497
Oregon	251,418	Crater Lake	53.2
Florida	151,939	Fort Peck Lake	981

...

Table 5.1: Desired target schema

```
SELECT
  Province.Name, Province.Area,
  Lake.Name, Lake.Area
FROM
  geo_lake, Lake, Province
WHERE
  geo_Lake.Lake = Lake.Name
  AND
  geo_Lake.Province = Province.Name
```

Figure 5.1: Desired schema mapping (SQL-)query generating the target schema in Table 5.1

is non-trivial, as it requires data analysts to 1) be familiar with the schema mapping query language (e.g., SQL) and 2) have a deep understanding of both the source database schemas and the target schema. To alleviate this problem, recent research [12, 50, 72, 84, 93, 105] has proposed a *sample-driven* or *example-based* approach to simplify schema mapping for the end user in the hope that users can describe their desired schema mapping by providing *example data records*—a few data records in the target schema.

Consider a quest to find all lakes, their areas and the states they belong to and the state areas from MONDIAL [69]—a relational geography dataset integrated from a number of data sources. The expected SQL query to obtain such a table is shown in Figure 5.1. With a sample-driven schema mapping system, the user only needs to provide a few data records as shown in Table 5.1.

Problems — While using examples to describe the target schema can potentially impose a lower requirement for non-expert users, an implicit assumption of *example correctness and consistency* behind these techniques may impede them from being effective in many use cases. Due to this requirement, the user has to make sure the data values in the provided examples are not only *correct* but also *precisely consistent* with the database in order for their algorithms to be functional (demonstrated in Section 5.2). This poses a pragmatic challenge for end users without perfect domain knowledge.

The issue itself cannot be easily addressed. First, auto-completion is limited to fixing user typos without considering factual errors. Users cannot simply check the database for the actual values because without a deep knowledge of the source database schema, it can be difficult to locate the schema elements of interest where the values exist. Second, acquiring exact data values from external data sources, such as Google and Wikipedia, may also fail due to the common problem of *data value-level inconsistency*: the value found in an external data source for the same element may be inconsistent to that in the source database.

Proposal — Instead of relaxing the correctness and consistency assumption, we increase the expressivity of the example description language for end users. In addition to exact values, the user can choose from an enriched set of constraints, including disjunctions of possible values and value ranges, to describe each cell. We denote these constraints as *multiresolution constraints*. In this case, the likelihood that the user mistakenly specifies

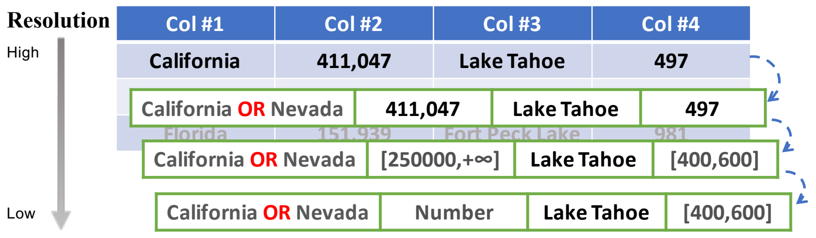


Figure 5.2: Examples at various resolutions to describe the target schema

incorrect or inconsistent examples will likely drop. Figure 5.2 exhibits a few different ways to specify the first record in Table 5.1 at various resolutions.

Once the user describes the target schema using these multiresolution constraints, we can synthesize the schema mapping queries matching the constraints.

Risks and Challenges — As multiresolution constraints are more relaxed than traditional sample-driven constraints, supporting schema mapping in this setting poses the risk of *inefficiency*: the time spent to synthesize satisfying candidate queries may significantly increase, forcing the user to wait longer, possibly an unacceptably long time.

This inefficiency arises due to two reasons. First, the complexity of the search space is $\mathcal{O}(n^d)$, where n is number of matching columns in the source database for a single column in the target schema, and d is the number of columns in the target schema. Allowing more relaxed constraints will substantially increase n, making the search space grow polynomially. Second, validating potential schema mapping queries against user constraints requires issuing expensive SQL queries on the database, which can be time-consuming.

Our Approach — We adapt an exploratory search approach used in [84] to quickly find a space of candidate solutions for our proposed workload and filter-based technique from [93] to validate the candidate queries. The validation process involves issuing SQL queries to the source database and creates a major performance bottleneck in many sample-driven schema mapping systems. To reduce the number of validations performed and improve the validation efficiency, we propose a *validation scheduling strategy guided by a probabilistic relational model*—BN-Greedy. In our experiments, the BN-Greedy strategy achieves a verification workload reduction of up to $\sim$ 69% and a runtime reduction of up to $\sim$ 30% compared to the best baseline strategy.

Organization — After motivating our problem with an example in Section 5.2, we summarize our contributions as follows:

1. We formally define the multiresolution constraint and the problem of sample-driven schema mapping with such constraints. (Section 5.3)
2. We propose a query validation scheduling strategy based on the Bayesian model to reduce the overall validation overhead (Section 5.4.2) in a filter-based query verification technique we adopt.

We experimentally evaluate the effectiveness of our proposed verification scheduling strategy in Section 5.6 and explore related work in Section 5.7.

5.2 Motivating Example

Mondial is a relational geography dataset, integrated from a number of data sources. Sharon, a graduate student from the department of Earth Science, wants to study the ground water availability for each state and needs to create a list of all lakes, their areas, and the states they belong to along with the state areas, using this data set.

Initially, Sharon chooses to use MWeaver [84], a classic sample-driven schema mapping system. To describe her desired table on MWeaver, Sharon needs to provide a few complete example data records from the table she desires. Initially, Sharon inputs “Lake Tahoe” as the first example. Although Sharon knows that Lake Tahoe is at the boundary of California and Nevada, she is not sure which state Lake Tahoe actually belongs to in Mondial and she wants to check the database to make sure. She finds a relation called “Lake” in Mondial and thinks it appears likely to contain this information; but she is disappointed to find that it in fact does not. As Sharon is not familiar with the database schema of Mondial, she cannot immediately figure out where else to find this information in the database. As a result, she simply puts “California” in the first cell as her best guess. She searches Wikipedia and finds the area of California is $423970km^2$ (it is $411047km^2$ on Mondial), so she puts this number in the second cell. In terms of the area of Lake Tahoe, Sharon only vaguely knows that it is between 400 and 600 km^2. She finds that the area is $490km^2$ on Wikipedia and puts this number in the fourth cell (the area is in fact $497km^2$ on Mondial). Not surprisingly, MWeaver fails to find a matching schema mapping query and returns an empty result indicating that the example Sharon provides may be incorrect. In fact, there are mistakes in the second and fourth cell and no query exists to provide this record.

In contrast, with our proposed system, Prism, Sharon has more options to specify examples at an imperfect resolution. For the state information, Sharon inputs two possible values, “California” and “Nevada”, and for the state area Sharon knows that both states are at least $250000km^2$, so she puts “$[250000, \infty]$” for the second cell. For area, she inputs a value range of “$[400, 600]$”. Figure 5.2 demonstrates this new example.

With the new example, Prism immediately finds the exact schema mapping query shown in Figure 5.1 as Sharon desires.

5.3 Problem Statement

In this project, we consider the source database, which is a relational database $\mathcal{D}$ with a *schema graph* $\mathcal{G}$ and d relations $\{\mathcal{R}_1, \ldots, \mathcal{R}_d\}$. In the schema graph $\mathcal{G} = (V, E)$, the vertices V correspond to the relations in $\mathcal{D}$ and edges E correspond to foreign/primary key constraints.

The objective is to derive a desired schema mapping query $\hat{q}$, which produces the target schema $\mathcal{T}$ of t projected columns/attributes on the source database $\mathcal{D}$. The end user offers a query $\mathcal{E}$ to describe $\mathcal{T}$, which is formalized as follows.

Definition 5.1 (User Input for Sample-driven Schema Mapping)**.** *The user input $\mathcal{E}$ is composed of a set of rows $\{\mathcal{E}_1, \ldots, \mathcal{E}_n\}$ denoting n example data records in the result table $\hat{q}(\mathcal{D})$. Each row $\mathcal{E}_i$ is comprised of a sequence of cells $\{\mathcal{E}_{i1}, \ldots, \mathcal{E}_{ik}\}$, where $\mathcal{E}_{ij}$ mapped to the j-th column in $\mathcal{T}$.*

When the user provides an exact value for a cell, we say the cell constraint is in the *perfect resolution*. However, when a precise cell value is not available, the constraint describing this cell needs to be in a "lower" resolution, such as an interval for ordinal data or a set of values for categorical data. In this project, we propose *multiresolution constraints* as a unified representation for cell values of various resolutions in the user input for sample-driven schema mapping. In this project, we focus on slightly relaxing the "exact value" requirement in traditional schema mapping (not changing it) at the cell level and limit the representation of multiresolution constraints to include cell values, value ranges and data types. Supporting other forms of low-resolution constraints, such as semantics and data types at the column level, is a possible direction to extend the current work and will be our future work. However, they may make assumptions different from that in the previous schema mapping research, and are hence, not examined in the current project.

We formally define the multiresolution constraint as follows:

Definition 5.2 (Multiresolution Constraint)**.** *A multiresolution constraint is a* disjunction *or* conjunction *of* atomic constraints $\{p_1, \ldots, p_k\}$*, where an atomic constraint* p *can be*

- *an expression describing a textual or numerical value,*
- *an expression describing a numerical range,*
- *an expression describing the data type of the cell (number, text or datetime),*
- *"$\star$", indicating that anything can be a match for this cell.*

The syntax for multiresolution constraints is formally presented in Figure 5.3.

Given user input $\mathcal{E}$ with multiresolution constraints, discovering the desired schema mapping query $\hat{q}$, is to find a query that *matches* $\mathcal{E}$, which is formalized as follows.

Definition 5.3 (Query Matching User Input Examples)**.** *Given user input* $\mathcal{E}$*, a query* q*, and database* $\mathcal{D}$*,* q *matches* $\mathcal{E}$ *if* $\forall \mathcal{E}_i = \{\mathcal{E}_{i1}, \ldots, \mathcal{E}_{ik}\} \in \mathcal{E}$*, there exists a record* $r = \{\mathcal{E}_1, \ldots, \mathcal{E}_k\} \in q(\mathcal{D})$ *such that* $\forall \mathcal{E}_j \in r$*: 1)* $\mathcal{E}_j$ *matches one of the atomic constraints in* $\mathcal{E}_j$ *when* $\mathcal{E}_j$ *is a disjunction of constraints, or 2)* $\mathcal{E}_j$ *matches all atomic constraints in* $\mathcal{E}_j$ *when* $\mathcal{E}_j$ *is a conjunction of constraints.*

$$
\begin{aligned}
\langle\text{mulres-cstr}\rangle &:= \langle\text{atom-cstr}\rangle \vee \langle\text{atom-cstr}\rangle \vee \ldots \\
&\mid \langle\text{atom-cstr}\rangle \wedge \langle\text{atom-cstr}\rangle \wedge \ldots \\
&\mid \star \\
\langle\text{atom-cstr}\rangle &:= \text{“data-value”}\ \langle\text{op}\rangle\ \textit{const} \\
&\mid \text{“data-range”}\ \langle\text{op}\rangle\ \langle\text{range}\rangle \\
&\mid \text{“data-type”}\ \langle\text{op}\rangle\ \langle\text{type}\rangle \\
\langle\text{range}\rangle &:= [\langle\text{val}\rangle, \langle\text{val}\rangle] \\
\langle\text{type}\rangle &:= \text{“text”} \mid \text{“num”} \mid \text{“datetime”} \\
\langle\text{op}\rangle &:= \text{“}{=}\text{”} \mid \text{“}{\neq}\text{”} \\
\langle\text{val}\rangle &:= \textit{const} \mid -\infty \mid +\infty
\end{aligned}
$$

Figure 5.3: Multiresolution constraint for a single cell

Following previous research [84, 93], we derive only Project-Join (PJ) SQL queries in this project. As a small set of examples usually do not uniquely identify the desired query $\hat{q}$, we return all PJ queries satisfying $\mathcal{E}$. While different ranking schemes have been proposed in previous research, reordering the returned result is beyond the interest of our work.

We formalize the query discovery problem as follows:

Problem 5.1. *Given user input $\mathcal{E}$, where each cell is a multiresolution constraint, and the source database $\mathcal{D}$, find the set of Project-Join queries, $\mathcal{Q}_o$, such that $\forall q \in \mathcal{Q}_o$, q matches $\mathcal{E}$.*

5.4 Query Synthesis Algorithms

We walk through a classic query discovery framework suitable for our problem in Section 5.4.1 and present a new scheduling technique to speed up the query validation process in Section 5.4.2.

5.4.1 Preliminaries

To find a complete set of satisfying PJ queries matching $\mathcal{E}$, an efficient two-phase "explore-and-verify" query discovery paradigm used by [93] is adopted:

1. Quickly reduce the search space to a set of candidate queries.
2. Efficiently verify the candidate queries by validating their sub-queries noted as *filters*.

Find Candidate Queries — Actual definitions of candidate queries and the techniques for finding them can vary for different query synthesis tasks as long as 1) candidate queries are a superset of all satisfying queries returned as the system output, 2) search process is efficient. Following the same practice used by [84, 93], we discover candidate columns in two steps:

1. Locate *candidate columns* for each column in the target schema $\mathcal{T}$.
2. Discover *candidate queries* by finding paths connecting candidate columns on the schema graph $\mathcal{G}$.

Candidate Column. Candidate columns, $\mathcal{A}_c$, are columns in the source database $\mathcal{D}$ that can potentially be matched to the columns in $\mathcal{T}$.

Assuming the column j in the user input $\mathcal{E}$ contains cells denoted as $\{\mathcal{E}_{1j}, \dots, \mathcal{E}_{nj}\}$. By the definition used in [93], the candidate columns $\mathcal{A}_c[j]$ for the column j in $\mathcal{T}$ is as follows:

$$\mathcal{A}_c[j] = \bigcap_{i=1}^{n} \mathcal{A}_c[\mathcal{E}_{ij}], \tag{5.1}$$

where $\mathcal{A}_c[\mathcal{E}_{ij}]$ denotes the columns in $\mathcal{D}$ containing the cell $\mathcal{A}_c[\mathcal{E}_{ij}]$.

In traditional sample-driven schema mapping, an inverted index $\mathcal{I}$ is usually leveraged to enable a rapid search for $\mathcal{A}_c[\mathcal{E}_{ij}]$, where all cell $\mathcal{E}_{ij}$ values are perfect-resolution ($\mathcal{E}_{ij}$ is a single atomic constraint). Hence, $\mathcal{A}_c[\mathcal{E}_{ij}] = \mathcal{I}_c[\mathcal{E}_{ij}]$. However, in our setting which includes low-resolution cell constraints, $\mathcal{I}$ is not immediately helpful as it only returns the locations of a single value, not a disjunction (or conjunction) of possible values (or ranges). To be able to benefit from using an inverted index in candidate column search, we use the following property.

Theorem 5.1. *Given cell constraints $\{\mathcal{E}_{1j}, \dots, \mathcal{E}_{nj}\}$ in column j of the user input $\mathcal{E}$, where each cell constraint $\mathcal{E}_{ij}$ is a multiresolution constraint, and the inverted index $\mathcal{I}$, the candidate columns for column j are:*

$$\mathcal{A}_c[j] = \begin{cases} \bigcap_{i=1}^{n} \mathcal{A}_c[\bigvee_{d=1}^{m} p_d] = \bigcap_{i=1}^{n} \Big(\bigcup_{d=1}^{m} \mathcal{I}[p_d] \Big), & \textit{if } \mathcal{E}_{ij} = \bigvee_{d=1}^{m} p_d \\ \bigcap_{i=1}^{n} \mathcal{A}_c[\bigwedge_{d=1}^{m} p_d] = \bigcap_{i=1}^{n} \Big(\bigcap_{d=1}^{m} \mathcal{I}[p_d] \Big), & \textit{if } \mathcal{E}_{ij} = \bigwedge_{d=1}^{m} p_d \end{cases} \tag{5.2}$$

The property can be proved by induction as below.

Proof. In the first scenario, where $\mathcal{E}_{ij} = \bigvee_{d=1}^{m} p_d$, let $m = 1$,

$$\mathcal{A}_c[\bigvee_{d=1}^{1} p_d] = \mathcal{A}_c[p_1] = \mathcal{I}[p_1] = \bigcup_{d=1}^{1} \mathcal{I}[p_d]. \tag{5.3}$$

Hence, the first half of Theorem 5.2 is true when $m = 1$.

Assuming when $m = k$,

$$\mathcal{A}_c[\bigvee_{d=1}^{k} p_d] = \bigcup_{d=1}^{k} \mathcal{I}[p_d]. \tag{5.4}$$

is true. Let $m = k + 1$,

$$\mathcal{A}_c[\bigvee_{d=1}^{k+1} p_d] = \mathcal{A}_c[\bigvee_{d=1}^{k} p_d \vee p_{k+1}]. \tag{5.5}$$

Recall the definition of $\mathcal{A}_c[\mathcal{E}]$ is the set of all columns containing at least a cell matching the constraint $\mathcal{E}$. Given two distinct constraints $\mathcal{E}_1$ and $\mathcal{E}_2$, it is axiomatic that the union of 1) columns containing at least a cell matching $\mathcal{E}_1$ and 2) columns containing at least a cell matching $\mathcal{E}_2$ is equivalent to the columns containing at least a cell matching $\mathcal{E}_1$ or $\mathcal{E}_2$, noted as

$$\mathcal{A}_c[\mathcal{C}_1] \cup \mathcal{A}_c[\mathcal{C}_2] = \mathcal{A}_c[\mathcal{C}_1 \vee \mathcal{C}_2] \tag{5.6}$$

Based on (5.4) and (5.6), continue with (5.5)

$$\begin{aligned}\mathcal{A}_c[\bigvee_{d=1}^{k} p_d \vee p_{k+1}] &= \mathcal{A}_c[\bigvee_{d=1}^{k} p_d] \cup \mathcal{A}_c[p_{k+1}] \\ &= \bigcup_{d=1}^{k} \mathcal{I}[p_d] \cup \mathcal{I}[p_{k+1}] = \bigcup_{d=1}^{k+1} \mathcal{I}[p_d]\end{aligned}$$

Thus, the first half of Theorem 5.2 is true when $m = k+1$. By the principle of induction, the first half of Theorem 5.2 must hold for $m \in \mathbb{N}$. The proof of the second half of Theorem 5.2 is same as that of the first half. Therefore, we have proved the correctness of Theorem 5.2. □

Candidate Query. A *candidate query* (CQ), $\mathcal{Q}_c = \langle \mathcal{J}_c, \mathcal{P}_c, \mathcal{M}_c \rangle$, is a Project-Join (PJ) query, where projected columns $\mathcal{P}_c = \{a_1, \dots, a_{col(\mathcal{T})}\}$ ($a_j \in \mathcal{A}_c[j]$), join path $\mathcal{J}_c$ is a *connected subgraph* (tree) of $\mathcal{G}$, and $\mathcal{M}_c$ is a one-to-one mapping between column a_j and column j in $\mathcal{T}$.

To search for a CQ given $\mathcal{A}_c$ and $\mathcal{G}$, we use an explorative-search algorithm similar to the TPW algorithm proposed in [84], which recursively "weaves" short join paths to formulate longer join paths in $\mathcal{G}$ covering more columns in $\mathcal{T}$ until a complete CQ covering all columns in $\mathcal{T}$ is found.

Query Verification Through Filters — Although each column in each CQ discovered thus far matches its corresponding column in $\mathcal{T}$, it is uncertain if each CQ matches the user input $\mathcal{E}$ (Definition 5.3) from a record-wise perspective because joins have yet to be evaluated.

Iteratively executing these CQs on $\mathcal{D}$ and checking their query result against $\mathcal{E}$ can be expensive. We leverage the concept of *filters* and two related properties—Filter-Query Dependency and Inter-Filter Dependency—proposed in [93] to efficiently validate CQs (prune invalid CQs).

Filters and Filter-Query Dependency. A *filter* is a succinct substructure of a candidate query used to partially verify a row $\mathcal{E}_i \in \mathcal{E}$. Suppose $\mathcal{E}_1$ in Figure 5.4 is a row from the user input $\mathcal{E}$. To verify if a candidate query $\mathcal{Q}_c$ matches $\mathcal{E}_1$, a subsection of $\mathcal{Q}_c$ in the area circled by dotted lines can be taken as a new query $\mathcal{Q}_f$ to verify the last two columns of $\mathcal{E}_1$ at a fairly low cost. If the query result of $\mathcal{Q}_f$ does not contain the last two columns of $\mathcal{E}_1$[6], $\mathcal{Q}_c$ will not include $\mathcal{E}_1$ in its query result either, and is thereby *invalid* for $\mathcal{E}$ ($\mathcal{Q}_c$ does not match $\mathcal{E}$). In this case, $\mathcal{Q}_f$ is a filter for $\mathcal{Q}_c$, and this relationship between $\mathcal{Q}_f$ and $\mathcal{Q}_c$ is referred to as *Filter-Query Dependency*, denoted as $\mathcal{Q}_f \succ_{-} \mathcal{Q}_c$.

Moreover, multiple candidate queries that have overlapping join paths will also have a subset of filters in common, which means if any filter in this subset fails, all CQs sharing this filter will be pruned. To that end, pruning invalid CQs through filter evaluation can potentially reduce more system execution time.

[6]For simplicity, we say $\mathcal{Q}_f$ *fails*.

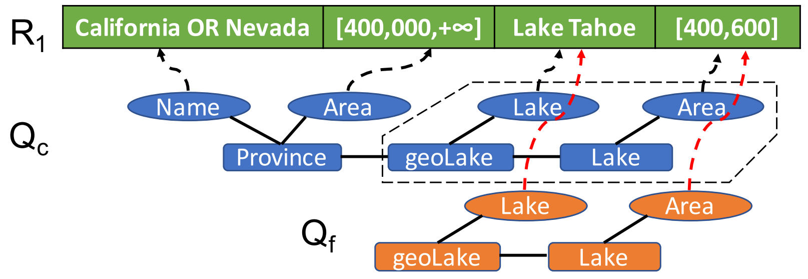


Figure 5.4: An example of a filter

Inter-Filter Dependency. More than one filter can be derived from a CQ, and if one is a subgraph (subsection) of another, a similar property exists between them as well. Suppose filter $\mathcal{Q}_{f_1}$ is a subgraph of filter $\mathcal{Q}_{f_2}$, if $\mathcal{Q}_{f_1}$ fails to match $\mathcal{E}$ (for its pertinent columns), $\mathcal{Q}_{f_2}$ fails too, denoted as $\mathcal{Q}_{f_1} \succ_{-} \mathcal{Q}_{f_2}$. Conversely, if $\mathcal{Q}_{f_2}$ matches $\mathcal{E}$, $\mathcal{Q}_{f_1}$ succeeds too, denoted as $\mathcal{Q}_{f_2} \succ_{+} \mathcal{Q}_{f_1}$. Therefore, a filter is not limited to just verifying (or pruning) a CQ but also other filters sharing the same (sub-)structures. We refer to this property as *Inter-filter Dependency*.

Due to the above two properties between filters and CQs, [93] proposes Algorithm 6 to efficiently verify (prune) the candidate queries. It first derives all filters for the given set of candidate queries $\mathcal{Q}_c$ and user examples $\mathcal{E}$. Next, it iteratively evaluates every filter on $\mathcal{D}$, and use each result to prune CQs and filters accordingly.

Challenges — Low-resolution cell constraints introduce more ambiguity into the user input $\mathcal{E}$. In this case, the number of candidate queries discovered in the first phase may significantly increase as more columns in $\mathcal{D}$ are admitted as candidate columns (demonstrated in Section 5.6.2). This poses a potential performance risk to the second phase of candidate query verification/validation, i.e., Algorithm 6, which takes the bulk of system

Algorithm 6: Validate candidate queries using filters

```
Data: Candidate queries $\mathcal{Q}$, user input examples $\mathcal{E}$
Result: Set of queries matching $\mathcal{E}$
1  $\mathcal{F} \leftarrow$ GetAllFilters($\mathcal{Q}$, $\mathcal{E}$);
2  while $\mathcal{F} \neq \emptyset$ do
3      $\mathcal{Q}_f \leftarrow$ SelectNextFilter($\mathcal{F}$);
4      Evaluate $\mathcal{Q}_f$ on $\mathcal{D}$;
5      if $\mathcal{Q}_f$ fails then
6          Remove $\{\mathcal{Q}_c | \mathcal{Q}_c \in \mathcal{Q}, q_f \succ_- \mathcal{Q}_c\}$ from $\mathcal{Q}$;
7          Remove $\{\mathcal{Q}_{f'} | \mathcal{Q}_{f'} \in \mathcal{F}, q_f \succ_- \mathcal{Q}_{f'}\}$ from $\mathcal{F}$;
8      else
9          Remove $\{\mathcal{Q}_{f'} | \mathcal{Q}_{f'} \in \mathcal{F}, q_f \succ_+ \mathcal{Q}_{f'}\}$ from $\mathcal{F}$;
10     Remove $\mathcal{Q}_f$ from $\mathcal{F}$;
11 Return $\mathcal{Q}$;
```

run time.

Algorithm 6 can also be separated into two parts: 1) filter discovery (line 1), and 2) filter validation (line 2-10). In Section 5.4.2, we present a Bayesian network-based filter scheduling strategy, BN-Greedy, as a solution to **SelectNextFilter** (line 3 in Algorithm 6)—a critical base component of Algorithm 6—to improve the efficiency of the filter validation process.

5.4.2 Filter Scheduling

The presence of low-resolution constraints in the user input introduces more uncertainties in candidate query discovery and places a higher burden on the follow-up candidate query validation phase. Since candidate query validation requires querying the database system frequently, it can be a major performance bottleneck. Therefore, it is reasonable to pursue a strategy associated with our proposed workload in order to maintain the interactive speed of the schema mapping system.

In each iteration of Algorithm 6, a filter will be chosen by **SelectNextFilter** (line 3), issued on the source database (line 4) and the validation result is used to prune other related filters/candidate queries following the Filter-Query Dependency and Inter-Filter Dependency properties (line 6-9). Once the user input examples are given, the set of candidate queries, and subsequently, the set of filters to validate are also fixed, the act of *arranging the validation of filters*, or *filter scheduling* (i.e., function **SelectNextFilter** in Algorithm 6 line 3), is essentially a critical base component that impacts the filter validation and overall system efficiency. In this project, we consider this problem of *filter scheduling* as the key problem to resolve, formalized as follows.

Problem 5.2 (Filter Scheduling)**.** *Given a set of filters $\mathcal{F}$, generated from the CQ set, find a sequence for these filters $\hat{\mathcal{F}} = \langle \mathcal{Q}_{f_i} \mid i \in [1, |\mathcal{F}|] \rangle$, such that the overall time cost of validating $\hat{\mathcal{F}}$ is minimal.*

This problem is proven to be NP-hard [93]. **In this project, we use Algorithm 6 for candidate query validation and propose a Bayesian network-based strategy, BN-Greedy, for SelectNextFilter in Algorithm 6, which is built upon a baseline strategy Naïve-Greedy and achieves a better performance in validating filters with multiresolution constraints.**

Baseline Strategies

We first discuss two strategies used in previous work.

Shortest-First Strategy*.* By definition, a "shorter" filter (a filter with small number of projected columns verifying a small portion of a CQ) is cheaper to validate and tends to be shared by more CQs, leading to a higher pruning power when it fails. Conversely, a longer filter has a lower pruning power if it fails. [84] leverages this property and adopts a simple strategy of *always prioritizing validations of the shortest filter*. We call this strategy the *Shortest-First Strategy*.

Naïve-Greedy Strategy. The Shortest-First Strategy is most effective only when all filters are equally likely to fail. In reality, a shorter filter is less likely to fail compared to a longer filter, which undermines its pruning power. [93] formalizes this intuition as a *greedy strategy* balancing the pruning power and the probability of failure when choosing a filter, shown as follows.

$$\hat{\mathcal{Q}}_f = \arg\max_{\mathcal{Q}_f \in \mathcal{F}} \frac{\mathbb{E}(W(\mathcal{Q}_f|\mathcal{F}))}{cost(\mathcal{Q}_f)} \tag{5.7}$$

$$\mathbb{E}(W(\mathcal{Q}_f|\mathcal{F})) = (1 - P_{\mathcal{Q}_{f_-}})W_+(\mathcal{Q}_f|\mathcal{F}) + P_{\mathcal{Q}_{f_-}} W_-(\mathcal{Q}_f|\mathcal{F}) \tag{5.8}$$

$\mathbb{E}(W(\mathcal{Q}_f|\mathcal{F}))$ denotes the *expected pruning power* of a filter $\mathcal{Q}_f$, where 1) $P_{\mathcal{Q}_{f_-}}$ denotes the probability that $\mathcal{Q}_f$ *fails*, 2) $W_+(\mathcal{Q}_f|\mathcal{F})$ and $W_-(\mathcal{Q}_f|\mathcal{F})$ denote the number of filters in $\mathcal{F}$ that are respectively pruned when $\mathcal{Q}_f$ *succeeds* and *fails*. In each iteration of Algorithm 6, this greedy algorithm chooses the filter with the highest *expected pruning power per unit of cost*.

While $W_+(\mathcal{Q}_f|\mathcal{F})$ and $W_-(\mathcal{Q}_f|\mathcal{F})$ are determined once f and $\mathcal{F}$ are given, *estimating $P_{\mathcal{Q}_{f_-}}$ remains a problem to resolve* which is formalized as follows.

Problem 5.3 (Filter Failure Probability (FFP) Estimation). *Given a $\mathcal{Q}_f$ and the source database $\mathcal{D}$, estimate the probability that $\mathcal{Q}_f$ fails on $\mathcal{D}$, noted as $P_{\mathcal{Q}_{f_-}}$.*

Instead of assigning the same likelihood of failure/success for all filters in the Shortest-First Strategy, [93] assumes the failure probability of a filter is proportional to its number of projected columns, and proposes a naïve model for the filter failure probability as $P_{\mathcal{Q}_{f_-}} = \bar{p} \cdot \frac{n_f}{Col(\mathcal{T})}$, where n_f is the number of projected columns in the filter $\mathcal{Q}_f$, $Col(\mathcal{T})$ is the total number of columns in the target schema $\mathcal{T}$ (equal for all filters), and $\bar{p}$ is a pre-defined constant. We refer to this as the *Naïve-Greedy strategy.*

Bayesian Network-based Greedy Strategy (BN-Greedy)

When multiresolution constraints are included in user-provided examples, both the Shortest-First and Naïve-Greedy strategies fail to account for the impact of the *actual values of the multiresolution constraints* within each filter and *inter-column correlations*, as illustrated by the following example.

Example 5.1. *Among the filters created for our motivating example, suppose both filters $\mathcal{Q}_{f_1}$ and $\mathcal{Q}_{f_2}$ verify if any row contains the value "California" with a property of value ranging from "400" to "600". $\mathcal{Q}_{f_1}$ joins the table "Lake" and "geo_Lake" and assumes "Province" (name) and "Area" (in km^2) are two corresponding attributes to project. $\mathcal{Q}_{f_2}$ only projects a single table on "Province" (name) and "Population". Since the schema mapping system has access to the source database, it can have a vague knowledge of the underlying data, it is definitely certain that California's population is far more than 600. Therefore, we have $P_{\mathcal{Q}_{f_{1-}}} \ll P_{\mathcal{Q}_{f_{2-}}}$ in reality. In this case, it is reasonable to prioritize the validation of $\mathcal{Q}_{f_2}$, because $\mathbb{E}(W(\mathcal{Q}_{f_1}|\mathcal{F})) < \mathbb{E}(W(\mathcal{Q}_{f_2}|\mathcal{F}))$, which means $\mathcal{Q}_{f_2}$ will in fact have a larger pruning power than $\mathcal{Q}_{f_1}$.*

However, Shortest-First assigns an equal priority to $\mathcal{Q}_{f_1}$ and $\mathcal{Q}_{f_2}$, because they have the same length, i.e., $n_{\mathcal{Q}_{f_1}} = n_{\mathcal{Q}_{f_2}}$. On the other hand, Naïve-Greedy Strategy prefers $\mathcal{Q}_{f_1}$ over $\mathcal{Q}_{f_2}$ because it also assumes $P_{\mathcal{Q}_{f_{1-}}} = P_{\mathcal{Q}_{f_{2-}}}$ given $n_{\mathcal{Q}_{f_1}} = n_{\mathcal{Q}_{f_2}}$, and when $W_-(\mathcal{Q}_{f_1}|\mathcal{F}) > W_-(\mathcal{Q}_{f_2}|\mathcal{F})$ is always true, it concludes that $\mathbb{E}(W(\mathcal{Q}_{f_1}|\mathcal{F})) > \mathbb{E}(W(\mathcal{Q}_{f_2}|\mathcal{F}))$.

Worse, let's assume there is a third constraint $\mathcal{Q}_{f_3}$ with the same projected columns verifying if the query constraints "Nevada" with a property of value "$\star$", which means $\mathcal{Q}_{f_3}$ is the least likely to fail, i.e., $P_{\mathcal{Q}_{f_{3-}}} \ll P_{\mathcal{Q}_{f_{1-}}} \ll P_{\mathcal{Q}_{f_{2-}}}$, since "$\star$" can match anything. However, both Shortest-First and Naïve-Greedy will once again make poor decisions and prioritize $\mathcal{Q}_{f_1}$ or $\mathcal{Q}_{f_2}$.

Here, we present the BN-Greedy strategy, built upon Naïve-Greedy, replacing its heuristic way to compute filter failure probabilities (FFP) (i.e., $P_{\mathcal{Q}_{f_-}}$ in (5.8)) with a Bayesian networks method.

By definition, a filter $\mathcal{Q}_f = \langle \mathcal{J}_f, \mathcal{P}_f, \mathcal{M}_f, \mathcal{E}, \mathcal{M}_c \rangle$ can be written as a Select-Project-Join (SPJ) query, where $\mathcal{P}_f$ are projected columns corresponding to a subset of columns in the target schema $\mathcal{T}$, $\mathcal{J}_f$ are the join conditions, $\mathcal{E}$ and $\mathcal{M}_c$ constitutes selections. Estimating the FFP of $\mathcal{Q}_f$ is equivalent to predicting the likelihood that the result set of $\mathcal{Q}_f$ is empty, a problem similar to *selectivity estimation* for a SPJ query.

The query result size along with an intermediate result—the joint frequency (probability)—required to compute the query result size in many selectivity estimation techniques is *valuable information* for computing the FFP. That is, the transformation from the selectivity estimation probability to the Bernoulli variable $\mathcal{Q}_f$ can be captured as

$$P_{\mathcal{Q}_{f_-}} = p(|\mathcal{Q}_f(\mathcal{D})| < 1), \tag{5.9}$$

where $|\mathcal{Q}_f(\mathcal{D})|$ denotes the size of the filter $\mathcal{Q}_f$ on the database $\mathcal{D}$.

Below, we first show a Bayesian network-based approach estimating the query result size of a filter, and then present how to perform the above transformation to finally estimate FFP.

Estimate filter result size using Bayesian models

Result size of filters with exact constraints in a single relation — First, assume that a filter $\mathcal{Q}_f$ validates data only from a single relation R ($\mathcal{J}_f = \emptyset$), also $\mathcal{P}_f = \{\mathcal{A}_1, \dots, \mathcal{A}_n\}$, $\mathcal{E} = \{c_1^m, \dots, c_n^m\}$ and $\mathcal{M}_c = \{\mathcal{A}_1 \to c_1^m, \dots, \mathcal{A}_n \to c_n^m\}$. The relational algebra expression of $\mathcal{Q}_f$ is

$$\mathcal{Q}_f = \pi_{\mathcal{A}_1,\dots,\mathcal{A}_n}(\sigma_{\mathcal{A}_1=c_1^m,\dots,\mathcal{A}_n=c_n^m}(R)) \tag{5.10}$$

The expected result size of $\mathcal{Q}_f$, $\mathbb{E}[|\mathcal{Q}_f(\mathcal{D})|]$, becomes

$$\mathbb{E}[|\mathcal{Q}_f(\mathcal{D})|] = P(\mathcal{A}_1 = c_1^m, \dots, \mathcal{A}_n = c_n^m) \cdot |R|, \tag{5.11}$$

where P is the joint probability mass function (PMF)[7] of the source database $\mathcal{D}$, and $|R|$ is the table size.

Chain rule of probability dictates that the joint probability can be computed using *conditional probabilities* and *marginal probabilities* [92]. Applying it to (5.11), we have

$$\begin{aligned}\mathbb{E}[|\mathcal{Q}_f(\mathcal{D})|] =& P(\mathcal{A}_1 = c_1^m | \mathcal{A}_2 = c_2^m, \dots, \mathcal{A}_n = c_n^m)\cdot \\ & P(\mathcal{A}_2 = c_2^m | \mathcal{A}_3 = c_3^m, \dots, \mathcal{A}_n = c_n^m)\cdot \\ & \dots \cdot P(\mathcal{A}_n = c_n^m) \cdot |R|\end{aligned} \tag{5.12}$$

Pre-computing all above conditional probabilities among different attributes in a database would give us a *perfect* P, in which case both $\mathbb{E}[|\mathcal{Q}_f(\mathcal{D})|]$ and the FFP of $\mathcal{Q}_f$ would be achieved at a high accuracy (in fact, deterministic). Yet, this can be extremely expensive and thereby infeasible in a real-world database.

[7]or probability density function (PDF) if attributes are continuous

Instead, we use Bayesian networks (BN) to model an *approximated* joint probability distribution of a database under the *conditional independence assumption* and to estimate $\mathbb{E}[|\mathcal{Q}_f(\mathcal{D})|]$.

Actually constructing a Bayesian network based on a single relation (equivalent to a standalone dataset) has been extensively studied in the past [13, 15, 21] and we use the classic K2 algorithm [21] with the default configuration offered by Weka [40] (version 3.8.0) in our actual implementation for model construction. Other algorithms are also possible.

Given a Bayesian network learned for R, (5.12) becomes

$$\mathbb{E}[|\mathcal{Q}_f(\mathcal{D})|] = \prod_{i=1}^{n} P(\mathcal{A}_i = c_i^m | \text{parents}(\mathcal{A}_i = c_i^m)) \cdot |R| \tag{5.13}$$

Result size of filters with exact constraints across multiple relations — So far, $\mathcal{Q}_f$ is limited to project data from only one relation. However, a filter may span over multiple relations in a database. [28] extends the definition of the BN to Probabilistic Relational Model (PRM) by introducing *join indicator* variables between tables in the database. We here adapt the concept of join indicator to address the problem of constructing a BN in a relational database setting to estimate the FFP we propose in the project.

A *join indicator* is a binary variable that helps to model correlations between columns from two relations that can be joined by a PK-FK.

Now, suppose $\mathcal{Q}_f$ includes a join between two relations R_1 and R_2. Then $\mathcal{Q}_f$ becomes

$$\mathcal{Q}_f = \pi_{\mathcal{A}_1,\ldots,\mathcal{A}_n}(\sigma_{\mathcal{A}_1=c_1^m,\ldots,\mathcal{A}_n=c_n^m}(R_1 \quad R_2)) \tag{5.14}$$

By (5.11)-(5.13), $\mathbb{E}[|\mathcal{Q}_f(\mathcal{D})|]$ becomes

$$\begin{aligned}\mathbb{E}[|\mathcal{Q}_f(\mathcal{D})|] &= P(\mathcal{A}_1 = c_1^m, \dots, \mathcal{A}_n = c_n^m, \cdot J_{R_1 R_2} = 1) \ \cdot |R_1||R_2| \\ &= \prod_{i=1}^{n} P(\mathcal{A}_i = c_i^m | \text{parents}(\mathcal{A}_i = c_i^m), J_{R_1 R_2} = 1) \cdot |R_1||R_2|\end{aligned} \tag{5.15}$$

Actually constructing a Bayesian network to estimate the joint probability in (5.15) is beyond our interest in this project and we use the same approach described in Algorithm 1 in [103], which generally involves two steps 1) construct a Bayesian network for each single relation (as we did before (5.13)), 2) model join indicators alone with two most correlated attributes of the joining relations involved. More implementation details can be found in Section 4 in [103].

Result size of filters with multiresolution constraints — So far, estimating the result size of a filter has been only focused on filters with only exact values (atomic predicate constraints). However, in this project, filters can alternatively be a disjunction (or conjunction) of values or "$\star$".

In a filter $\mathcal{Q}_f$, when a cell constraint $\mathcal{E}_{ij}$ is a disjunction of predicate constraints, i.e., $\mathcal{E}_{ij} = \bigvee_{d=1}^{m} p_d$, we can divide $\mathcal{Q}_f$ into a set of *mutually exclusive* filters with only atomic constraints, $\{\mathcal{Q}_{f_1}, \dots, \mathcal{Q}_{f_m}\}$, where $\mathcal{E}_{ij} = \{p_d\}, \mathcal{E}_{ij} \in \forall \mathcal{Q}_{f_d}, d = \{1, \dots, m\}$.

When $\{\mathcal{Q}_{f_1}, \dots, \mathcal{Q}_{f_n}\}$ are mutually exclusive, Probability Theory dictates that the joint probability of $\mathcal{Q}_f$ is

$$P(\mathcal{Q}_f) = \sum_{d=1}^{m} P(\mathcal{Q}_{f_d}), \tag{5.16}$$

and each of $P(\mathcal{Q}_{f_i})$ can be obtained by (5.15).

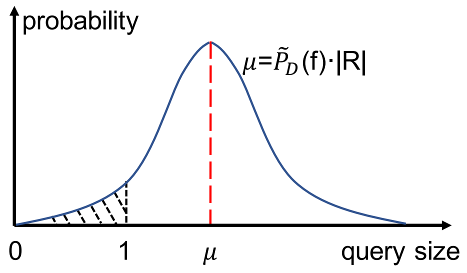


Figure 5.5: Possible distribution of the filter query size

Alternatively, if a cell constraint $\mathcal{E}_{ij}$ from a filter $\mathcal{Q}_f$ is specified as "$\star$" indicating that it can match anything, we borrow the same idea in above dividing $\mathcal{Q}_f$ into a set of *mutually exclusive* filters with atomic constraints, $\{\mathcal{Q}_{f_1}, \ldots, \mathcal{Q}_{f_{|\mathcal{A}_j|}}\}$, where $\mathcal{E}_{ij}$ in each $\mathcal{Q}_{f_d}$ is a unique value in the column $\mathcal{A}_j$, to which $\mathcal{E}_{ij}$ is mapped to. In this case, (5.16) and (5.15) can be leveraged to obtain $P(\mathcal{Q}_f)$.

Infer FFP using Bayesian models

Obtaining an *approximated* result size or joint probability of a filter is not sufficient. Our objective is to compute the probability that the result set is empty (result size is less than one).

If we could model the underlying probability distribution (possibly a bell-shaped distribution like Figure 5.5 with mean equal to $\widetilde{P}(\mathcal{Q}_f)\cdot|R|$), we would be able to estimate the FFP (the shaded area in Figure 5.5).

A Bayesian network is an approximation of the joint distribution of a dataset. Imprecisions occur in various steps in Bayesian network learning, such as structure learning, parameter learning. Accurately propagating all these imprecisions to the final result is non-trivial [53] and unnecessary, and thereby beyond the scope of our project. We here take a different approach.

We assume that the assumption of *row-independence* holds in the database $\mathcal{D}$: correlation does not exist across data entries in the database. In this case, given a random data instance in the dataset, the probability that it matches all constraints involved in the filter $\mathcal{Q}_f$ is $P(\mathcal{Q}_f)$. With that, to calculate the likelihood that the set of data matching $\mathcal{Q}_f$ is empty, we can model this problem as a *biased coin toss* problem where the number of total flips is $|\mathcal{R}|$, i.e., the table size, and the biased probability of "head" in each independent flip is $P(\mathcal{Q}_f)$.

Probability theorem dictates that the probability distribution of all possible outcomes (selectivity size) in tossing a biased coin follows the *Binomial distribution*. We use the results from the central limit theorem and the relationship between Binomial and Normal distribution [54] and, considering $|\mathcal{R}|$ to be a sufficiently large number, model the selectivity size distribution $B(|\mathcal{R}|, P(\mathcal{Q}_f))$ as

$$P(X) = \mathcal{N}(\mu = |\mathcal{R}| \cdot P(\mathcal{Q}_f),\ \sigma^2 = |\mathcal{R}| \cdot P(\mathcal{Q}_f) \cdot (1 - P(\mathcal{Q}_f)) \tag{5.17}$$

Now,using Equation 5.9 (Figure 5.5), the failure probability for a filter $\mathcal{Q}_f$ is $P_{\mathcal{Q}_{f\_}} = P(X < 1)$.

To use the standard Normal distribution, let us define the variable $Z = (X - \mu)/\sigma$. Then, using the Z-table, the failure probability (FFP) is

$$P_{\mathcal{Q}_{f\_}} = \Phi(\frac{1-\mu}{\sigma}) = \Phi\Big(\frac{1 - |\mathcal{R}| \cdot P(\mathcal{Q}_f)}{\sqrt{|\mathcal{R}| \cdot P(\mathcal{Q}_f) \cdot (1 - P(\mathcal{Q}_f))}}\Big) \tag{5.18}$$

5.5 Risks and Limitations

The effectiveness of our proposed technique depends on the fidelity of the source schema and user input, and also subjected to certain limitations in expressivity.

Accessibility of source schema $\mathcal{G}$ — We assume that the source schema $\mathcal{G}$ is provided, i.e., the primary/foreign keys constraints are already known (to our system, not to the end user). In situations where such information is not explicitly given, we can first apply a PK/FK constraint discovery mechanism [91].

Fidelity of user input $\mathcal{E}$ — Successfully discovering the desired schema mapping query $\hat{q}$ (assuming one exists) using our proposed solution is predicated on the *fidelity of the user input* $\mathcal{E}$: the examples provided in $\mathcal{E}$ must perfectly match a subset of the result set of $\hat{q}$. Yet, this is still a weaker assumption than that made in previous sample-based schema mapping research. Users who are able to use previous systems must be able to use ours, and even if they cannot provide exact examples they may still be able to contribute some vague knowledge on our system, which is aligned to the general expectation for our research.

Expressivity of the synthesized queries — The expressive power of the inferred schema mapping queries is limited to PJ queries in this project. Recent QRE research [23, 61, 102, 105] has shown some success in inferring SPJ queries with selection predicates. However, these works usually make a *closed-world* assumption on the input $\mathcal{E}$: the complete result set of the desired schema $\mathcal{T}$ needs to be provided. The closed-world assumption is usually applicable in a non-interactive situation where the problem is to reverse engineer a SPJ query when the complete query result can be obtained. However, in a human-in-the-loop setting, the user is expected to provide all the hints based on her knowledge. In this case, the closed-world assumption is much less realistic than an open-world assumption, which we make in this project, where the input can be a subset of the result set of the desired schema, usually a few instances. With the closed-world assumption, inferring correct select predicates or filters can be almost impossible with only a few example instances [93] because there can be almost infinite number of PJ queries with the satisfying selection predicates. For example, suppose the Table 5.1 contains all the examples provided by the user. In an open-world setting, as no information is given about what column/attribute a potential selection predicate could set be on, possible list of candidate predicates can include **LakeArea** < 1000, **LakeArea** < 2000, **LakeArea** < 3000 and so on. Therefore, we take a step back and only infer PJ queries in this project. Developing better forms of user constraints that are both 1) easy to provide for the user and 2) helpful for the system to discover the right selection predicates to enable the synthesis of SPJ queries will be our future work.

5.6 Experiments

Despite that our proposed multiresolution constraints for sample-driven schema mapping targets on reducing the requirement for user knowledge of the source data, it is still interesting to understand how the usability of the new schema mapping system changes from two perspectives: 1) system execution time, 2) user validation effort.

In this section, we present experiments to answer the following questions in the rest of this section.

- How does the *number of candidate queries* and *candidate query validation time* change in the presence of low-resolution constraints (without any optimization)?
- How does the system validation cost of the proposed Bayesian-model-

Name	# tables	# cols	# key pairs	Max # joins	Data types
MONDIAL	40	167	48	4	Numeric, String, Temporal
Financial	8	55	9	5	Numeric, String, Temporal
Hepatitis	7	26	6	2	String

Table 5.2: Dataset Details

based filter scheduling strategy compare with the baseline approach? (Section 5.6.3)

- How does the user validation effort change with low-resolution constraints? (Section 5.6.4)

5.6.1 Experimental Setup

Hardware and SQL Engines — We implemented our proposed schema mapping method along with different filter scheduling strategies in a prototype system called PRISM. For the underlying database systems, we used MySQL 5.5.59. All systems were running on a 16-core Intel Xeon server with 128 GB RAM.

Data Sets — We used the following three real-world relational data sets[8], with their statistics shown in Table 5.2.

- *Mondial*: A geography dataset compiled from various geographical Web data sources including CIA World Factbook, Wikipedia and etc.
- *Financial*: A financial database of financial data used in the 1999 European KDD Cup [9].

[8]All available at `https://relational.fit.cvut.cz/`

- *Hepatitis*: A Hepatitis database used in PKDD 2002 Discovery Challenge.

Synthetic User Queries for Experiments — To perform a comprehensive evaluation on above questions, we choose to conduct a simulation study on a wide range of synthesized test cases with various multiresolution constraints in four categories: Exact, Disjunctive, Incomplete, Mix. An actual user study on a PRISM will be our future work.

Exact. In each test case, the input is a sample of data records in the target schema. All cell values are exact and complete; no empty cell is allowed. This is the input accepted by traditional sample-driven schema mapping systems.

Disjunctive. This is similar to an exact test case, except that a disjunction of possible values or value ranges are allowed in one or multiple cells in the provided example.

Incomplete. The examples in the use input are exact and complete, except that there can be cells with "$\star$" in the user input, indicating the values are missing and they can match anything.

Mix. Some of the cells can be a disjunction of possible values or value ranges, and some of the cells can be "$\star$". The rest are exact and complete cell values. (mixture of Disjunctive and Incomplete cells)

To obtain a large collection of various test cases, we propose the following approach to synthesize a test case:

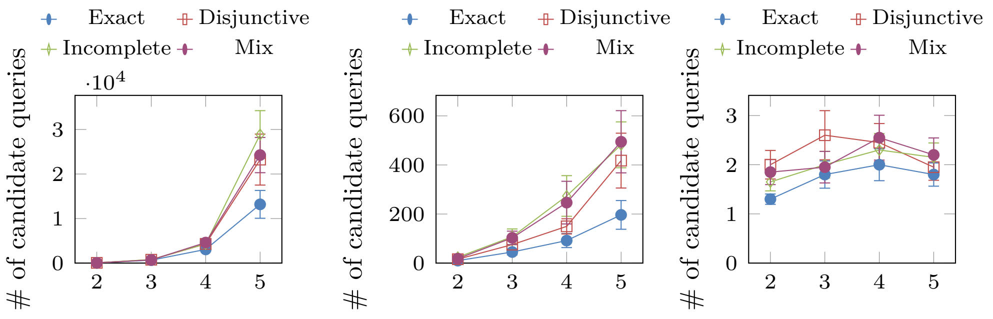
1. Take a Project-Join query generated by a random walk on the schema graph of the source database as the target schema mapping query. The join path lengths (# of joins) of these PJ queries range from 2 to the max join length in each database.

2. Execute the PJ query on the source database and sample some result tuples as *exact* test cases,

3. Randomly select a sampled portion of cells in the above examples and replace them with multiresolution constraints to formulate *disjunctive/incomplete* test cases. To create a test case representing incomplete user input, we simply set the values to "$\star$" for the chosen cells. To create a test case with disjunctive cell values, indicating the user is uncertain about the actual values, we simply couple the actual cell value with several distinct values. For example, if the actual value is "California", we replace it by "California $\vee$ Nevada $\vee \ldots$". When the cell value is numerical, we replace the value by a value range $\pm\triangle$, where $\triangle = 50\%$, and append multiple other distinct value ranges, similar to disjunctive textual constraints.

Many critical metrics we use for evaluating system performance, such as execution time and number of filters, can be largely impacted by different factors in these synthetic test cases. To develop a comprehensive understanding of the problem and system performance, we synthesize the test cases varying the following factors:

- p: number of columns in the target schema (and desired schema mapping query).
- r: number of rows in the user input.
- s: the density (ratio) of cells with low-resolution constraints in the user input.
- w: number of atomic constraint in each cell with a disjunctive constraint (clause) in the user input.



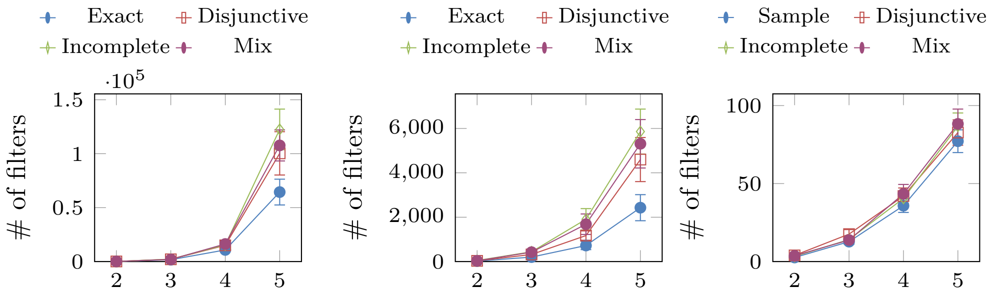
(a) Hepatitis **(b) Mondial** **(c) Financial**

Figure 5.6: Number of candidate queries to be validated at the presence of multiresolution constraints ($p \in [2,5], r = 2, s = 50\%, w = 2$)

5.6.2 Overhead of Multiresolution Constraints

Before assessing the effectiveness of our proposed optimizations for our proposed constraints, we should first understand the motivation behind it. In this section, we demonstrate that the use of multiresolution constraints in describing the target schema may introduce a critical system performance issue and eventually hurt the system usability.

Overview — We examine the baseline systems on exact, disjunctive, incomplete test cases. To understand how severe the performance issue becomes as the schema mapping task becomes more complicated, we vary the number of columns, p, from 2 to 5, and choose a reasonable set of values for the rest of the factors: $r = 2, s = 50\%, w = 2$. For each test setting, we pick 50 random test cases with simulated user input and correspondingly schema mapping query (for verifying purposes). Recall that candidate queries are PJ queries created through finding a path connecting multiple candidate columns on the schema graph, without running it on the database and examining its query result (Section 5.4.1). Multiresolution constraint is a



(a) Hepatitis **(b) Mondial** **(c) Financial**

Figure 5.7: Number of filters to be validated at the presence of multiresolution constraints ($p \in [2, 5], r = 2, s = 50\%, w = 2$)

softer constraint, as opposed to traditional exact sample-driven constraints, which may consequently induce the discovery of more candidate queries. The increased number of candidate queries will certainly result in an escalation of validation effort (possibly quantified by number of filters) in the query validation phase and also system execution time. Therefore, to evaluate the increased system overhead, we measure the execution of each test case from two perspectives: 1) number of candidate queries, 2) number of filters.

Results — The three line charts in Figure 5.6 show the average numbers of candidate queries in three classes of user input for all three different datasets as the number of projected columns increases. Figure 5.7 presents the number of filters derived from these candidate queries.

In Hepatitis, as the number of columns in the target schema p grows from 1 to 5, the number of candidate queries in the Disjunctive test cases and Incomplete test cases increases by 503× and 503×, more rapidly than traditional Exact test cases, which is 300 ×. In terms of number of filters, Incomplete test cases grow by >8000×, which is also much faster than Exact test cases 1000×.

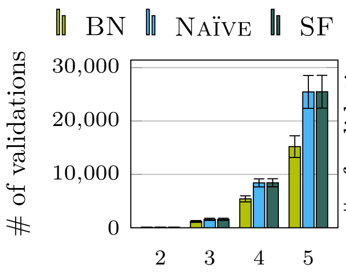
In terms of the Financial test set, although the number of candidate queries does not vary much between different classes of input or numbers of projected columns, an interesting observation is that the number of filters increases much faster than the number of candidate columns when p increases. This is because more filters need to be derived to verify a longer candidate query.

The above results provide evidence that the amount of filters increases faster than the number of candidate queries, which means the later filter validation workload also increases rapidly. Also, low-resolution constraints provided in the user input impose a large overhead on the system, which suggests that further optimizing the filter validation strategy is in great necessity.

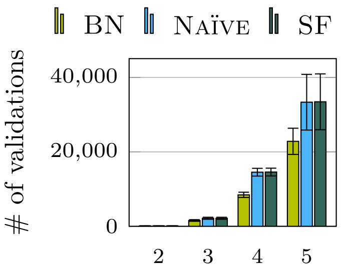
5.6.3 Filter Validation Efficiency

Since the validation result of an individual filter may prune other filters, the ordering of filter validation may dramatically impact the actual number of filters being validated on the source database, and thereby influence the overall filter validation and system efficiency. In Section 5.4.2, we proposed a machine-learning (Bayesian-network) based approach, BN-Greedy, to arrange the sequence of filters to be validated and reduce the actual number of filters validated. Here, we evaluate the effectiveness of our proposed approach in saving the amount of verification required.

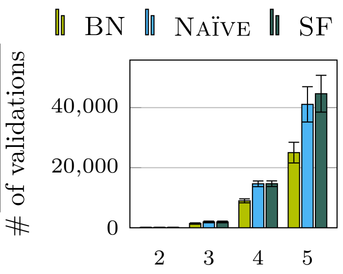
Overview — In this experiment, we experimentally compared BN-Greedy with the baseline strategies Naïve-Greedy and Shortest-First on tasks with different classes of constraints on all data sets, and measured the number of actual filter validations on the database system. The end goal of filter



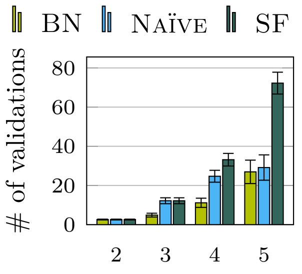
(a) **Hepatitis, Exact**



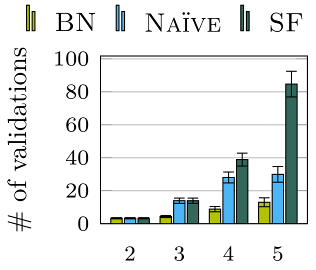
(b) **Hepatitis, Incomplete**



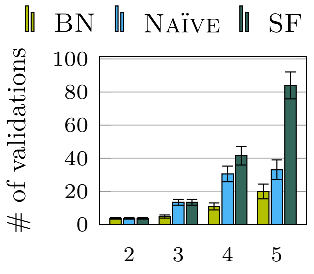
(c) **Hepatitis, Mix**



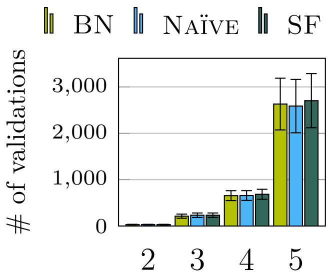
(d) **Financial, Exact**



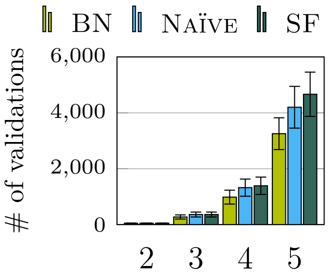
(e) **Financial, Incomplete**



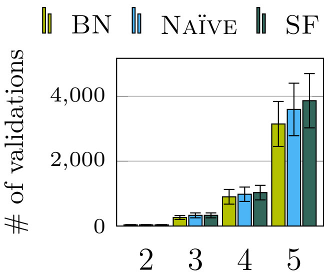
(f) **Financial, Mix**



(g) **Mondial, Exact**



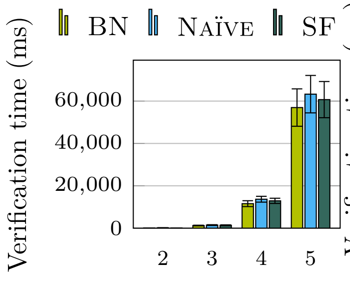
(h) **Mondial, Incomplete**



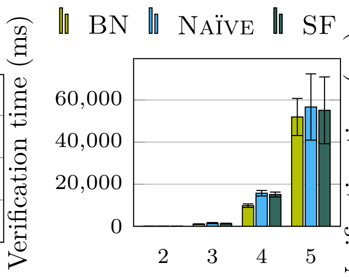
(i) **Mondial, Mix**

Figure 5.8: Compare BN-Greedy (BN), Naïve-Greedy (Naïve), and Shortest-First (SF) on the number of validations ($p \in [2, 5], r = 2, s = 50\%, w = 2$)

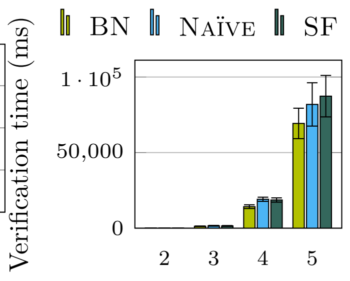
scheduling is to improve the filter validation efficiency. However, the actual time can be greatly impacted by many other factors, such as database



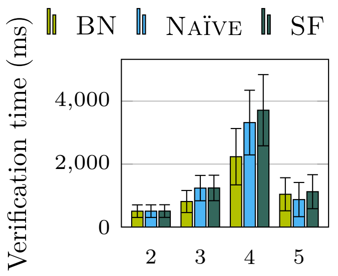
(a) Hepatitis, Exact



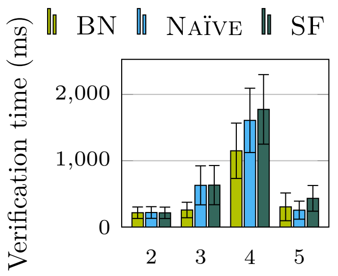
(b) Hepatitis, Incomplete



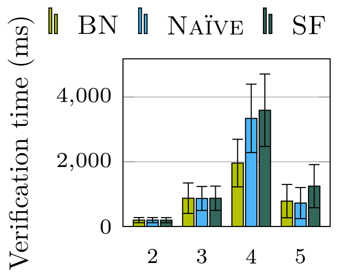
(c) Hepatitis, Mix



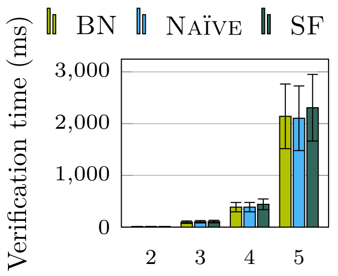
(d) Financial, Disjunction



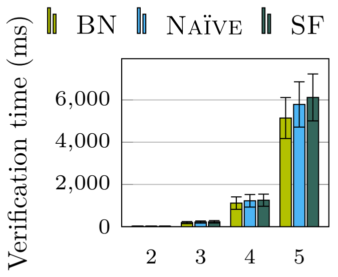
(e) Financial, Incomplete



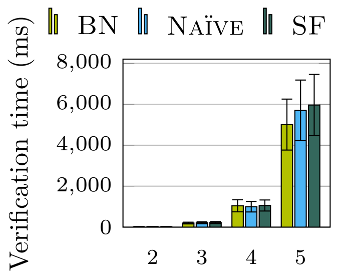
(f) Financial, Mix



(g) Mondial, Exact



(h) Mondial, Incomplete



(i) Mondial, Mix

Figure 5.9: Compare BN-Greedy (BN), Naïve-Greedy (Naïve), and Shortest-First (SF) on validation time ($p \in [2,5], r = 2, s = 50\%, w = 2$)

configurations, cost model used, etc. Still, in this experiment, we measured the actual filter validation timespan for curiosity.

To simulate tasks of different difficulties, we vary the number of columns, p, from 2 to 5, and set the rest of the factors to default values $r = 2, s = 50\%, w = 2$, same as Section 5.6.2.

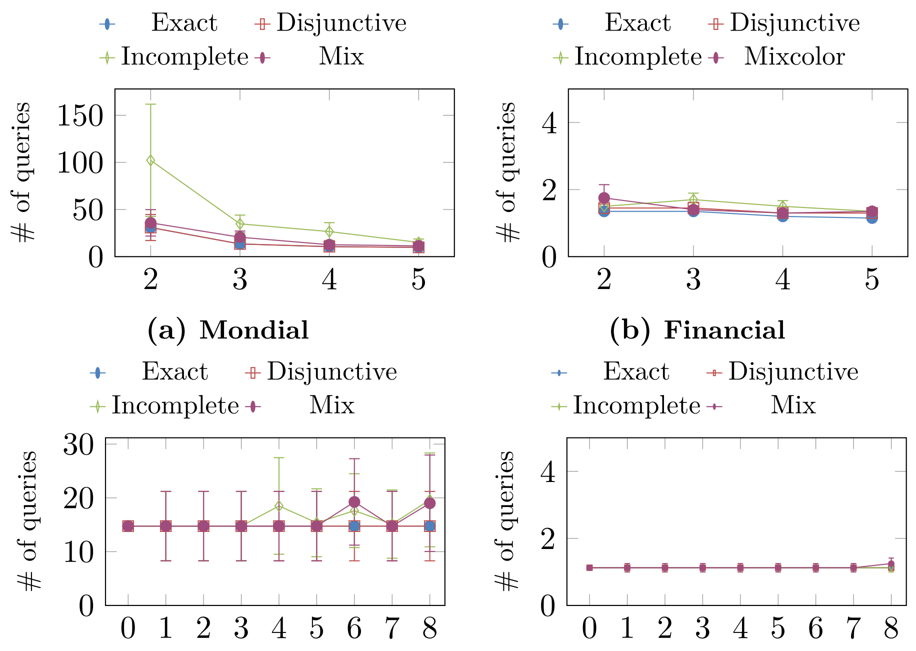
Results — Figure 5.8 shows the actual number of filters validated on the source database given different classes of user input for all three data sets. In experiments with the Hepatitis data set, BN-Greedy achieves a reduction of up to 43.8% and 25.8% on average compared to Naïve-Greedy. When compared to the other strategy, Shortest-First, BN-Greedy achieves a reduction of up to 44.0% and 26.1% on average. In experiments with Financial, BN-Greedy on average saves 39.5% in the actual number of filter validations, compared to the Naïve-Greedy strategy. In the Incomplete test set, the reduction is as much as $\sim 69\%$ when the number of columns in $\mathcal{T}$ is four or five. The reduction relative to Shortest-First is even higher in Financial, which is up to 84.7% and 52.2% on average. As for experiments with the Mondial data set, the reduction is up to 25.7% compared to Naïve-Greedy, and 30.2% compared to Shortest-First, which is not as significant as those in previous two test sets. This is mostly because Mondial is a data set with facts where fewer strong correlations between different columns exist, unlike the previous two datasets. This is usually not where data modeling methods like Bayesian networks shine at.

The actual system runtime cost for the same set experiments is shown in Figure 5.9. In terms of the first data set, Hepatitis, BN-Greedy manages to save 18.0% runtime on average compared to Naïve-Greedy and 11.6% runtime on average compared to Shortest-First. On the Financial data set, BN-Greedy achieves on average reduces $\sim 10\%$ time cost compared to Naïve-Greedy. In particular, the average reduction is $\sim 30\%$ in the expensive cases when $p = 4$. Compared to the other baseline strategy, Shortest-First, the reduction is up to 59.3% and 20.5% on average. As for Mondial, BN-Greedy achieves a decrease in filter validation time of up to 12.4% and 16.5% compared to Naïve-Greedy and Shortest-First.

Overall, the BN-Greedy filter scheduling strategy we proposed yields the least number of actual filter validations in almost all test cases. Considering that there is a lower bound for both metrics (# of filter validations and time) because there is an optimal sequence of filters to validate in theory, this experiment provides strong evidence that BN-Greedy is effective in 1) harnessing the pruning power of filters to a greater extent, and 2) alleviate the performance issue introduced by low-resolution constraints allowed in proposed schema mapping framework compared to the other two scheduling strategies.

5.6.4 User Validation Effort

Multiresolution constraints give the user more freedom in composing queries with imprecise and/or incomplete knowledge. The tradeoff, however, is that as samples with multiresolution constraints may not be precise, the system will lose some "pruning power" while searching in a large space of possible schema mapping queries. A potential risk is that the system may end up returning 10× as many CQs to the user as the satisfying CQs. The task of schema mapping is not complete until the end user read and understand the returned schema mapping queries and finally pick the desired one among them, which we refer to as "user validation". A 10× boost of the end result size may significantly increase the user effort in validating them, and ultimately hurt the usability of the system supporting multiresolution constraints. In this experiment, we evaluated Prism from the angle of its *user validation effort*.



(a) Mondial (b) Financial (c) Mondial (d) Financial

Figure 5.10: Number of satisfying schema mapping queries found when the size of user input increases ($r \in [2, 5], s = 50\%, p = 4, w = 2$) and when the amount of low-resolution constraints in the user input increases (absolute $s \in [0, 8], r = 4, p = 4, w = 2$)

Overview — The amount of user validation effort for a given schema mapping task may be associated with many properties of the returned query set, including query complexity, database complexity, and the size of the query set, etc. In this experiment, we consider the most obvious factor, *query set size*, i.e., the number of the satisfying schema mapping queries output by the system, with the intuition that if the system were to return relatively the same number of satisfying schema mapping queries for test cases with different kinds of user queries, the comprehensibility of the system output should be similar for different classes of user constraints.

What might significantly impact the size of the output is 1) *number of user constraints* and 2) *density of low-resolution constraints*. Intuitively, the more constraints the user provides, the fewer satisfying schema mapping queries should exist. Similarly, the less precise the constraints are, the more schema mapping queries would suffice.

To this end, we performed two experiments varying 1) number of rows of constraints when the ratio of cells with low-resolution constraints remains the same, and 2) number of cells of low-resolution constraints when the number of rows of constraints remains the same. We used three synthetic test sets introduced in Section 5.6.1.

In the first experiment, we varied the number of rows of the user constraints, r, from two to five. Similar to previous experiments, the density of low-resolution constraints, s, was set to 50%. In the second experiment, we varied the number of cells with low-resolution constraints from zero to eight. **Note that when this value is zero, the test case is essentially an "Exact" test case (with only complete sample-driven constraints).** The number of rows of user constraints, r, was set to four, so that we could have enough cells to be replaced by low-resolution constraints. In both experiments, we chose the synthetic queries with the maximum number of joins for the target schemas in each data set.

Results — Figure 5.10 shows the number of satisfying schema mapping queries matching the user input $\mathcal{E}$ as the number of rows of user constraints increases in tests and the number of satisfying queries found as the number of low-resolutions cells increases when the size of the user input $\mathcal{E}$ is fixed ($r = 4$) with both Financial and Mondial data sets.

In experiments with the Financial dataset, the number of output satisfying schema mapping queries are low in all different test sets, and no significant difference can be observed between different test sets. In tests on Mondial, when r goes from 2 to 3, the number of satisfying queries of the Mix test set drops from more than 300 to less than 100 ($> 70\%$). Similarly, the Disjunctive test also experiences a substantial decrease ($> 51\%$) in the number of output queries. While the decrease is slight in the Incomplete test cases, when $r = 3$, when it goes to 4, the actual number of queries still becomes quite close to that of the Exact test case ($\sim 2\times$). Also, when $r = 4$, the output sizes of both Disjunctive and Mix are almost same as that of Exact. The result is still acceptable considering that the user only provides only half as much information as she needs to provide for a traditional sample-driven schema mapping system (50% of cells in Incomplete are empty).

In tests with both Mondial and Financial, we observe that the number only slightly fluctuates which suggests that when the user is able to provide four examples to describe the target schema, having some low-resolution constraints in these examples does not impact the effort to validate the returned satisfying queries.

5.7 Related Work

Schema mapping and query discovery — Composing SQL queries for schema mapping is known to be non-trivial and burdensome for both professionals and naïve end users. Researchers from both academia and industry have made attempts to facilitate this process for end users in the past two decades. These developed techniques can be generally categorized into two classes based on the interaction model: *schema-driven* and *sample-driven*. IBM Clio [83], Microsoft BizTalk are notable schema mapping systems that support the schema-driven model, which requires hints of possible matching relations and columns from end users. Another thread of research projects [12, 50, 72, 84, 93, 105] focused on supporting a sample-driven model in human-in-the-loop schema mappings. Instead of soliciting matching hints, these systems only ask the user for some data records in the target schema.

Based on the assumption about the user input, these works can be categorized into two classes: *open-world* and *closed-world*. Our work along with [84, 93] make an *open-world* assumption about the user input: the user can provide a subset of possible tuples of the desired schema. [84, 93] assume that the user example is precise (high-resolution). Our project targets leveraging coarse (low-resolution) user knowledge in schema mapping. On the other hand, [23, 61, 77, 101, 102, 105] make a *closed-world* assumption about the user input that the user is expected to provide a complete result set of the desired schema, where the search problem is more tractable since the search space is smaller than that in an open-world setting.

Program synthesis — Our end goal is to synthesize schema mapping programs. The problem of program synthesis is the automatic construction of programs using hints from the users like constraints and examples and has recently drawn interest from many researchers. Besides schema mapping, program synthesis has been broadly explored in many problem domains such as data cleaning [7, 29, 46], schema mapping [12, 72, 84, 93, 105], regular expressions [11], bit-manipulation programs [31], and recently neural networks [113], where writing programs is non-trivial for humans.

Different technical approaches have been proposed to resolve the program synthesis problem. The logic-solver-based approach [32, 44] does not apply because it requires modeling the entire source database using second order logic which is infeasible. Another approach—version space algebra—usually fits the situation where finding matching programs is inexpensive and the space of matching programs is large and is thereby not applicable in our problem setting. We formulate our problem as a search problem and, inspired by [84, 93], we developed a machine-learning-guided search-based algorithm to discover a complete set of matching schema mapping queries.

Selectivity estimation — Selectivity estimation has been critical for query optimization in DBMS and we apply this concept to schema mapping to reduce the candidate query validation overhead. Techniques of selectivity estimation often seek to construct succinct representations (e.g., samples [34, 64], histograms [14, 73, 82] and probabilistic models [28, 38, 78, 103]) of underlying data to approximate its joint distribution through scanning the database. Like [28, 78, 103], we leverage probabilistic models rather than sample-based methods because the selectivities of the queries (filters) we consider in our problem setting is often small and may not be covered by the chosen samples. Histogram-based methods are often used to estimate queries with a small number of attributes which make it not realistic in our problem because the target schema is not limited to only two or three columns. Unlike [14, 78], we are blind to the upcoming queries (or filters) and construct the model offline whereas [14, 78] take an query-driven approach and adaptively learn data representations from previous query results.

5.8 Conclusion

In this project, we presented a sample-driven schema mapping paradigm with support for a richer set of constraints to describe the target schema. The user may provide constraints at various "resolutions": not only exact values at row level, but also multiple possible values or value ranges. To tackle a more challenging query search problem posed by the new constraints, we present a Bayesian network-based scheduling strategy to speed up the query verification process, which achieves a verification workload reduction of up to $\sim 69\%$ and a runtime reduction of up to $\sim 30\%$ compared to the best baseline strategy in our experiments. In the future, we would like to extend Prism to consume more user knowledge in various forms to further reduce the required expertise for non-expert users in schema mapping.

Chapter 6

Conclusions and Future Work

Data preparation is a tedious process for data users and the traditional solution of composing ad-hoc programs may require high programming skills that are beyond many data users. In this dissertation, we try to address this issue by presenting three example-guided program synthesis, or Programming By Example (PBE), systems—Foofah, Clx, and Prism—that are able to generate executable data preparation programs for data users without expertise in programming. Foofah and Clx targets transforming spreadsheet or string data into a desired form. Presumably, the Foofah user provides input-output examples—sample input spreadsheet data in its raw form and the target form—and Foofah will derive a data transformation program—a sequence of parameterized pre-defined operations—converting the raw spreadsheet data in its original form into the target form. Clx on the other hand applies PBE in a slightly different task domain—string data transformation. A pattern discovery step is prepended to PBE in Clx so that users can identify data format errors and specify positive examples in forms of patterns, a representation easier to understand and specify than strings used by other PBE string transformation systems. Prism considers the problem of SQL query synthesis using the example data records from the target table. Besides exact data records, imprecise data records—data records with disjunctions, value ranges or null values—can also be provided to the system which makes it more accessible to users less familiar with the database content.

There are multiple directions in which this dissertation can be extended in order to make the technology we propose prevalent in the future.

6.1 Future Work

Increase the expressivity of PBE data transformations — In Chapter 3, we present FOOFAH, a PBE system that is able to generate a matching data transformation program in the form of a sequence of traditional transformation operations defined in [86]. When workflow changes, existing operators may be insufficient to describe the desired transformation and new operations may need to be defined and added to the PBE system. Future work includes exploring a systematic approach to define data transformation operations for spreadsheet data or relational data and incorporate them into the PBE synthesizer core.

PBE data transformation with machine learning based program synthesizers — Both FOOFAH and CLX rely on enumerative search and rule-based heuristics to discover programs matching the user input. Such a design is usually specific to a fixed set of operators, but if new operators are incorporated to adapt to a new workflow, rules-based heuristics may be hard to reuse. Redesigning new heuristics can only be done by experts who are familiar with the domain and hence costly. Thus, we think future work may include exploring opportunities to use machine learning to guide the search and to optimize the combination of heuristic and pruning rules.

Data-aware pattern discovery algorithms for PBE transformations — In Chapter 4, we propose to use patterns as a representation alternative to actual strings in PBE data transformation to lower the difficulty of specifying examples for end users. Clx uses a rule-based pattern clustering strategy to identify patterns within the given dataset. Although the hierarchy itself is helpful in reducing the number of patterns users initially need to comprehend, users may still be shown a large number of patterns since the pattern clustering strategy will eventually discover a complete set of patterns at all levels of abstraction. Future work should include advancing the pattern discovery algorithm that is able to make smart decisions to present patterns at appropriate levels of abstraction for a given dataset.

Explore bounds of example representations for PBE data preparation — Clx and Prism is an attempt to explore varied forms of examples (patterns as examples or imprecise examples) as opposed to precise and highly-specific examples users provide in classic PBE data preparation. We envision that opportunities could be explored to 1) support column-based descriptions/hints (as opposed to row-based examples), such as semantic hints, metadata constraints, tackle tasks like schema mappings, 2) identify other sub-domains within data preparation (e.g., data migration and data exploration) where new forms of examples that are more friendly to non-expert users could be allowed and leveraged in PBE data preparation.

Bibliography

[1] Z. Abedjan, L. Golab, and F. Naumann. “Profiling Relational Data: A Survey”. In: *The VLDB Journal* 24.4 (Aug. 2015), pp. 557–581. ISSN: 1066-8888.

[2] Z. Abedjan, J. Morcos, M. N. Gubanov, I. F. Ilyas, M. Stonebraker, P. Papotti, and M. Ouzzani. “Dataxformer: Leveraging the Web for Semantic Transformations”. In: *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org, 2015.

[3] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. “DataXFormer: A robust transformation discovery system”. In: *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*. IEEE Computer Society, 2016, pp. 1134–1145.

[4] B. Alexe, B. T. Cate, P. G. Kolaitis, and W.-C. Tan. “Characterizing schema mappings via data examples”. In: *ACM Transactions on Database Systems* 36.4 (2011), p. 23.

[5] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama. “Search-based program synthesis”. In: *Communications of the ACM* 61.12 (2018), pp. 84–93.

[6] M. Banko, M. J. Cafarella, S. Soderland, M. Broadhead, and O. Etzioni. “Open Information Extraction from the Web”. In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. IJCAI’07. Hyderabad, India: Morgan Kaufmann Publishers Inc., 2007, pp. 2670–2676.

[7] D. W. Barowy, S. Gulwani, T. Hart, and B. Zorn. "FlashRelate: Extracting Relational Data from Semi-Structured Spreadsheets Using Examples". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 218–228.

[8] J. Bauckmann, U. Leser, F. Naumann, and V. Tietz. "Efficiently Detecting Inclusion Dependencies". In: *2007 IEEE 23rd International Conference on Data Engineering*. 2007, pp. 1448–1450.

[9] P. Berka. *PKDD99 discovery challenge*. http://lisp.vse.cz/pkdd99/chall.htm,1999. 1999.

[10] P. A. Bernstein and L. M. Haas. "Information integration in the enterprise". In: *Communications of the ACM* 51.9 (2008), pp. 72–79.

[11] A. Blackwell. "SWYN: A visual representation for regular expressions". In: *Your Wish Is My Command: Programming by Example* (2001), pp. 245–270.

[12] A. Bonifati, U. Comignani, E. Coquery, and R. Thion. "Interactive Mapping Specification with Exemplar Tuples". In: *Proceedings of the 2017 ACM International Conference on Management of Data*. SIGMOD '17. Chicago, Illinois, USA: Association for Computing Machinery, 2017, pp. 667–682. ISBN: 9781450341974.

[13] R. R. Bouckaert. "Bayesian belief networks: from construction to inference". PhD thesis. 1995.

[14] N. Bruno, S. Chaudhuri, and L. Gravano. "STHoles: A Multidimensional Workload-Aware Histogram". In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD '01. Santa Barbara, California, USA: Association for Computing Machinery, 2001, pp. 211–222. ISBN: 1581133324.

[15] W. Buntine. "A guide to the literature on learning probabilistic networks from data". In: *IEEE Transactions on knowledge and data engineering* 8.2 (1996), pp. 195–210.

[16] M. J. Cafarella, A. Halevy, D. Z. Wang, E. Wu, and Y. Zhang. "Webtables: exploring the power of tables on the web". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 538–549.

[17] Z. Chen and M. Cafarella. "Automatic web spreadsheet data extraction". In: *Proceedings of the 3rd International Workshop on Semantic Search over the Web*. 2013.

[18] Z. Chen, M. Cafarella, J. Chen, D. Prevo, and J. Zhuang. "Senbazuru: a prototype spreadsheet database management system". In: *Proceedings of the VLDB Endowment* 6.12 (2013), pp. 1202–1205.

[19] Z. Chen, M. Cafarella, and H. Jagadish. "Long-tail vocabulary dictionary extraction from the web". In: *WSDM*. ACM. 2016, pp. 625–634.

[20] A. Cheung, A. Solar-Lezama, and S. Madden. "Optimizing database-backed applications with query synthesis". In: *PLDI*. 2013.

[21] G. Cooper and E. Herskovits. "A Bayesian method for the induction of probabilistic networks from data". In: *Machine Learning* 9.4 (1992), pp. 309–347.

[22] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. "Robustfill: Neural program learning under noisy i/o". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 990–998.

[23] A. Fariha and A. Meliou. "Example-Driven Query Intent Discovery: Abductive Reasoning using Semantic Similarity". In: 2019.

[24] K. Fisher and R. Gruber. "PADS: A Domain-specific Language for Processing Ad Hoc Data". In: *PLDI*. Vol. 40. 6. ACM. 2005, pp. 295–304.

[25] K. Fisher, D. Walker, K. Q. Zhu, and P. White. "From dirt to shovels: fully automatic tool generation from ad hoc data". In: *ACM SIGPLAN Notices*. Vol. 43. 1. ACM. 2008, pp. 421–434.

[26] Y. Gao, S. Huang, and A. Parameswaran. “Navigating the data lake with datamaran: Automatically extracting structure from log datasets”. In: *Proceedings of the 2018 International Conference on Management of Data*. 2018, pp. 943–958.

[27] M. R. Gary and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. 1979.

[28] L. Getoor, B. Taskar, and D. Koller. “Selectivity estimation using probabilistic models”. In: *ACM SIGMOD*. 2001.

[29] S. Gulwani. “Automating string processing in spreadsheets using input-output examples”. In: *POPL*. 2011.

[30] S. Gulwani, W. R. Harris, and R. Singh. “Spreadsheet data manipulation using examples”. In: *Communications of the ACM* 55.8 (2012), pp. 97–105.

[31] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. *Component based synthesis applied to bitvector circuits*. Tech. rep. Technical Report MSR-TR-2010-12, Microsoft Research, 2010.

[32] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. “Synthesis of loop-free programs”. In: 2011.

[33] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer. “Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts”. In: *UIST*. 2011.

[34] P. J. Haas, J. F. Naughton, and A. N. Swami. “On the relative cost of sampling for join selectivity estimation”. In: *Proceedings of the thirteenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM. 1994, pp. 14–24.

[35] W. R. Harris and S. Gulwani. “Spreadsheet table transformations from examples”. In: *PLDI*. 2011.

[36] P. E. Hart, N. J. Nilsson, and B. Raphael. “A formal basis for the heuristic determination of minimum cost paths”. In: *Systems Science and Cybernetics, IEEE Transactions on* 4.2 (1968), pp. 100–107.

[37] Y. He, X. Chu, K. Ganjam, Y. Zheng, V. Narasayya, and S. Chaudhuri. “Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations”. In: *PVLDB*. 2018.

[38] M. Heimel, M. Kiefer, and V. Markl. “Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation”. In: *SIGMOD Conference*. 2015.

[39] J. M. Hellerstein, J. Heer, and S. Kandel. “Self-Service Data Preparation: Research to Practice.” In: *IEEE Data Eng. Bull.* 41.2 (2018), pp. 23–34.

[40] G. Holmes, A. Donkin, and I. H. Witten. “Weka: A machine learning workbench”. In: *Proceedings of ANZIIS'94-Australian New Zealnd Intelligent Information Systems Conference*. IEEE. 1994, pp. 357–361.

[41] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. “TANE: An efficient algorithm for discovering functional and approximate dependencies”. In: *The computer journal* 42.2 (1999), pp. 100–111.

[42] D. Huynh and S. Mazzocchi. *OpenRefine*. `http://openrefine.org`. 2012.

[43] F. Islam, V. Narayanan, and M. Likhachev. “Dynamic Multi-Heuristic A*”. In: *IEEE International Conference on Robotics and Automation*. 2015.

[44] S. Jha, S. Gulwani, S. Seshia, A. Tiwari, et al. “Oracle-guided component-based program synthesis”. In: *ICSE*. 2010.

[45] 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. Chicago, Illinois, USA: ACM, 2017, pp. 1607–1610.

[46] Z. Jin, M. R. Anderson, M. Cafarella, and H. V. Jagadish. “Foofah: Transforming Data By Example”. In: *SIGMOD*. 2017.

[47] Z. Jin, C. Baik, M. Cafarella, and H. V. Jagadish. “Beaver: Towards a Declarative Schema Mapping”. In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. HILDA’18. Houston, TX, USA: ACM, 2018, 10:1–10:4.

[48] Z. Jin, C. Baik, M. Cafarella, H. Jagadish, and Y. Lou. “Demonstration of a Multiresolution Schema Mapping System”. In: *CIDR*. 2019.

[49] Z. Jin, M. Cafarella, H. V. Jagadish, S. Kandel, M. Minar, and J. M. Hellerstein. “CLX: Towards verifiable PBE data transformation”. In: *EDBT*. 2019.

[50] D. V. Kalashnikov, L. V. Lakshmanan, and D. Srivastava. “FastQRE: Fast Query Reverse Engineering”. In: *SIGMOD*. 2018.

[51] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. “Enterprise data analysis and visualization: An interview study”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (2012), pp. 2917–2926.

[52] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. “Wrangler: Interactive visual specification of data transformation scripts”. In: *CHI*. ACM. 2011, pp. 3363–3372.

[53] G. D. Kleiter. “Propagating imprecise probabilities in Bayesian networks”. In: *Artificial Intelligence* 88.1-2 (1996), pp. 143–161.

[54] K. Krishnamoorthy. *Handbook of statistical distributions with applications*. Chapman and Hall/CRC, 2016.

[55] T. A. Lau, P. M. Domingos, and D. S. Weld. “Version Space Algebra and its Application to Programming by Demonstration.” In: *ICML*. 2000.

[56] T. Lau, S. A. Wolfman, P. Domingos, and D. S. Weld. “Programming by demonstration using version space algebra”. In: *Machine Learning* 53.1-2 (2003), pp. 111–156.

[57] V. Le and S. Gulwani. “FlashExtract: A framework for data extraction by examples”. In: *PLDI*. Vol. 49. 6. ACM. 2014, pp. 542–553.

[58] M. Lenzerini. “Data integration: A theoretical perspective”. In: *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. 2002, pp. 233–246.

[59] A. Leung, J. Sarracino, and S. Lerner. “Interactive parser synthesis by example”. In: *PLDI*. 2015.

[60] F. Li and H. Jagadish. “Constructing an interactive natural language interface for relational databases”. In: *Proceedings of the VLDB Endowment* 8.1 (2014), pp. 73–84.

[61] H. Li, C.-Y. Chan, and D. Maier. “Query from examples: An iterative, data-driven approach to query construction”. In: *Proceedings of the VLDB Endowment* 8.13 (2015), pp. 2158–2169.

[62] Y. Li, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Jagadish. “Regular expression learning for information extraction”. In: *EMNLP*. 2008.

[63] H. Lieberman. *Your wish is my command: Programming by example*. Morgan Kaufmann, 2001.

[64] R. J. Lipton, J. F. Naughton, and D. A. Schneider. *Practical selectivity estimation through adaptive sampling*. Vol. 19. 2. ACM, 1990.

[65] S. Lohr. “For big-data scientists, janitor work is key hurdle to insights”. In: *The New York Times* 17 (2014).

[66] S. Lopes, J.-M. Petit, and F. Toumani. “Discovering interesting inclusion dependencies: application to logical database tuning”. In: *Information Systems* 27.1 (2002), pp. 1–19.

[67] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. “Jungloid mining: helping to navigate the API jungle”. In: *PLDI*. 2005.

[68] H. Massalin. “Superoptimizer: a look at the smallest program”. In: *ASPLOS II*. 1987.

[69] W. May. *Information Extraction and Integration with* FLORID*: The* MONDIAL *Case Study*. Tech. rep. 131. Available from http://dbis.informatik.uni-goettingen.de/Mondial. Universität Freiburg, Institut für Informatik, 1999.

[70] R. J. Miller, L. M. Haas, and M. A. Hernández. “Schema mapping as query discovery”. In: *VLDB*. Vol. 2000. 2000, pp. 77–88.

[71] T. M. Mitchell. “Generalization as search”. In: *Artificial intelligence* 18.2 (1982), pp. 203–226.

[72] D. Mottin, M. Lissandrini, Y. Velegrakis, and T. Palpanas. “Exemplar queries: Give me an example of what you need”. In: *PVLDB*. 2014.

[73] M. Muralikrishna and D. J. DeWitt. “Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries”. In: *SIGMOD*. 1988.

[74] M. Neuhaus and H. Bunke. *Bridging the gap between graph edit distance and kernel machines*. World Scientific Publishing Co., Inc., 2007.

[75] N. OpenData. *Times Square Food & Beverage Locations” data set*. `https://opendata.cityofnewyork.us/`. 2017.

[76] S. Padhi, P. Jain, D. Perelman, O. Polozov, S. Gulwani, and T. Millstein. “FlashProfile: a framework for synthesizing data profiles”. In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), pp. 1–28.

[77] K. Panev and S. Michel. “Reverse Engineering Top-k Database Queries with PALEO.” In: *EDBT*. 2016, pp. 113–124.

[78] Y. Park, S. Zhong, and B. Mozafari. “QuickSel: Quick Selectivity Learning with Mixture Models”. In: *SIGMOD*. 2019.

[79] D. Patil. *Data Jujitsu*. ” O’Reilly Media, Inc.”, 2012.

[80] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. “Scaling up superoptimization”. In: *ASPLOS*. 2016.

[81] G. Piatetsky. “Four main languages for analytics, data mining, data science”. In: *URL http://www. kdnuggets. com/2014/08/four-main-languages-analyticsdata-mining-data-science. html* (2014).

[82] V. Poosala and Y. E. Ioannidis. “Selectivity Estimation Without the Attribute Value Independence Assumption”. In: *VLDB*. 1997.

[83] L. Popa, Y. Velegrakis, M. A. Hernández, R. J. Miller, and R. Fagin. "Translating web data". In: *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment. 2002, pp. 598–609.

[84] L. Qian, M. J. Cafarella, and H. Jagadish. "Sample-driven schema mapping". In: *SIGMOD*. 2012.

[85] V. Raman and J. Hellerstein. *An Interactive Framework for Data Cleaning*. Tech. rep. 2000.

[86] V. Raman and J. M. Hellerstein. "Potter's Wheel: An interactive data cleaning system". In: *VLDB*. Vol. 1. 2001, pp. 381–390.

[87] T. Rattenbury, J. M. Hellerstein, J. Heer, S. Kandel, and C. Carreras. *Principles of Data Wrangling: Practical Techniques for Data Preparation*. " O'Reilly Media, Inc.", 2017.

[88] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev. "Refactoring with synthesis". In: *OOPSLA*. 2013.

[89] M. Research. *Microsoft Program Synthesis using Examples SDK*. `https://microsoft.github.io/prose/`. 2017.

[90] J. Rissanen. "Modeling by shortest data description". In: *Automatica* 14.5 (1978), pp. 465–471.

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

[92] D. A. Schum. *The evidential foundations of probabilistic reasoning*. Northwestern University Press, 2001.

[93] Y. Shen, K. Chakrabarti, S. Chaudhuri, B. Ding, and L. Novik. "Discovering queries based on example tuples". In: *SIGMOD*. 2014.

[94] R. Singh. "BlinkFill: Semi-supervised programming by example for syntactic string transformations". In: *Proceedings of the VLDB Endowment* 9.10 (2016), pp. 816–827.

[95] R. Singh and S. Gulwani. “Learning semantic string transformations from examples”. In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 740–751.

[96] R. Singh and S. Gulwani. “Predicting a correct program in programming by example”. In: *CAV*. 2015.

[97] R. Singh and A. Solar-Lezama. “Synthesizing data structure manipulations from storyboards”. In: *ESEC/FSE*. 2011.

[98] A. Solar-Lezama. *Program synthesis by sketching*. ProQuest, 2008.

[99] M. Stonebraker, D. Bruckner, I. F. Ilyas, G. Beskales, M. Cherniack, S. B. Zdonik, A. Pagan, and S. Xu. “Data Curation at Scale: The Data Tamer System”. In: *CIDR*. 2013.

[100] S.-G. Synthesis. *SyGuS-COMP 2017: The 4th Syntax Guided Synthesis Competition took place as a satellite event of CAV and SYNT 2017*. http://www.sygus.org/SyGuS-COMP2017.html. 2017.

[101] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. “Query by output”. In: *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM. 2009, pp. 535–548.

[102] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy. “Query reverse engineering”. In: *The VLDB Journal* 23.5 (2014), pp. 721–746.

[103] K. Tzoumas, A. Deshpande, and C. S. Jensen. “Efficiently adapting graphical models for selectivity estimation”. In: *VLDB* 22.1 (2013), pp. 3–27.

[104] C. F. Vasters. *BizTalk Server 2000: A Beginner’s Guide*. McGraw-Hill Professional, 2001.

[105] 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*. PLDI 2017. Barcelona, Spain: Association for Computing Machinery, 2017, pp. 452–466.

[106] I. H. Witten and D. Mo. “TELS: Learning text editing tasks from examples”. In: *Watch what I do*. MIT Press. 1993, pp. 183–203.

[107] B. Wu and C. A. Knoblock. “An Iterative Approach to Synthesize Data Transformation Programs”. In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI’15. Buenos Aires, Argentina: AAAI Press, 2015, pp. 1726–1732.

[108] B. Wu, P. Szekely, and C. A. Knoblock. “Learning data transformation rules through examples: Preliminary results”. In: *IIWeb*. 2012.

[109] B. Wu, P. Szekely, and C. A. Knoblock. “Minimizing user effort in transforming data by example”. In: *IUI*. ACM. 2014, pp. 317–322.

[110] H. Yao and H. J. Hamilton. “Mining functional dependencies from data”. In: *Data Mining and Knowledge Discovery* 16.2 (2008), pp. 197–219.

[111] E. Zhu, Y. He, and S. Chaudhuri. “Auto-join: joining tables by leveraging transformations”. In: vol. 10. 10. VLDB Endowment, 2017, pp. 1034–1045.

[112] K. Zhu, K. Fisher, and D. Walker. “Learnpads++: Incremental inference of ad hoc data formats”. In: *Practical Aspects of Declarative Languages* (2012), pp. 168–182.

[113] B. Zoph and Q. V. Le. “Neural Architecture Search with Reinforcement Learning”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.