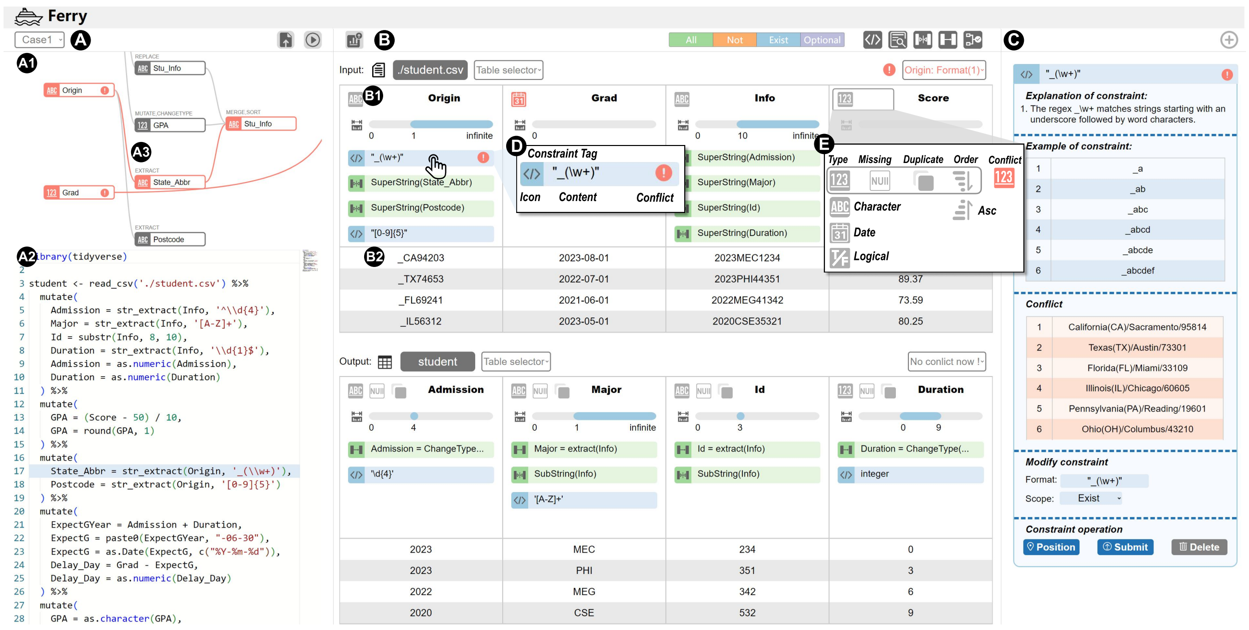
Ferry: Toward Better Understanding of Input/Output Space for Data Wrangling Scripts

Zhongsu Luo, Kai Xiong, Jiajun Zhu, Ran Chen, Xinhuan Shu, Di Weng, and Yingcai Wu



```
library(tidyverse)

student <- read_csv('./student.csv') %>%
  mutate(
    Admission = str_extract(Info, '^\\d{4}'),
    Major = str_extract(Info, '[A-Z]+'),
    Id = substr(Info, 8, 10),
    Duration = str_extract(Info, '\\d{1}$'),
    Admission = as.numeric(Admission),
    Duration = as.numeric(Duration)
  ) %>%
  mutate(
    GPA = (Score - 50) / 10,
    GPA = round(GPA, 1)
  ) %>%
  mutate(
    State_Abbr = str_extract(Origin, '_(\\w+)'),
    Postcode = str_extract(Origin, '[0-9]{5}')
  ) %>%
  mutate(
    ExpectGYear = Admission + Duration,
    ExpectG = paste0(ExpectGYear, "-06-30"),
    ExpectG = as.Date(ExpectG, c("%Y-%m-%d")),
    Delay_Day = Grad - ExpectG,
    Delay_Day = as.numeric(Delay_Day)
  ) %>%
  mutate(
    GPA = as.character(GPA),
```

Origin	Grad	Info	Score
_CA94203	2023-08-01	2023MEC1234	
_TX74653	2022-07-01	2023PHI44351	89.37
_FL69241	2021-06-01	2022MEG41342	73.59
_IL56312	2023-05-01	2020CSE35321	80.25

Admission	Major	Id	Duration
2023	MEC	234	0
2023	PHI	351	3
2022	MEG	342	6
2020	CSE	532	9

"_(\w+)"

Explanation of constraint:
1. The regex _\w+ matches strings starting with an underscore followed by word characters.

Example of constraint:

1	_a
2	_ab
3	_abc
4	_abcd
5	_abcde
6	_abcdef

Conflict

1	California(CA)/Sacramento/95814
2	Texas(TX)/Austin/73301
3	Florida(FL)/Miami/33109
4	Illinois(IL)/Chicago/60605
5	Pennsylvania(PA)/Reading/19601
6	Ohio(OH)/Columbus/43210

Modify constraint

Format: "_(\w+)"
Scope: Exist

Constraint operation

Position Submit Delete

Fig. 1: The interface of Ferry. (A) Script View displays the associations of data columns involved in the transformation (A1) and the uploaded data wrangling scripts (A2); (B) Constraint View displays the input/output space with sample data to clarify the space; (C) Detail View presents constraint details to aid in understanding the table space and allows users to manipulate the constraint.

Abstract—Understanding the input and output of data wrangling scripts is crucial for various tasks like debugging code and onboarding new data. However, existing research on script understanding primarily focuses on revealing the process of data transformations, lacking the ability to analyze the potential scope, i.e., the space of script inputs and outputs. Meanwhile, constructing input/output space during script analysis is challenging, as the wrangling scripts could be semantically complex and diverse, and the association between different data objects is intricate. To facilitate data workers in understanding the input and output space of wrangling scripts, we summarize ten types of constraints to express table space and build a mapping between data transformations and these constraints to guide the construction of the input/output for individual transformations. Then, we propose a constraint generation model for integrating table constraints across multiple transformations. Based on the model, we develop Ferry, an interactive system that extracts and visualizes the data constraints describing the input and output space of data wrangling scripts, thereby enabling users to grasp the high-level semantics of complex scripts and locate the origins of faulty data transformations. Besides, Ferry provides example input and output data to assist users in interpreting the extracted constraints and checking and resolving the conflicts between these constraints and any uploaded dataset. Ferry's effectiveness and usability are evaluated through two usage scenarios and two case studies, including understanding, debugging, and checking both single and multiple scripts, with and without executable data. Furthermore, an illustrative application is presented to demonstrate Ferry's flexibility.

Index Terms—Data wrangling, Visual analytics, Constraints, Program understanding

1 Introduction

• Z. Luo, K. Xiong, J. Zhu, R. Chen and Y. Wu are with the State Key Lab of CAD&CG, Zhejiang University, Hangzhou, China. E-mail: {zhongsuluo, kaixiong, jiajunzhuchris, chenran928, ycwu}@zju.edu.cn.
• X. Shu is with School of Computing, Newcastle University, Newcastle Upon Tyne, United Kingdom. E-mail: xinhuan.shu@newcastle.ac.uk
• D. Weng is with the School of Software Technology, Zhejiang University, Ningbo, China. E-mail: dweng@zju.edu.cn. D. Weng is the corresponding author.

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

Data wrangling is a crucial process that makes data consumable for downstream applications [29]. It involves an assortment of data transformations [53], such as filtering and sorting, which can be performed by writing wrangling scripts using programming languages and their transformation libraries (e.g., R and tidyverse [66]).

Numerous scenarios require data workers to understand and make appropriate changes to existing data wrangling scripts, such as debugging scripts issues and reusing scripts for new datasets [67]. Prior studies [51, 67, 68] primarily focus on revealing the transformation processes conducted by these scripts from the data perspective, i.e., *how* tables change over the course of transformations. However, the importance of understanding the input and output space of these scripts, i.e., *what* input data can be accepted by the scripts and *what* output data will be produced accordingly, is largely overlooked by current approaches. Such understanding significantly enhances efficiency and effectiveness in a wide range of scripts editing scenarios. For instance, to reuse a script written years ago or by others for a new data format, data workers need to ascertain whether the input data space of the script can accommodate the new format and whether the output data space generated on the new data will align with the transformation goals.

In addition, the current approaches for scripts understanding often rely solely on the limited datasets supplied by users. The lack of a holistic understanding of the input and output data scopes may fail to uncover latent errors within the data transformation processes. For example, issues such as unexpected missing values in the transformation output may only manifest when a column's values in the input data exceed a specific threshold – a scenario that may not be covered in the sample data provided by users. This can lead to an incomplete understanding of how the scripts handle edge cases or atypical data, potentially resulting in unforeseen complications during broader applications. Furthermore, the existing approaches may prove inadequate when users aim to analyze scripts in the absence of data, such as scripts obtained from online sources, or when attempting to repurpose scripts for new datasets where direct execution encounters runtime errors.

Inspired by the widespread use of constraints in prior work [2, 8, 38, 39] to restrict tables by imposing requirements, we leverage constraints to express the inputs and outputs of data wrangling scripts, and propose an interactive visualization system that automatically constructs input/output space with constraints to alleviate the limitations of the aforementioned scripts understanding approaches. However, developing such a system poses two primary challenges:

Extraction of Table Constraints. Constructing input/output table space for scripts without data requires static analysis of transformation code on tables. However, the diversity and complexity of the transformation types and their implementation approaches [68] make their constraints on tables non-trivial to determine and extract. Moreover, as data wrangling scripts often contain multiple steps of transformations, it is challenging to integrate constraints inferred from these transformations into a single table, involving inheritance and iterative updating of constraints throughout the data wrangling process.

Presentation of Table Space. As constraints are abstract and lack the concreteness of real data, existing data profiling tools are unable to reveal their characteristics, making them less intuitive to understand. Additionally, multiple constraints construct table space, and expressing the overall effects of these constraints on tables is challenging.

In this work, we collaborated closely with four experts to iteratively formulate design requirements for supporting a holistic understanding of the input and output space for scripts. Based on these requirements, we develop an interactive visual analysis system, called Ferry, to address the two challenges. For the first challenge, we summarize ten types of constraints to express the input/output space and build a mapping that illustrates the effects of diverse data transformations on these constraints. Both the ten constraint types and the mapping are summarized based on data wrangling knowledge from previous work [21, 31, 64, 68] and validated on a real-world data wrangling code dataset [68]. Additionally, we propose a rule-based constraint generation model that constructs the input/output space through iterative forward/backward inference and propagation of constraints. For the second challenge, we design different visualizations for explaining individual constraints and utilize sample data to assist in understanding the overall effects of constraints on the table. With Ferry's assistance, users can understand the script input and output by inspecting table constraints and sample data. Ferry enables users to upload their own data to determine whether the script can handle it and customize output table constraints based on the requirements of downstream tasks. In addition, Ferry supports identifying the causes and locations of conflicts between constraints and data to assist users in modifying the script.

To evaluate Ferry's effectiveness and usability, we introduced two usage scenarios that demonstrate improvements in user efficiency: onboarding new data into existing scripts and fixing issues; and debugging scripts without executable data. Next, we conducted two case studies with four experienced data practitioners in different tasks and collected their feedback and suggestions on Ferry. Finally, we integrated Ferry into an existing scripts understanding system to showcase its flexibility.

In summary, the contributions of this paper are as follows:

- We summarize ten types of table constraints for exhibiting the table space and establish a mapping between transformations and these constraints to represent data wrangling knowledge.
- We propose a constraint generation model capable of inferring the input/output space for data wrangling scripts.
- We develop Ferry, an interactive system designed to enhance the efficiency of data workers in tasks requiring a comprehensive understanding of the input and output of data wrangling scripts.

2 Related Work

2.1 Data Wrangling

Data wrangling is a tedious and error-prone task that can consume up to 80% of a data worker's time and effort [6, 30]. Data workers often use programming languages and their libraries to complete data wrangling tasks, such as pandas [49] in Python and tidyverse [66] in R.

However, the use of scripting languages for data wrangling tasks presents two significant challenges for data workers. For those with limited programming skills, script writing can be inefficient. In response, some works focus on enhancing data wrangling efficiency. Moreover, understanding data wrangling scripts, essential for script reuse and debugging, remains a significant challenge. Consequently, other works focus on assisting users in understanding scripts.

Enhancing efficiency. On the one hand, some approaches seek to lower the barrier to data wrangling through interactive methods. Commercial systems, such as Microsoft Excel [14], OpenRefine [47], Trifacta [64], and Tableau Prep Builder [60], enable users to select data operators on their menus. Wrangler [21, 30] recommends and executes appropriate data transformations based on user interactions. Some tools help users synthesize transformation code using programming by example [12, 28]. Rigel [5] introduces a declarative mapping approach that accomplishes data transformation by allowing users to drag data attributes. Some works support data transformation for various types of data, such as graph [4, 23, 25, 36, 40, 63], website [1, 27, 34, 35, 42] and semi-structured [26, 37, 69] data. On the other hand, some approaches enable data wrangling tasks for users through automation methods or natural language. Sutton et al. [59] implemented a tool capable of automating data transformation by comparing differences between data tables. Auto-Pipeline [71] introduces a "by-target" paradigm for pipeline synthesis to achieve automatic data transformation. Data Formulator [65] employs an AI agent to automate the transformation of input data into the format required for visualization. NL2Rigel [26] achieves declarative data transformation through natural language. However, these works are more concerned with how to accomplish data wrangling tasks and do not aid users in understanding scripts or its input/output.

Understanding scripts. Some existing tools employ interactive methods to aid users in debugging and understanding scripts. Unravel [56] is a debugging tool that supports the pipe script structure of the R language, enabling users to explore and understand the code. Some tools exist as plugins to integrate with users' workflows. For instance, WrangleDoc [70] is a JupyterLab plugin that employs program synthesis techniques to generate summaries of output tables, assisting users in identifying errors within scripts. TweakIt [32] is a Microsoft Excel plugin that helps users understand and explore the effect of code on data. Another category of tools concentrates on depicting the semantics of data wrangling scripts through visualization techniques. SOMNUS [67] has designed a set of glyphs that reveal data transformations and employs node-link diagrams to illustrate the changes occurring in data tables within scripts, thereby revealing the semantics of the scripts. Datamations [51] employs animations to demonstrate the changes in data. COMANTICS [68] has designed a pipeline that detects changes in data tables and infers the data transformation type in scripts using a CNN model. These works focus on demonstrating data transformation effects without revealing the script's input/output space and rely on executable data. Consequently, we designed Ferry, a tool that can display a script's input/output space by analyzing the semantic information of data wrangling within the script.

2.2 Constraints-based Methods

Constraints are widely utilized in various domains, such as software engineering [15,33,50], visualization [43,52,58,72] and databases [7,39], to restrict the possible values for a variable, which can represent partial information about the variable [3]. In a constraint program, unknowns such as variables are expressed through a series of constraints, and the solutions must satisfy all the restrictions of these constraints [43]. Constraint programming standardizes knowledge representation, enabling users to model expertise as high-level expressions. Many prior works have employed constraint programming across various domains. Data wrangling knowledge can be represented through the depiction of changes in tabular data. Currently, there are tools available for repairing data structures [9,13,38], testing spreadsheets [2,24], testing databases [7,39], and some commonly used constraint solvers [8,16,17,41], all of which can represent tabular data through constraints. However, they are designed to allow users to specify constraints to represent knowledge rather than to automatically extract constraints. Some tools extract technical constraints from code through #IFDEF variability [33,45,54,62]. ExtractFix [15] extracts information from scripts and infers the constraints necessary for their correct execution, using constraints to fix vulnerabilities in the program. However, they are incapable of extracting data wrangling knowledge from scripts. To fill this gap, we summarized ten types of table constraints and established a mapping between the constraints and data transformation types to represent data wrangling knowledge.

Existing work on explaining constraints typically focuses on constraint solving process [73]. Common methods include search-tree based visualization [11,20,57], visualization of variables [10], and visualization of constraints propagation [18,19]. Some works focus on the visualization of the constraints themselves, employing methods such as ring and tree visualizations [22], as well as network diagrams [48,61], to display pertinent details of the constraints. These works are not suited for presenting constraints that describe table characteristics and the table space. To address this deficiency, we capture constraints through constraint tags and represent table space using sample data, thereby aiding users in understanding constraints and table space.

3 Table Constraints

A script's input/output space describes the possible sets of input/output tables that the script can handle/generate. It declares which specific columns the table contains and what data characteristics (e.g., value range, data format) these columns should/could possess. Prior research has widely used constraints to effectively capture the scope of these tables, which impose requirements defining necessary data characteristics in terms of schema, format, values, and other aspects. Therefore, we utilize constraints to express the input/output space. In this section, we summarize ten types of table constraints applicable to the data wrangling domain and establish a mapping to guide the construction of the input/output space for data transformations using these constraints.

3.1 Methodology

Numerous studies on data constraints [2,8,38,39] exist, but they are not tailored to the requirements of tables in the data wrangling domain. To clarify what types of constraints data transformations can impose on tables and what impact each transformation will have on these constraints, we extensively surveyed existing work in data wrangling [21,31,68], databases [7,39], and data generation [8,16,17,41]. We performed a comprehensive analysis and summary through the following four steps.

First, we analyzed and summarized 22 commonly used data transformations and their impacts on table changes, based on the multi-table framework for data wrangling [31] and the semantic descriptions in Wrangler [21], Trifacta [64], and COMANTICS [68]. These transformations were considered domain knowledge in data wrangling. Second, we reviewed data constraints in existing constraint solvers (e.g., Z3 [8,41] and Clingo [16,17]) and relational databases (e.g., NOT NULL, UNIQUE). We identified the specific characteristics each constraint imposes on tables, yielding a candidate pool of constraints. Third, we established a mapping (see Sec. 1 of the supplementary material for details) showing the effects of the 22 transformations on table constraints from the candidate pool. This mapping guides the construction of input/output constraints for individual transformations, further described in Sec. 3.3. During construction, we found some constraints, like statistical constraints requiring a specific distribution (e.g., normal distribution), indicating that they do not contribute to expressing the semantics of these transformations. Thus, we removed these constraints from the pool. We also introduced new constraints to better capture the semantics of transformations, such as the derived relation describing the specific transformation that produced the current column. Fourth, we organized and categorized the updated candidate pool, resulting in ten types of constraints (see Sec. 3.2) that express the table space in data wrangling.

To validate the effectiveness of the ten table constraints and their mapping to data transformations, we conducted an experiment using a real-world dataset from COMANTICS [68], which included 921 lines of data transformation code with its data. In this experiment, we executed each line of code in the dataset and obtained its input and output tables. Thereafter, we checked whether these tables complied with the constraints specified in our mapping for the corresponding transformations and assessed if additional table constraints were required to further clarify the semantics. The results of the experiment showed that all input and output tables adhered to the constraints specified in our mapping, and no additional table constraints were needed.

3.2 Constraint Types

This subsection describes the ten types of table constraints highlighted in bold as follows. Note that these constraints are applied to the table columns to express the input/output table space.

- **Data types** for columns are classified into four common categories: Character, Numeric, Date, and Logic. Some of the following nine table constraints vary based on the data type.
- **Range** represents the value ranges for Numeric/Date data and the string length boundaries for Character data. Using string length as the range for Character not only aligns with the continuous nature of Numeric/Date ranges for unified visualization, but also helps uncover errors in operations like 'substr' that rely on string length, thereby facilitating debugging (as described in Sec. 5.2).
- **Format** specifies the text format of data values. We use format patterns (e.g., '0.00' and 'yyyy-mm-dd') for Numeric/Date data and regular expressions (e.g., '\w+') for Character data.
- **Order** represents the sorting of data values (e.g., ascending, descending, or disordered)
- **Specific Value** denotes particular values within the data.
- **Missing Value** refers to absent data values (e.g., NA, NULL).
- **Duplicate Value** represents repeated values within the data.
- **Comparative relation** focuses on the comparison between columns. It defines numerical/temporal comparisons (e.g., less than $<$ and greater than $>$) for Numeric/Date data and set relations (e.g., substring $\subset$) for Character data.
- **Combination relation** describes conditions where multiple columns need to satisfy the same constraint. For example, for Numeric columns, the sum of Columns A and B must be less than 10. For Character columns, the concatenated values of Columns A and B must be unique. For Date columns, the time difference between Columns A and B cannot exceed one day.
- **Derived relation** emphasizes that values in specific columns are derived from other columns. For example, Column B values are derived from algebraic operations on Column A (e.g., $B = A + 2$).

3.3 Mapping Data Transformations to Table Constraints

To construct the input and output space for individual transformations, we analyzed the impact of transformations on constraints and built a mapping between them (see Sec. 3.1). For each transformation, we first identified the required columns in the input table and the constraints that these columns must satisfy for the transformation to be executed correctly. For example, with the R code `df2 = drop_na(df1, colA)`, df1 must contain column *colA* for the code to execute successfully. Additionally, based on the semantics of this code, missing values are allowed in *colA* of the input table (Constraint 1). To enrich the understanding of the input space, i.e., what input the code can handle, we introduced semantic constraints (like Constraint 1) that can reveal the semantics of the transformation to indicate the possibilities in the input table. Semantic constraints are optional and not mandatory for the input table to satisfy. During the construction of the mapping, we found some semantic constraints that we recommend satisfying to avoid potential errors. For example, in the code `tbl2 = filter(tbl1, colB>60)`, apart from the constraint that tbl1 must contain a column *colB* with a numeric data type (Constraint 2), we also include a semantic constraint that the values in *colB* may be > 60. However, if all values in *colB* of tbl1 are ≤ 60, the output table tbl2 will be empty, rendering subsequent transformations/analyses meaningless. Hence, we recommend that *colB* should have at least one value > 60 (Constraint 3). Constraints like this, which can affect the output results, are recommended to have some values in the input that satisfy the constraint, which can facilitate users in debugging potential issues in the output table. Furthermore, considering all the constraints of the input table, we analyzed the constraints that the output table would possess after the transformation. For example, in the aforementioned `drop_na` example, the output table df2 has a constraint that there are no missing values in *colA* (Constraint 4).

As shown above, constraints can be categorized into four scopes based on the required quantity of satisfying data values:

- **(A) all** values must satisfy the constraint (e.g., Constraint 2).
- **(N)** all values do **not** satisfy the constraint (e.g., Constraint 4).
- **(E) exist** values that satisfy the constraint (e.g., Constraint 3).
- **(O) optional** values satisfy the constraint (e.g., Constraint 1).

4 FERRY SYSTEM

4.1 Requirement Analysis

Our goal is to help data workers understand the input and output space of data wrangling scripts, facilitating script reuse, data onboarding, debugging, and maintenance. We interviewed two data scientists (E1-E2) and two data analysts (E3-E4) from a national research laboratory who frequently engage in writing and understanding data wrangling scripts on a daily basis. The interviews aimed to identify their needs, strategies, and challenges in understanding scripts inputs and outputs. They emphasized that understanding script inputs and outputs is essential for many tasks. E3 noted, *"When reusing a [data wrangling] script from the web, I need to first figure out if it can handle my data and how well the output matches my expectations before deciding to reuse it."* E1 added that using her data often led to runtime errors or abnormal output (e.g., NA values). Debugging these errors by analyzing the script is time-consuming. A tool that directly reveals script inputs and outputs, and explains failures or unexpected outputs, would greatly enhance their efficiency.

Inspired by our survey and analysis that constraints can express the input/output space of scripts (Sec. 3), we demonstrated examples of table space defined by constraints to our interviewees to verify whether this method could help their understanding of the input/output space. They unanimously agreed to that method, and E1 commented, *"Compared to individual data tables, table space constructed by constraints could reveal a more comprehensive picture of script inputs and outputs."* Additionally, they expressed a desire for the tool not only to reveal script inputs and outputs but also to facilitate their subsequent tasks. Following the guidelines proposed by Munzner [44], we iteratively extracted and summarized the following six design requirements through multiple rounds of discussions.

R1: Reveal input and output table space. Understanding the input and output of a script is crucial for data workers to complete various tasks. E4 remarked, *"Displaying the input a script can process, along with its potential output, significantly economizes time during script reuse."* Our interviewees recognized the use of constraints to reveal the input/output space. Consequently, the system needs to extract constraints and construct the table space.

R2: Explain table constraints. As table constraints are presented using symbols, formulas, or patterns (such as regular expressions), these representations are abstract to people. The situation gets tougher as a table space is usually constructed by multiple constraints. Users face challenges in understanding the overall effects of these constraints on the table. Therefore, a clear explanation of the constraints is required to facilitate comprehension.

R3: Support modification of constraints. Data workers desire to adjust table space if the script's input is inconsistent with their existing data or if the current output does not align with the requirements of downstream tasks. E2 stated, *"When looking for data trends, I prefer the output data not to contain missing values."* Therefore, the system should enable users to add, modify, or remove table constraints.

R4: Detect the conflict. Conflicts can uncover issues within scripts, as well as discrepancies between scripts and data, and between scripts and user requirements. There are two types of conflicts: (1) conflicts between constraints, due to script errors or user modifications clashing with existing constraints; and (2) conflicts between constraints and data, when current data and script constraints mismatch, possibly causing execution errors or unexpected output. Consequently, the system needs to detect these conflicts and elucidate their origins or the implicated data to users.

R5: Reveal associations between data columns. Disentangling column associations helps users understand how constraints derive from the input to output. When users modify output constraints, they need to determine which input column constraints are affected. Similarly, when user-uploaded data conflicts with an input constraint, users need to identify the affected output columns.

R6: Locate the origins of constraints. Locating the origins of constraints greatly aids users in understanding and debugging scripts. For example, onboarding new data can lead to conflicts with the script's existing constraints. Locating the source of the constraints can identify the causes of conflicts and the related code, facilitating the debugging process and completing downstream tasks.

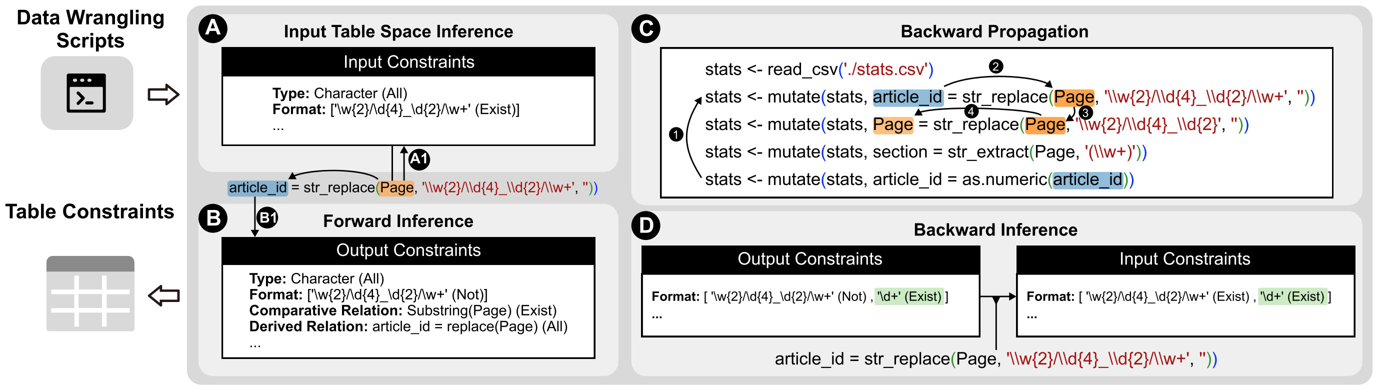
4.2 System Overview

In response to the above requirements, we designed and developed Ferry to help users understand the input/output table space of data wrangling scripts. Ferry encompasses three views (Fig. 1): **A)** The Script View contains a script editor showing data wrangling scripts and supports script modification and locating of the constraint origins (**R6**). A lineage graph at the top shows the association of data columns involved in the transformation (**R5**). Constraints for the input and output table space are generated after users upload scripts (**R1**). **B)** The Constraint View displays the input and output table space (**R1**) and sample data for each column (**R2**). Users can select tables for display and apply filters to refine constraint types and scopes. When users upload new data, any conflicts with the constraints are highlighted (**R4**). **C)** The Detail View provides in-depth information on each constraint within a data column. Users can access explanations and sample data for each constraint (**R2**). Conflicting entries between a constraint and uploaded data are presented (**R4**). The Detail View also provides origin tracing and allows for modifying constraints (**R3**).

4.3 Script View

The Script View (Fig. 1A) presents user-uploaded scripts (Fig. 1A1) and the lineage of data columns specifically involved in transformations (Fig. 1A2). This view displays a summary of data transformations, supports script modification, and illustrates the origins of constraints (**R6**). In the top-right corner of the Script View, distinct buttons are available for uploading user scripts and generating constraints for the current script. Ferry activates a constraint generator (Sec. 4.3.1) to produce the input and output table space (**R1**). It generates a node-link diagram that visualizes the lineage of the relevant data columns (**R5**).

Lineage Graph. Lineage Graph's nodes symbolize data columns, each with an icon denoting its data type. The edges between nodes represent the data transformations. As shown in Fig. 1A3, *State_Abbr* extracts data from *Origin* that matches the regular expression `_(\\w+)`. The data is merged with *Stu_Info* and *GPA* to update *Stu_Info*, which is sorted in descending order. If input constraint conflicts are detected upon data upload, the Lineage Graph will display a conflict icon on the



Input Constraints:
Type: Character (All)
Format: ['\w{2}/\d{4}_\d{2}/\w+' (Exist)]
...

```
article_id = str_replace(Page, '\\w{2}/\\d{4}_\\d{2}/\\w+', ''))
```

Output Constraints:
Type: Character (All)
Format: ['\w{2}/\d{4}_\d{2}/\w+' (Not)]
Comparative Relation: Substring(Page) (Exist)
Derived Relation: article_id = replace(Page) (All)
...

```
stats <- read_csv('./stats.csv')
stats <- mutate(stats, article_id = str_replace(Page, '\\w{2}/\\d{4}_\\d{2}/\\w+', ''))
stats <- mutate(stats, Page = str_replace(Page, '\\w{2}/\\d{4}_\\d{2}', ''))
stats <- mutate(stats, section = str_extract(Page, '(\\w+)'))
stats <- mutate(stats, article_id = as.numeric(article_id))
```

Output Constraints:
Format: ['\w{2}/\d{4}_\d{2}/\w+' (Not) , '\d+' (Exist)]
...

Input Constraints:
Format: ['\w{2}/\d{4}_\d{2}/\w+' (Exist) , '\d+' (Exist)]
...

```
article_id = str_replace(Page, '\\w{2}/\\d{4}_\\d{2}/\\w+', ''))
```

Fig. 2: Constraint Generator workflow. (A) Input Table Space Inference extracts information from code to generate constraints for the input; (B) Forward Inference uses code information to generate output constraints and transfer constraints from input to output; (C) Backward Propagation updated constraints from subsequent steps to preceding ones; (D) Backward Inference transfers constraints to the input from the output.

corresponding nodes. It will also highlight conflicting columns and their paths, indicating effects on related outputs. When output constraints are modified, the nodes and paths of affected input columns will be highlighted with color-coding. This color-coding reflects the scope of the modifications, thereby revealing the impacted input columns.

4.3.1 Constraint Generator

We designed and implemented a Constraint Generator that extracts information from the data wrangling scripts to construct the input and output table space. The Constraint Generator comprises four modules: **A)** Input Table Space Inference, generating input constraints for single-step data transformation; **B)** Forward Inference, generating output constraints for single-step data transformation; **C)** Backward Propagation, propagating updated constraints from subsequent steps to previous ones; and **D)** Backward Inference, which serves Backward Propagation by transferring updated constraints from output to input. A detailed description of these four modules is provided below:

Input Table Space Inference. Input table constraints are extracted based on the predefined mapping between data transformation types and constraints (Sec. 3.3). For instance, `str_replace(Page, '\\w{2}/\\d{4}_\\d{2}/\\w+', '')`, which implements a replace data transformation (Fig. 2A). According to the mapping, this transformation generates the input table's constraints (e.g., type and format). The content of these particular constraints is determined by the function utilized in the code and its corresponding parameters (Fig. 2A1).

Forward Inference. After constructing the input table space for a transformation step, we apply forward inference through the code to generate the output table space for the current step (Fig. 2B). In this step, following the established mapping (Sec. 3.3), we generate constraints for the output and propagate applicable input constraints to the output. Output constraints are generated based on data transformation and code parameters (e.g., format), while unaffected constraints (e.g., type) are propagated from input to output (Fig. 2B1).

Backward Propagation. In the first two steps, the Constraint Generator incrementally builds and expands the table space, which renders the constraints within the tables more specific. This specificity allows for the refinement of prior constraints. For instance, as shown in Fig. 2C, the *article_id* is converted from a character to a numeric type in the fifth step. This conversion generates a semantic constraint, as defined in Sec. 3.3, that requires at least one value in *article_id* to match a digital format, conforming to the regular expression "\d+". The input column *article_id* for this step is derived from the *Page* in the second step (Fig. 2C1). Since the constraints for *article_id* have been updated, the constraints for *Page* must be correspondingly updated (Fig. 2C2). After the *Page* is updated, the revised content is propagated through Forward Inference to subsequent steps (Fig. 2C3). In this code block, the *Page* is transformed to generate new columns, *Page* and *section* (Fig. 2C4). Backward Propagation is triggered when Input Table Space Inference updates the constraints of a derived column that does not exist in the input table. As depicted in Fig. 2C, updating the constraints of *article_id* by the Constraint Generator triggers. This process terminates when it updates columns in the original data table, exemplified by the Page column in Fig. 2C, or when updated constraints cannot be transferred to the input table space.

Backward Inference. During Backward Propagation, constraints from the output table space are transferred back to the input table space (Fig. 2C2). As illustrated in Fig. 2D, the Backward Propagation process updates the output (add the constraint `"\d+" (Exist)`) of the code: `article_id = str_replace(Page, '\\w{2}/\\d{4}_\\d{2}/\\w+', '')`. Backward Inference determines whether the updated constraints can be propagated to the input based on the transformation type and parameters. In this case, `"\d+"(Exist)` is transferable and is propagated to the input space.

Our prototype system currently supports 28 common R functions, such as `filter`, `drop_NA` and `substr` (see Sec. 2 of the supplementary material for details). When encountering unsupported functions, Ferry alerts the user and highlights the relevant code in the script, and only displays the input/output space results parsed up to that code.

4.4 Constraint View

The Constraint View (Fig. 1B) shows input and output table space constraints for the current script (**R2**). The Upload Data button enables users to upload data for onboarding. Ferry's Conflict Detector (Sec. 4.4.1) identifies conflicts between the uploaded data and existing constraints (**R4**); upon detection, the corresponding conflicting constraints are highlighted within the Constraint View. The view provides two filters enabling constraint selection and display by type and scope. The main area of the Constraint View contains two table space panels: by default, the upper panel displays the input, and the lower panel displays the output table space. The upper-left corner of each table space panel displays an input/output label, a source icon (file or variable), the table's name, and a table space selector. The upper-right corner contains a conflict menu to view all conflicts. Each column in the table space panel is split into two sections: the upper section displays constraints (Fig. 1B1), and the lower section presents sample data (Fig. 1B2) from Ferry's Sample Data Generator (Sec. 4.4.2) (**R2**).

Constraint Scope Color Encoding. Currently, constraints are categorized into four scopes (Sec. 3.3), each encoded with a distinct color: green for All (A), orange for Not (N), blue for Exist (E), and purple for Optional (O). Within the Constraint View, all constraint tags (see Fig. 1D) adhere to this color-coding scheme.

Table Space Panel. The table space panel displays the columns of the selected table space. Each column is divided into an upper section showing its constraints and a lower section displaying sample data consistent with those constraints. An icon that represents the semantic meaning of each constraint type is used to convey the distinct types of constraints. Icons for type, missing value, duplicate value, and order constraints appear at the column's top (see Fig. 1E) when their

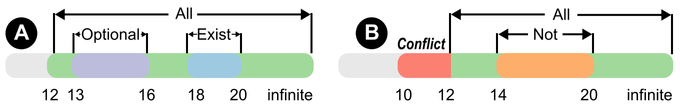


Fig. 3: In the table space panel, the Range component uses overlay bars to represent different scopes (A), allowing the representation of non-continuous ranges and data conflicts (B).

scopes are All, Exist, or Optional. An overlay bar chart visualizes range constraints, as illustrated in Fig. 3A. The base layer of the chart is gray, signifying the full potential span of the range (e.g., for Character type data, the range indicates the string length, with a minimum of 0 and a maximum of infinity). We stack bars of other colors (based on the previously defined scope color encoding) onto the base layer to represent different constraint ranges (as shown in Fig. 3A), including discontinuous ones. For example, in Fig. 3B, the green bar indicates that all data should fall within the range starting from 12, while the orange bar indicates that no data should exist between 14 and 20, resulting in a discontinuous range: from 12 to 14 and from 20 onwards. If data conflicts with the range constraint, a red bar appears (e.g., the bar with data ranging from 10 to 12 in Fig. 3B). Other types of constraints are represented with the constraint tag, as shown in Fig. 1D. An icon representing the constraint type is in a dark box on the left, while the right side shows the constraint content. The color of the constraint tags aligns with the previously defined scope color encoding. Uploaded data may conflict with existing constraints, which results in a conflict icon being displayed to the right of the constraint tag, as depicted in Fig. 1D. Within each column, constraint tags are arranged by scope, with tags representing constraints that affect a larger quantity of data placed in earlier positions (i.e., All, Not, Exist, and Optional). Conflicts, requiring user attention, are marked with red color encoding. Variations in red color encoding for conflicts are shown in different components, as illustrated in Fig. 1D and Fig. 1E. Furthermore, conflict-indicating tags are placed at the top of the column, regardless of their scope. Additionally, columns with conflicts are moved to the leftmost side of the panel, ensuring that users can readily spot them.

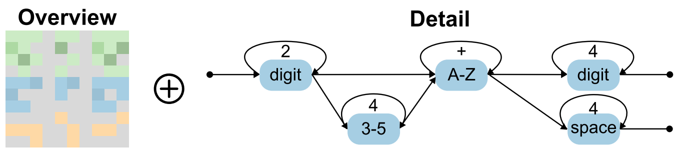


Fig. 4: Design alternatives for visualizing constraint's overview and detail.

Design alternative. During our iterative design process, we explored various approaches to visualizing constraints based on feedback from experts (E1-E4, detailed in Sec. 4.1). Initially, we followed the "overview to detail" design philosophy [55], creating representations for constraints at multiple levels of granularity, as illustrated in Fig. 4. For the overview, we experimented with a matrix in the table space panel, using colors to denote constraint types and gradients to indicate their quantities. However, our iterative evaluations, including testing and discussions with the experts, revealed that the matrix visualization did not sufficiently support deeper explorations based on constraint types or quantities within the table space. For the details, we considered methods such as visualizing regular expressions to help understand the constraints' content. Despite their potential to clarify complex concepts, expert preferences were strongly inclined towards directly viewing the constraints themselves, supplemented by sample data conforming to these constraints. Therefore, we shifted to a direct visualization approach (Fig. 1B1) supplemented with illustrative examples (Fig. 1B2).

Interactions. Users can upload data by clicking the Upload Data button at the top of the view and using the selector to determine the types and scopes of constraints for display in the table space panel. Within each column of the table space panel, clicking a column name displays all associated constraints in the Detail View, whereas clicking an individual icon or constraint tag shows details for that specific constraint. Clicking an icon or constraint tag locates the origins of the constraint in the Script View's script editor using color.

4.4.1 Conflict Detector

The Conflict Detector is designed to recognize the two categories of conflicts mentioned in the previous requirements (**R5**): **(A)** Conflicts between constraints and **(B)** Conflicts between constraints and data.

Conflicts between constraints. This type of conflict primarily encompasses two scenarios: First, conflicts among data transformations within the scripts indicate errors that occur during the scripting process. Second, conflicts caused by modifying constraints contradict existing ones. Detecting such script conflicts aids users in both identifying errors within the script and recognizing discrepancies between the script and current task requirements, thereby informing necessary script revisions.

Conflicts between constraints and data. Detecting this type of conflict identifies inconsistencies between user data and scripts, thereby detecting execution errors or potential data errors in the output. It also uncovers script deficiencies that fail to transform the input into the expected output. For instance, when a user modifies the output constraints to require a column containing no missing values, but the script does not execute a `drop_na` operation on that column. If the input contains missing values in that column, this leads to a conflict, signifying the script's current inability to process such missing values.

4.4.2 Sample Data Generator

One type of constraint can denote an attribute of data, such as range or format. Constraints enable data workers to efficiently extract attribute-specific information. However, when data workers need to understand table space, integrating different types of constraints becomes a difficult task, increasing their cognitive burden.

A Sample Data Generator was designed to address the challenges of integrating constraints for table space comprehension. Initially, we used only the Z3-solver [8, 41] to generate sample data. However, this data lacked semantic richness, impeding user comprehension. To enhance sample data generation, we employed GPT-4 [46] alongside the Z3-solver. Based on expert discussions, we chose the Z3-solver for precise numeric data generation and GPT-4 for generating semantically rich data. Constraints were translated into Z3-solver-compatible formats and GPT-4-suitable prompts. Experiments showed that the Z3-solver consistently produced stable data, while GPT-4's outputs occasionally violated constraints. To address this issue, prompts were restructured into subtasks and simpler constraints were used in a rules-based generation approach (e.g., duplicates, missing values, etc.). A validation process was implemented to ensure data accuracy by identifying and regenerating any samples that failed to meet the constraints.

4.5 Detail View

The Detail View (Fig. 1C) is designed to present comprehensive information on constraints (**R2**). Within the Constraint View, clicking a column name populates the Detail View with its constraints; clicking a constraint tag reveals details of that specific constraint. A button in the top-right corner lets users append new constraints to existing columns (**R3**). Detail View provides explanations and examples to elucidate constraints, enhancing user understanding. Detail View supports multiple operations on constraints, including editing content and labels (**R3**), tracing origins within scripts (**R5**), and deleting them. Should conflicts arise between user-uploaded data and constraints, the Detail View will display the conflict data (**R5**), facilitating the identification of causes.

Constraint information. When a user selects a constraint tag, the tag expands to reveal two primary information categories: (1) Explanation of constraint, and (2) Example of constraint. In the Explanation of constraint section, the system describes the constraint in natural language to assist users in understanding its content. Recognizing that some users might not easily understand natural language descriptions, we provide sample data in the Example of constraint section.

Conflict data. If the uploaded data conflicts with an existing constraint, the Conflict section appears in the expanded constraint tag. The Conflict section lists instances of uploaded data that fail to meet the constraint. Located below the Example of constraint, this section helps users compare discrepancies between the conflict and the example.

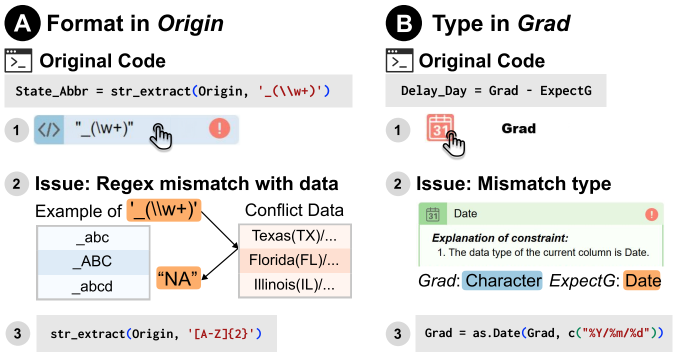
Constraint operation. In the expanded constraint tag's lower part, there are two sections: Modify constraint and Constraint operations. In the Modify constraint section, users can edit the current constraint's content and scope. Afterward, by clicking the Submit button in the Constraint operations section, Ferry will recalculate the modified constraints, update the table space, and assess whether the modification conflicts with other constraints. Additionally, the Delete button, located to the right of the Modify constraint section, allows users to remove the current constraint. Similar to modification, Ferry will update the table space and check for potential conflicts. Upon clicking the Position button on the left of the Constraint operations section, the system highlights the corresponding code within the Script View's script editor. A correlation is thus facilitated between constraints and their source scripts, enhancing understanding and facilitating code reuse.

5 Evaluation

This section demonstrates Ferry's effectiveness and usability through two usage scenarios, two case studies, and user interviews. Besides, an example application has been implemented to showcase its flexibility.

5.1 Usage Scenarios

5.1.1 Onboarding New Data Into an Existing Script



```
State_Abbr = str_extract(Origin, '_(\\w+)')
```

```
"_(\w+)"
```

```
str_extract(Origin, '[A-Z]{2}')
```

```
Delay_Day = Grad - ExpectG
```

Explanation of constraint:
1. The data type of the current column is Date.

```
Grad = as.Date(Grad, c("%Y/%m/%d"))
```

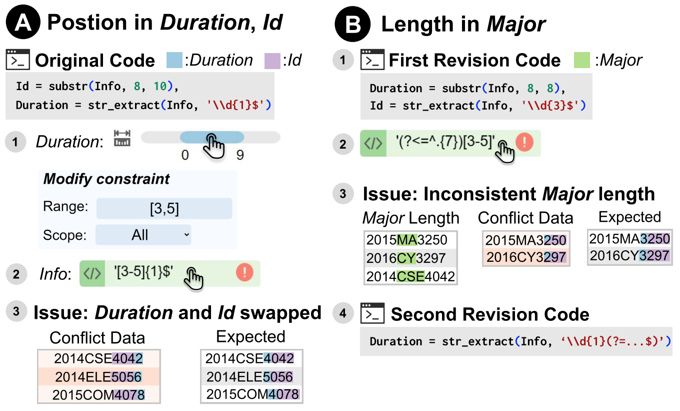
Fig. 5: Two initial issues presented by Ferry: (A) Data in *Origin* not matching the regular expression for extraction; (B) Data type mismatch between *Grad* and *ExpectG*.

In this scenario, Windy, a data analyst, was tasked with preprocessing data exported from a student information system for follow-up analysis. However, due to recent system updates that caused changes in the data, she encountered issues when directly onboarding the new data into an existing data wrangling script. This scenario demonstrates how Windy used Ferry to resolve these issues and modify the original script.

Windy uploaded the script to Ferry and clicked the Run button. The Constraint View represented the input/output space of the current script. Upon uploading the data, she identified the following two conflicts (Fig. 5A1 and B1) within the Constraint View.

Format Issue in *Origin*. Windy first investigated the format conflict issue in the *Origin*. After she clicked on the corresponding constraint tag (Fig. 5A1), the script editor highlighted the conflicting code (see Original Code in Fig. 5A) and the Detail View presented example data that both comply with and conflict with the constraint (Fig. 5A2). Windy discovered a mismatch between the regular expression and the *Origin's* value, causing "NA" in the output. She observed that the conflict's propagation on the Lineage Graph (Fig. 1A1) could compromise the output *Stu_Info*'s value. Thus, resolving the conflict was crucial. Based on the format characteristics of the new data, Windy revised the extraction rule of the code (Fig. 5A3).

Type Issue in *Grad*. Subsequently, Windy clicked the red icon (Fig. 5B1) in *Grad*. The Detail View showed that its expected data type should be Date, while it currently was Character (see Issue in Fig. 5B2). This is because calculations with *ExpectG* (Date type) require *Grad*



```
Id = substr(Info, 8, 10),
Duration = str_extract(Info, '\\d{1}$')
```

```
'[3-5]{1}$'
```

```
Duration = substr(Info, 8, 8),
Id = str_extract(Info, '\\d{3}$')
```

```
'(?<=^.{7})[3-5]'
```

```
Duration = str_extract(Info, '\\d{1}(?=...$)')
```

Fig. 6: Two additional issues after modification: (A) incorrect extraction positions for *Duration* and *Id*; (B) inconsistent string length for *Major*.

also to be Date. Therefore, Windy added the type conversion code (Fig. 5B3) before the original code in the script.

After making these modifications, Windy clicked the Run button. Ferry generated new conflict-free constraints, indicating that the previous issues had been resolved. Windy then examined these new constraints with their sample data to verify the absence of further issues.

Extraction Issue in *Duration*, *Id*. Windy found that the Range of *Duration* was [0-9], which did not meet the expected duration of 3 to 5 years for a student program. She then clicked the Range constraint of *Duration*, modified the Range in the Detail View to [3-5], and changed the scope to All (Fig. 6A1), ensuring all data in *Duration* conformed to this range. This modification caused a chain effect in the Format constraint of *Info*, which replaced '`\d{1}$`' with '`[3-5]{1}$`', bringing a new conflict with data (Fig. 6A2). Windy clicked the constraint tag to review the conflict data in the Detail View (Fig. 6A3). She found the last digits in *Info* did not fall within the range [3-5], but the fourth-to-last digits did. She realized *Info*'s organization had changed: the student's *Id* (originally in the 8th-10th positions) and the *Duration* (originally the last digit) had been swapped (see Issue in Fig. 6A3). Therefore, the original code for generating the *Id* and *Duration* (see Original Code in Fig. 6A) needed to be modified (Fig. 6B1).

Length Issue in *Major*. After re-executing the modified script, Windy observed a new Format conflict in *Info* due to the regex pattern "`(?<=^.{7})[3-5]`", which required the eighth character to be a digit between 3 and 5 (Fig. 6B2). Upon further review of the conflict data in the Detail View, she found that discrepancies in the length of *Major* caused the issue with the expected position of *Duration* (see Issue in Fig. 6B3). While most *Major* entries comprised three characters, the presence of two-character entries (Fig. 6B3) led to inaccuracies in position-based extraction. To rectify this, Windy modified the extraction code to target the fourth-to-last digits (Fig. 6B4). After rerunning the script, no conflicts occurred. Windy further verified and confirmed that the current script met the requirements.

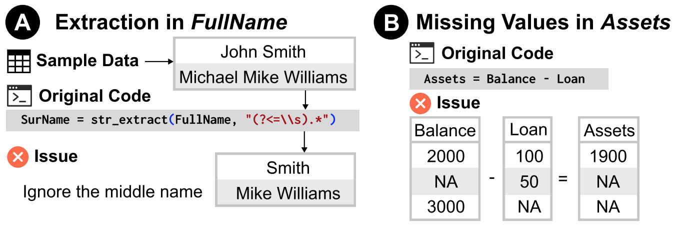
We utilized Unravel [56] to replicate the same usage scenario for comparison with Ferry (see Sec. 3 of the supplementary material for details). This usage scenario highlights Ferry's capabilities in data onboarding and script debugging, focusing on two key aspects:

- Ferry enhances debugging efficiency by identifying error causes and locating them in the script. It reduces users' effort in manually inferring errors between data and code, such as format errors in *Origin* and type errors in *Grad* (Fig. 5A and B).
- Ferry aids users in uncovering latent errors in the script, such as *Duration* extraction necessitating reverse positioning (Fig. 6A), ensuring comprehensive and accurate debugging.

5.1.2 Debugging Scripts Generated by LLMs

This section outlines how a data worker uses Ferry to debug a script without data access. Leo, a bank data worker, was tasked with developing a processing module for customers' account balances and investment information. This module, as an integral part of the bank's data processing framework, primarily involves three data transformation tasks: (1) Extract the *SurName* from the *FullName*; (2) Generate the *RegDate* from the *RegYearMonth* and *RegDay*; and (3) Generate the *Assets* from the *Balance* and *Loan*. To maintain consistency, the module must be implemented in R using the tidyverse library.

With the rapid development of LLMs (e.g., GPT-4), powerful code generation has become possible. To reduce workload, Leo decided to use GPT-4 to generate the script that meets the task requirements. Due to the sensitivity of the data, Leo could neither directly access it for debugging nor upload a sample to GPT. He only provided the task requirements and crafted a prompt to generate the R script. However, LLM-generated code may contain errors, requiring thorough testing before deployment. Therefore, Leo used Ferry to examine the generated script and reviewed the constraints and sample data for each column.



John Smith
Michael Mike Williams

```
SurName = str_extract(FullName, "(?<=\\s).*")
```

Smith
Mike Williams

```
Assets = Balance - Loan
```

Balance		Loan		Assets
2000		100		1900
NA	-	50	=	NA
3000		NA		NA

Fig. 7: Issues in the second usage scenario: (A) Ignoring middle names when extracting *Surname* from *FullName*; (B) Missing values in *Assets* due to a calculation involving NA values.

Extraction Issue in *FullName*. He first identified errors in the sample data for the *SurName*, which is intended to represent the user's last name. However, entries such as "Mike Williams" were found (see Issue in Fig. 7A). He clicked the constraint tag in *SurName*. Ferry highlighted the relevant code (see Original Code in Fig. 7A). After reviewing the code, he realized that the script failed to account for middle names, instead extracting all characters following the first space in *FullName* as *SurName*. Leo modified the extraction rule to retrieve the last word from *FullName* using the regex `"\w+$"`. After re-executing the script, he observed that the sample data in *SurName* was displayed correctly.

Missing Values in *Assets*. Leo then reviewed the missing values in the output table space panel. He noticed the missing value icon in *Assets*. Upon clicking the column, he found in the Detail View that the scope of the missing value constraint was set to Optional. According to the task requirements, *Assets* is calculated by subtracting the *Loan* amount from the account *Balance* and should not contain missing values. To identify the causes of such missing values, he clicked the Position button in the Detail View, and the Origin Code in Fig. 7B was highlighted. Leo knew that in R, any numerical calculation involving NA results in NA (see Issue in Fig. 7B). Thus, missing values in *Balance* and *Loan* (indicating no balance/loan) should be replaced with 0 before calculation.

This scenario illustrates Ferry's ability to help users understand a script's input/output space without executable data. It assists users in identifying hidden errors, debugging the script, and enhancing its robustness by providing corner cases and constraints:

- Ferry generates comprehensive sample data using column name semantics and constraints to help users uncover hidden errors.
- Ferry's constraints help users identify corner cases, such as missing values in *Assets*, thereby enhancing script robustness.

5.2 Case Studies

To further evaluate the effectiveness and usability of Ferry, we recruited four experienced data practitioners (P1-P4) and conducted two case studies with them. P1, P2, and P3 are PhD students in data science, whereas P4 is a professional data analyst. Each has at least three years of experience in data wrangling, while exhibits varied proficiency in R: P1 and P3 are highly proficient, P2 and P4 are familiar but primarily use Python and SQL. During the studies, each practitioner first received a 15-minute training session covering the system's features, instructions on interpreting views, and a basic training case. Then, they were asked to independently complete the two cases, each taking approximately 30 minutes. Finally, the interviews (Sec. 5.3) were conducted to collect feedback on Ferry.

The first case required practitioners to complete the same task as described in the first usage scenario (Sec. 5.1.1). We introduce the second case in this section about a code verification task without available data provided. It demonstrates how Ferry assists data practitioners in verifying input-output compatibility between two modules and guides them to make necessary adjustments.

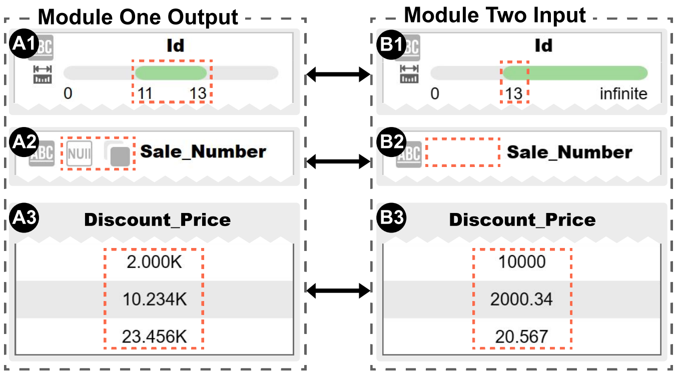


Fig. 8: Mismatches between the output of Module One and the input of Module Two: (1) Inconsistent *Id* string lengths; (2) Missing values in *Sale_Number*; (3) Discrepancies in the units of *Discount_Price*.

In the second case, data practitioners were tasked with verifying two sales data processing modules. Module One processes sales data exported from the system into a manipulable format. Module Two computes and transforms the data for visual analytics. To ensure module compatibility, practitioners needed to confirm that the output from Module One aligned with the input requirements of Module Two.

After uploading the scripts of both data processing modules to Ferry, practitioners focused on the Constraint View. They adjusted Module Two's output constraints to exclude missing values, adhering to downstream task requirements. Then, they used the table space panel's filter to display Module One's output on the upper panel and Module Two's input on the lower panel for direct comparison. Subsequently, practitioners verified the compatibility between module inputs and outputs by examining constraints and sample data.

In the Constraint View, Module One's *Id* string length was constrained to [11-13] (Fig. 8A1), while Module Two required a minimum of 13 characters for the same column (Fig. 8B1). Consequently, only 13-character *Id* outputs from Module One met Module Two's requirements. Practitioners traced the origin of the *Id* constraint in Module One by expanding the 'Minimum Length: 11' constraint in the Detail View and clicking the Position button. In the script editor of the Script View, the code `Id = str_extract(Id,'\\w{3}(\\w{2})?\\d{8}')` was highlighted. Practitioners used the same method to examine why Module Two required the *Id* to have a minimum length of 13. They found that `Product_Id = substr(Id, 6, 13)` failed to accommodate *Id* columns of only 11 characters. The *Id* data extraction method required modification to not solely rely on character position, such as: `Product_Id = str_extract(Id, '\\d{8}$')`.

An inconsistency arose in *Sale_Number*: Module One's output featured icons for missing and duplicate values (Fig. 8A2), which were absent in Module Two's input (Fig. 8B2). Based on the task requirements, *Sale_Number* could contain duplicate values, but missing values were unacceptable. Practitioners noted in the Detail View that the current scope was labeled as Optional, indicating possible missing values. After reviewing the highlighted code in the script editor, they identified that executing the separate operation introduced missing values (e.g., separating 'ABC/' by '/' leaves the second column empty). Thus, to guarantee no missing values in the final output, they recommended a `Drop_NA` operation for this column: `drop_na(Sale_Number)`.

Practitioners examined Module One's *Discount_Price* output with sample data like 10.123K (Fig. 8A3), whereas Module Two's input required values without the 'K' suffix (Fig. 8B3). Despite Module One's output seeming compatible with Module Two, practitioners reviewed the script for quality assurance. Discovering that both modules erroneously divided *Discount_Price* by 1000 and compromised accuracy, practitioners removed the redundant division operation from Module Two. Ultimately, under Ferry's guidance, practitioners identified the causes of the mismatches between the modules.

5.3 User Interviews

After the case studies, we conducted semi-structured interviews to collect feedback and suggestions on Ferry. All practitioners appreciated the design of Ferry, including the design within the Constraint View and the constraint information and operations provided in the Detail View. Related to daily practice, P2 noted, *"I frequently reuse scripts from code repositories. Ferry enables me to generate the input and output for those scripts that lack associated data, which assists in verifying the scripts' compliance with my requirements."* P3 acknowledged that Ferry facilitates onboarding and debugging new datasets: *"Ferry identifies mismatches between my data and the script, it indicates the locations of conflicts within the script, thereby improving code modification efficiency."* P3 and P4 reported that Ferry enables more comprehensive debugging of scripts and aids in discovering hidden errors. P1 liked the Lineage Graph, noting, *"The lineage graph delineates the pathways influenced by conflicts, which helps me find which columns may contain errors without reading the script back and forth."*

In addition, practitioners provided constructive feedback on Ferry, suggesting enhancements to further improve the system. (1) P1 and P3 observed that multiple constraints in a single column can overburden readers. While sample data helps with understanding, data profiling could familiarize users with the table upfront. P1 suggested improving constraint-data interaction to aid comprehension. (2) P2 observed that the detailed column headers in the Constraint View can be cumbersome for multi-column tables. An overview method in the Constraint View could narrow the header space of data columns, thereby enhancing readability. (3) P4 mentioned the case of verifying the compatibility between two data modules. He proposed interactive mapping or highlighting between outputs and inputs to boost reading efficiency. (4) All practitioners reported that learning to modify constraints, particularly regular expressions, is challenging for those inexperienced with regex.

5.4 Example Application

This subsection demonstrates how Ferry can be integrated into another visualization system to aid in understanding the transformation semantics of data wrangling scripts. Existing efforts [51, 67] in understanding data wrangling scripts often rely on executable data. For instance, SOMNUS [67] is a script visualization system that generates corresponding glyphs for each transformation based on input data to depict its semantics. However, the effectiveness of such glyphs would be compromised when the input lacks data that reflects the transformation semantics. For example, in Fig. 9A, the input table of the filter transformation does not have counterexamples to the filtering condition (i.e., `colB>60`), failing to manifest the semantics of deleting rows. Moreover, glyphs cannot

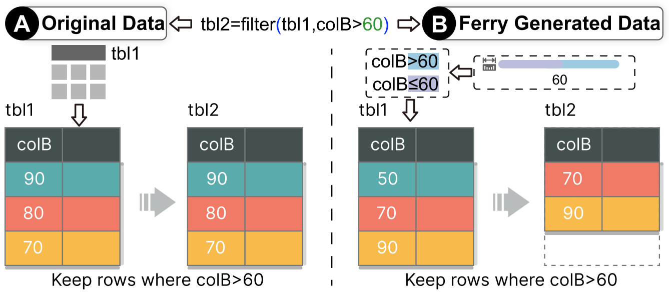


Fig. 9: The original data in SOMNUS can not reveal semantics (A), while data generated by Ferry can reveal semantics (B).

be generated when input data is unavailable or when runtime errors occur. To address these limitations, we integrated Ferry into SOMNUS by automatically generating suitable inputs to augment its capability. Specifically, we modify the mapping summarized in Sec. 3.3 by setting all semantic constraints of transformations as the Exist label and utilize the synthetic data generated by the Sample Data Generator (Sec. 4.4.2) as the input for the script. Fig. 9B illustrates an example of glyphs generated by SOMNUS before and after integrating Ferry.

6 Discussion

Evaluation: In this paper, we utilize ten types of table constraints to express script input/output space. While we verified these constraints and their mappings to transformations on a real-world dataset [68], the extent to which the constraints generated by Ferry can capture the scope of input/output and meet the needs of data workers in various downstream tasks requires further validation. In future work, we plan to recruit more users and design more diverse tasks to conduct comprehensive studies.

Scalability: Our constraint generation model is rule-based and customized for transformation code. Currently, Ferry's prototype supports 28 common functions from the tidyverse library in R. However, implementations of data transformations vary significantly across programming languages, and extending our work to various codes presents a tedious and labor-intensive challenge. COMANTICS [68] leverages an AI model to automatically infer the semantics of data wrangling code but requires executable input data. In the future, we will explore using large language models for static semantics inference and constraint extraction with sample data validation.

Capability: Ferry's capability is limited in two aspects. First, Ferry progressively enriches the constraints of input/output tables through step-by-step transformation parsing, yielding increasingly clear table space. However, some transformations hinder the accumulation and propagation of constraints. For example, fold/unfold can alter a table's schema, and summarize operations will combine rows by aggregation, both of which renders previously inferred constraints invalid. Second, Ferry can identify the causes and locations of conflicts within scripts. However, it currently lacks the capability to recommend code modifications to users. Future iterations will focus on developing a human-in-the-loop technique that integrates machine insights with human expertise and user intent to enhance Ferry's usability.

Opportunity: In addition to providing explanatory input data to convey transformation semantics (see Sec. 5.4), Ferry has potential applications in other scenarios. For example, Ferry could be extended to support script annotation to foster comprehension and collaboration. While it is common for programmers to annotate a function's input/output, specifying the input/output tables, which are 2-D data structures with multiple rows and columns, for a data wrangling module is not as straightforward. To bridge this gap, text comments or documentation describing the input/output of data wrangling modules can be generated based on the table constraints obtained from Ferry.

7 Conclusion

In this work, we present Ferry, an interactive visual analysis system that reveals the input/output space of data wrangling scripts. Ferry assists users in comprehending the space through three coordinated views that visualize the input/output space along with sample data. The system uses a constraint generation model, informed by a mapping between data transformations and ten types of constraints for expressing the table space. Our evaluation demonstrates that Ferry is effective in facilitating the understanding of the script input/output and completing various tasks, such as onboarding data, debugging, and validating scripts.

8 Acknowledgments

The work was supported by National Key R&D Program of China (2022YFE0137800), Key "Pioneer" R&D Projects of Zhejiang Province (2023C01120) , NSFC (U22A2032), and the Collaborative Innovation Center of Artificial Intelligence by MOE and Zhejiang Provincial Government (ZJU). We gratefully thank the anonymous reviewers for their valuable comments.

References

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

[2] R. Abreu, B. Hofer, A. Perez, and F. Wotawa. Using constraints to diagnose faulty spreadsheets. *Software Quality Journal*, 23(2):297–322, 2015. 2, 3

[3] R. Barták. Constraint programming: What is behind. *Proceedings of CPDC99*, pages 7–15, 1999. 3

[4] A. Bigelow, C. Nobre, M. Meyer, and A. Lex. Origraph: Interactive network wrangling. In *Proceedings of IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 81–92, 2019. 2

[5] R. Chen, D. Weng, Y. Huang, X. Shu, J. Zhou, G. Sun, and Y. Wu. Rigel: Transforming tabular data by declarative mapping. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):128–138, 2023. 2

[6] T. Dasu and T. Johnson. *Exploratory Data Mining and Data Cleaning*. John Wiley, 2003. 2

[7] C. de la Riva, M. J. Suárez-Cabal, and J. Tuya. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 67–74, 2010. 3

[8] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008. 2, 3, 6

[9] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. *ACM SIGPLAN Notices*, 38(11):78–95, 2003. 3

[10] P. Deransart, M. V. Hermenegildo, and J. Maluszynski. Analysis and visualization tools for constraint programming, constrain debugging (discipl project). volume 1870. Springer, 2000. 3

[11] G. Dooms, P. Van Hentenryck, and L. Michel. Model-driven visualizations of constraint-based local search. *Constraints*, 14(3):294–324, 2009. 3

[12] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. Gulwani. Wrex: A unified programming-by-example interaction for synthesizing readable code for data scientists. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2020. 2

[13] B. Elkarablieh and S. Khurshid. Juzi: A tool for repairing complex data structures. In *Proceedings of the 30th International Conference on Software Engineering*, pages 855–858, 2008. 3

[14] M. Excel. Microsoft Excel Spreadsheet Software. https://www.microsoft.com/en-us/microsoft-365/excel. 2

[15] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Transactions on Software Engineering and Methodology*, 30(2):1–27, 2021. 3

[16] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014. 3

[17] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Communications*, 24(2):107–124, 2011. 3

[18] M. Ghoniem, H. Cambazard, J.-D. Fekete, and N. Jussien. Peeking in solver strategies using explanations visualization of dynamic graphs for constraint programming. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–36, 2005. 3

[19] M. Ghoniem, N. Jussien, and J. Fekete. VISEXP: visualizing constraint solver dynamics using explanations. In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference, Miami Beach, Florida, USA*, pages 263–268, 2004. 3

[20] S. Goodwin, C. Mears, T. Dwyer, M. G. de la Banda, G. Tack, and M. Wallace. What do Constraint Programming Users Want to See? Exploring the Role of Visualisation in Profiling of Models and Search. *IEEE Transactions on Visualization and Computer Graphics*, 23(1):281–290, 2017. 3

[21] P. J. Guo, S. Kandel, J. M. Hellerstein, and J. Heer. Proactive wrangling: Mixed-initiative end-user programming of data transformation scripts. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, pages 65–74, 2011. 2, 3

[22] B. Haugen and J. Kurzak. Search Space Pruning Constraints Visualization. In *Proceedings of the Second IEEE Working Conference on Software Visualization*, pages 30–39, 2014. 3

[23] J. Heer and A. Perer. Orion: A system for modeling, transformation and visualization of multidimensional heterogeneous networks. *Information Visualization*, 13(2):111–133, 2014. 2

[24] B. Hofer, A. Riboira, F. Wotawa, R. Abreu, and E. Getzner. On the Empirical Evaluation of Fault Localization Techniques for Spreadsheets. In *Proceedings of the Fundamental Approaches to Software Engineering*, pages 68–82, 2013. 3

[25] Y. T. Hu, M. Burch, and H. van de Wetering. Visualizing dynamic data with heat triangles. *Journal of Visualization*, 25(1):15–29, 2022. 2

[26] Y. Huang, Y. Zhou, R. Chen, C. Pan, X. Shu, D. Weng, and Y. Wu. Interactive Table Synthesis with Natural Language. *IEEE Transactions on Visualization and Computer Graphics*, To appear. 2

[27] J. P. Inala and R. Singh. WebRelate: Integrating web data with spreadsheets using examples. In *Proceedings of the ACM on Programming Languages*, pages 1–28, 2017. 2

[28] Z. Jin, M. R. Anderson, M. Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *Proceedings of the ACM International Conference on Management of Data*, pages 683–698, 2017. 2

[29] S. Kandel, J. Heer, C. Plaisant, J. Kennedy, F. van Ham, N. H. Riche, C. E. Weaver, B. Lee, D. Brodbeck, and P. Buono. Research directions in data wrangling: Visualizations and transformations for usable and credible data. *Information Visualization*, 10(4):271–288, 2011. 1

[30] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 3363–3372, 2011. 2

[31] S. Kasica, C. Berret, and T. Munzner. Table scraps: An actionable framework for multi-table data wrangling from an artifact study of computational journalism. *IEEE Transactions on Visualization and Computer Graphics*, 27(2):957–966, 2021. 2, 3

[32] S. Lau, S. S. Srinivasa Ragavan, K. Milne, T. Barik, and A. Sarkar. TweakIt: Supporting End-User programmers who transmogrify code. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–12, 2021. 2

[33] D. M. Le, H. Lee, K. C. Kang, and L. Keun. Validating Consistency between a Feature Model and Its Implementation. In *Proceedings of the Safe and Secure Software Reuse*, pages 1–16, 2013. 3

[34] J. Lin, J. Wong, J. Nichols, A. Cypher, and T. A. Lau. End-user programming of mashups with vegemite. In *Proceedings of the 14th International Conference on Intelligent User Interfaces*, pages 97–106, 2009. 2

[35] S. Liu, D. Peng, H. Zhu, X. Wen, X. Zhang, Z. Zhou, and M. Zhu. MulUBA: Multi-level visual analytics of user behaviors for improving online shopping advertising. *Journal of Visualization*, 24(6):1287–1301, 2021. 2

[36] Z. Liu, S. B. Navathe, and J. T. Stasko. Network-based visual analysis of tabular data. In *Proceedings of IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 41–50, 2011. 2

[37] Z. H. Liu, B. Hammerschmidt, and D. McMahon. JSON data management: Supporting schema-less development in RDBMS. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1247–1258, 2014. 2

[38] M. Z. Malik, J. H. Siddiqui, and S. Khurshid. Constraint-based program debugging using data structure repair. In *Proceedings of the Verification and Validation Fourth IEEE International Conference on Software Testing*, pages 190–199, 2011. 2, 3

[39] M. Marcozzi, W. Vanhoof, and J.-L. Hainaut. Test input generation for database programs using relational constraints. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, pages 1–6, 2012. 2, 3

[40] H. Mei, H. Guan, C. Xin, X. Wen, and W. Chen. DataV: Data visualization on large high-resolution displays. *Visual Informatics*, 4(3):12–23, 2020. 2

[41] F. Mora, M. Berzish, M. Kulczynski, D. Nowotka, and V. Ganesh. Z3str4: A Multi-armed String Solver. In *Proceedings of the Formal Methods*, pages 389–406, 2021. 3, 6

[42] J. Morcos, Z. Abedjan, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. DataXFormer: An interactive data transformation tool. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 883–888, 2015. 2

[43] D. Moritz, C. Wang, G. L. Nelson, H. Lin, A. M. Smith, B. Howe, and J. Heer. Formalizing Visualization Design Knowledge as Constraints: Actionable and Extensible Models in Draco. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):438–448, 2019. 3

[44] T. Munzner. A Nested Model for Visualization Design and Validation. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):921–928, 2009. 4

[45] S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering*, 41(8):820–841, 2015. 3

[46] OpenAI. GPT-4. https://openai.com/gpt-4. 6

[47] OpenRefine. Openrefine (previously google refine). https://openrefine.org/, 2023. (Retrieved: Aug 20th, 2023). 2

[48] M. Paltrinieri. A visual constraint-programming environment. In *Proceedings of the Principles and Practice of Constraint Programming*, pages 499–514, 1995. 3

[49] T. pandas development team. pandas-dev/pandas: Pandas, Feb. 2020. 2

[50] N. Piccolotto, M. Bögl, T. Gschwandtner, C. Muehlmann, K. Nordhausen, P. Filzmoser, and S. Miksch. TBSSvis: Visual analytics for Temporal Blind Source Separation. *Visual Informatics*, 6(4):51–66, 2022. 3

[51] X. Pu, S. Kross, J. M. Hofman, and D. G. Goldstein. Datamations: Animated explanations of data analysis pipelines. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–14, 2021. 1, 2, 9

[52] Z. Qu and J. Hullman. Keeping Multiple Views Consistent: Constraints, Validations, and Exceptions in Visualization Authoring. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):468–477, 2018. 3

[53] 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. 1

[54] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 461–470, 2011. 3

[55] B. Shneiderman. The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 336–343, 1996. 6

[56] N. Shrestha, T. Barik, and C. Parnin. Unravel: A fluent code explorer for data wrangling. In *Proceedings of the 34th Annual ACM Symposium on User Interface Software and Technology*, pages 198–207, 2021. 2, 7

[57] H. Simonis, P. Davern, J. Feldman, D. Mehta, L. Quesada, and M. Carlsson. A Generic Visualization Platform for CP. In *Proceedings of the Principles and Practice of Constraint Programming*, pages 460–474, 2010. 3

[58] K. Su, J. Zhang, D. Xie, and J. Tao. Importance guided stream surface generation and feature exploration. *Visual Informatics*, 7(2):54–63, 2023. 3

[59] C. Sutton, T. Hobson, J. Geddes, and R. Caruana. Data diff: Interpretable, executable summaries of changes in distributions for data wrangling. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2279–2288, 2018. 2

[60] Tableau. Tableau: Business Intelligence and Analytics Software. https://www.tableau.com. 2

[61] S. Takahashi. Visualizing constraints in visualization rules. In *Proceedings of the CP2000 Workshop on Analysis and Visualization of Constraint Programs and Solvers*, 2000. 3

[62] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems*, pages 47–60, 2011. 3

[63] C. Tominski, G. Andrienko, N. Andrienko, S. Bleisch, S. I. Fabrikant, E. Mayr, S. Miksch, M. Pohl, and A. Skupin. Toward flexible visual analytics augmented through smooth display transitions. *Visual Informatics*, 5(3):28–38, 2021. 2

[64] Trifacta. The Trifacta Data Engineering Cloud. https://www.trifacta.com/. 2, 3

[65] C. Wang, J. Thompson, and B. Lee. Data formulator: AI-Powered Concept-Driven Visualization Authoring. *IEEE Transactions on Visualization and Computer Graphics*, 30(1):1128–1138, 2024. 2

[66] H. Wickham, M. Averick, J. Bryan, W. Chang, L. D. McGowan, R. François, G. Grolemund, A. Hayes, L. Henry, J. Hester, M. Kuhn, T. L. Pedersen, E. Miller, S. M. Bache, K. Müller, J. Ooms, D. Robinson, D. P. Seidel, V. Spinu, K. Takahashi, D. Vaughan, C. Wilke, K. Woo, and H. Yutani. Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686, 2019. 1, 2

[67] K. Xiong, S. Fu, G. Ding, Z. Luo, R. Yu, W. Chen, H. Bao, and Y. Wu. Visualizing the Scripts of Data Wrangling With Somnus. *IEEE Transactions on Visualization and Computer Graphics*, 29(6):2950–2964, 2023. 1, 2, 9

[68] K. Xiong, Z. Luo, S. Fu, Y. Wang, M. Xu, and Y. Wu. Revealing the Semantics of Data Wrangling Scripts With Comantics. *IEEE Transactions on Visualization and Computer Graphics*, 29(1):117–127, 2023. 1, 2, 3, 9

[69] K. Xiong, X. Xu, S. Fu, D. Weng, Y. Wang, and Y. Wu. JsonCurer: Data Quality Management for JSON Based on an Aggregated Schema. *IEEE Transactions on Visualization and Computer Graphics*, 30(6):3008–3021, 2024. 2

[70] C. Yang, S. Zhou, J. L. Guo, and C. Kästner. Subtle bugs everywhere: Generating documentation for data wrangling code. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 304–316, 2021. 2

[71] J. Yang, Y. He, and S. Chaudhuri. Auto-pipeline: Synthesize data pipelines by-target using reinforcement learning and search. *Proceedings of the VLDB Endowment*, 14(11):2563–2575, 2021. 2

[72] Y. Ye, J. Hao, Y. Hou, Z. Wang, S. Xiao, Y. Luo, and W. Zeng. Generative AI for visualization: State of the art and future directions. *Visual Informatics*, 8(2):43–66, 2024. 3

[73] X. Zhu, M. A. Nacenta, Ö. Akgün, and D. Zenkovitch. Solvi: A visual constraint modeling tool. *Journal of Computer Languages*, 78:101242, 2024. 3